sparrow-lang Documentation

Release 0.10.33

Lucian Radu Teodorescu

Dec 01, 2019

Contents:

CHAPTER 1

Introduction

1.1 Mission

To be a general-purpose programming language that integrates efficiency, flexibility and naturalness.

1.2 Vision and goals

Sparrow's vision is to help programmers enjoy the act of programming; the difficulty of a programming task should be *equal* to the difficulty of the problem. Not simpler, not harder. If programming is simpler than the problem itself, then the programmer loses control. And, on the long term, this has negative consequences on the programmers and their results.

1.2.1 General-purpose programming language

A true general-purpose programming language is a language that is **good** at solving a large variety of programming problems. It shall be good at system programming, game programming, programming web servers and UIs. Moreover, it shall be able to adapt to new domains.

There are just a few programming languages that fit into this category. From the list of languages that are considered general-purpose, exclude all dynamic languages, all languages that use some kind of virtual-machine and all languages that have garbage collection.

Aside for the true general-purpose programming languages, we have a lot of languages that are somehow specialized in various domains (even if the languages are not domain-specific *per se*). Considering the high pace at which the domains of interest are changing in the IT industry, fundamental concepts of these programming languages are slowly becoming obsolete. Therefore, programmers will start using the wrong tools for the job.

Sparrow aims at being such a general-purpose programming language, that can be used to solve a large variety of software.

1.2.2 Efficiency

Efficiency is a major concerns for most of the software systems, at least the large ones. We all have the frustrations of software being too slow; slower than it used to be. There are domain fields in which performance is crucial (e.g., games).

Programming languages shall allow the programmers to write efficient code. That is, they shall enable two things:

- allow programmers to write efficient code
- allow programmers to reason about how high-level code is translated into low-level constructs

For the first point, a programming language shall be able to generate code that is similar to the code generated by a C program. All the abstractions that the language uses by default shall have zero efficiency costs.

The second point is more important, and often overlooked. A programmer cannot write efficient code without being able to reason about the efficiency of the written code, even in the presence of multiple abstraction layers. Take for example C++ code written with a lot of STL and Boost templates. The user can dwell into each abstraction layer and reason about the performance impact of it. One good example is Boost.GIL; although there are a lot of abstraction layers, the compiler can generate efficient code and moreover, the user has the tools to reason about the performance of all these layers.

That is an important part of the Sparrow vision: let the programmer write high-level (natural) code, that would generate efficient binary code. Efficiency by default.

1.2.3 Naturalness

Programmers are the main bottleneck for software development. They have to understand the system and all its details. Pure complexity.

The main job of a programming language is to help the programmer deal with that complexity. It shall make it easier for the programmer to understand and reason about the code. The programmer shall have means for translating the software designs into code in a straight-forward way. To achieve this, programming languages must strive to provide the following:

- clean syntax
- conciseness
- coherence in its core concepts
- ability to express high-level concepts
- ability to use the appropriate programming paradigm

The items cover from the low-level machinery of writing the code, to the high-level mapping of design to code. Programming languages typically focus on the first bullets on this list, but they completely ignore the last ones. And those are the most important.

Making it easier to map design into code is one of the core goals for Sparrow. Making it possible to create (zero-cost) high-level abstractions and allowing the programmer to use the most appropriate programming paradigm are vital for this problem.

Most programming languages insist that `if` and `for` are the main tools of a programmer. They are wrong. A lot of problems can be far easier if we just use a different programming paradigm. Sometimes using a declarative approach is the easiest way to solve the problem; think for example of expressing a parser in an EBNF-like notation. Other times what makes more sense is using a query language directly.

1.2.4 Flexibility

Flexibility is the ability of a language to be well applied to a large variety of problems.

A flexible language shall **NOT** do the following:

- assume one programming paradigm is a silver bullet
- insist that there is a *canonical way* of solving all problems
- assume that the creator of the language knows the right tools that programmers will need in the future

Instead, a flexible programming language shall:

- be simple
- focus more on library features and less on core language features
- enable extending the language with new abstractions
- enable adding new programming paradigms to the language
- provide means of growing the programming language by its community.

Sparrow strongly believes in these principles. Sparrow aims at implementing new features and programming paradigms for the language as library extensions. This way, a small core language can be extended easily to move with the trends and best-practices in programming.

In this sense, Sparrow can be a core language that nests a lot of domain specific languages within it.

1.2.5 Metaprogramming

Sparrow relies on compile-time metaprogramming to improve on efficiency, naturalness and flexibility. For this vision to come true, we need a compile-time metaprogramming system that is flexible and easy to use.

We call it *hyper-metaprogramming*, and it has the following traits:

- metaprogramming is Turing complete
- metaprograms can be written with the same abstractions and constructs as traditional programs
- the syntax of metaprograms is identical to the syntax of regular programs
- semantics shall be the same
- a programmer can write data structures and algorithms that work both at compile-time and run-time without duplicating code

Sparrow has all these by default. Actually, all the code is compile-time ready, meaning that the user can invoke it during compilation. This provides an easy way of executing complex algorithms during the compilation.

This is the approach envisioned by Sparrow to be extremely flexible.

We also use metaprogramming to bridge the gap between naturalness and efficiency. We can allow the programmer to write high-level code that can be translated by the use of metaprogramming into low-level efficient code. The more information we have at compile-time, the more efficient the resulting code will be.

CHAPTER 2

Quickstart

2.1 Compiling the compiler

2.1.1 External dependencies:

- CMake (3.9.0 or later)
- Boost (1.54 or later)
- LLVM 5-7

2.1.2 Building on Mac or Unix-like platforms

Ensure you have all the dependencies installed. The typical way is to install them from a package manager (brew, apt, etc.).

Create a "build" directory under the Sparrow trunk folder and go there ($sprDir/build). Then run the following commands

```
cmake ..
cmake --build .
```

Depending on your system, CMake may not find a proper version of LLVM, and it will fail the first step; you'll see this appear in the CMake output. It may be required for you to pass the path to the CMake files of the LLVM installation. Here is an example of passing that directory:

```
cmake .. -DLLVM_DIR=/usr/local/opt/llvm/lib/cmake/llvm
```

After these steps, optionally, you can also install it:

```
cmake --build . -- install
```

Tested on:

- Ubuntu Trusty Tahr (14)
- Arch Linux
- MacOS (OS X 10.14)

The Ubuntul and MacOS will be actively maintained. Arch Linux compilation may break from time to time, as Arch is continuously updating packages.

Compiling on CentOS doesn't work out of the box, because all packages are too old. A lot of manual work is required for this to work.

Note on the compiler. Sparrow is actively tested with Clang compiler; however GCC should work too. Please note that during the compilation of the Sparrow compiler *llc* executable is needed; this is part of LLVM.

2.1.3 Compiling Sparrow compiler on Windows 64-bit

Not tried since a long while. Expect errors.

2.1.4 Running the tests

Go into "test" folder of Sparrow distribution and run:

```
./test-all.py
```

the tests should run successfully (at least the vast majority of them)

2.1.5 Using docker

Get the `sparrowlang/sparrow` docker image. This contains a pre-compiled Sparrow installation inside it. Try running:

```
SparrowCompiler -help
```

Then, to run tests you can run:

```
./test-all.py
```

CHAPTER 3

Tutorials

3.1 PhD thesis documentation [slightly outdated]

- See the PhD presentation (accompanied by this video)
- Read the PhD thesis on Sparrow

These present a slightly outdated version of Sparrow, but nevertheless it provides a good tour of Sparrow's most important features, with multiple examples and measurements. It also contains the design rationale for Sparrow.

3.2 Beginner's introduction to Sparrow

This page provides a small introduction into Sparrow programming language. We expose the basic features that allow users to write traditional programs. The aim is to introduce the basic concepts that users will encounter while programming in Sparrow, without diving into more complex features.

By no means, this presents all the important concepts that a Sparrow programmer should know to be proficient, but it covers the core ones. It is more a tour of the low-level primitives that Sparrow exposes.

3.2.1 Hello, world!

In following computer science tradition, the first example program that is given in a language is the *Hello, world!* program. In Sparrow, a program that prints the text `Hello, world!` to the console can be written as:

```
fun sprMain
    cout << "Hello, world!" << endl
```

The `sprMain` function will be called when the program starts.

Similarly to a C++ program, to print something to the console Sparrow uses the insertion operator to put the string into the `cout` object. This object represents the standard console output. Adding an `endl` expression at the end causes the program to print a new-line character.

Please note that Sparrow is an indent-based programming language. Although the user can write {, } and ;, those can be inferred from the layout of the code.

3.2.2 Values and expressions

The cout construct can be used to display to the console various values, as shown below:

```
cout << 1 << endl
cout << -1 << endl
cout << 3.141592 << endl
cout << "square root of 2 is " << 1.41421356237 << endl
```

The values can also be used in arithmetic expressions:

```
cout << (2+2) << endl
cout << (12-4) << endl
cout << (2*3.141592 / 180.0) << endl
cout << "square root of 2 is " << Math.sqrt(2) << endl
```

Here the parentheses are optional, but are added for better readability.

The standard arithmetic operations (+, -, * and /) can be used for numbers (integers, floating point). The standard operators can be overloaded, and moreover, non-standard operators can be defined by the user.

The Math.sqrt(2) is a call to a function defined in the Sparrow standard library that computes the square root of its argument. In Sparrow, functions can be called by following the conventions of most programming languages: *funName(args)*.

A line such as cout << (2+2) << endl is an expression statement. 2+2 is a subexpression, cout and endl are operands, and << is an operator. This is an expression that, instead of producing an useful value, will print something to the console.

3.2.3 Variables

An important concept of imperative programming languages is the variable. A variable is an *alias* to a memory location in which we store values. Here are some examples of defining variables in Sparrow:

```
var n = 10
var m: Int = 15
var f: Float = 0.3
var greet = "Hello, world!"
var k: Int64
var p1, p2, p3: String
```

When declaring a variable one needs to supply the name of the new variable (or variables), an optional type, and an optional initial value. It is not possible to have a variable definition that lacks both the type and the initial value. If the variable receives an initial value without a type, the type of the value will be taken from the initializer. If the variable does not have an initial value, the variable will be *default constructed* (more precisely, the default constructor will be called for the variable); this assures that the variable is initialized.

After definition, the variables can be used in any place in which a value can be used:

```
cout << "Our greeting: " << greet << endl
cout << m-n << endl
cout << f*n << endl
```

Moreover, the value of a variable can be changed at any time by calling the assignment operator:

```
m = 20
k = m-n
p1 = "Our greeting: "
p2 = "Hello"
p3 = ", world!"
```

Like C++, Sparrow provides a series of operators on integer values. For example:

```
k = ++m     // m will become 21, and k will become 21 too
k = m++     // m will become 22, and k will get the old value: 21
k -= n      // subtract n from k
n /= 2      // divide n by two
```

3.2.4 Basic types

In Sparrow, any value, variable, or expression needs to have a well defined type. A type determines the way a value can be encoded in the system's memory and what operations are valid for that variable.

The standard library defines `Int` as the main type to be used for storing signed integers; it has 32 bits. For better control, it also defines `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`; as their name implies, these cover both signed and unsigned, of sizez 8, 16, 32 and 64 bit. `Int` and `Int32` are aliases.

To represent floating point numbers, the language defines the type `Float`; this is 64 bit. For better control, the language also defines `Float32` and `Float64`. In this case too, `Float` is an alias to `Float64`.

To represent booleans the language defines the `Bool` type. To represent characters we have the `Char` type. In Sparrow, strings use UTF-8 encoding, so setting the `Char` to 8 bits is an obvious choice.

In the most basic form, strings can be represented as a `StringRef` type. This just refers to the string, but does not hold ownership of the string data. To use a string with ownership of data, one can use the `String` type. String literals have the type `StringRef`, but there is an implicit conversion between a `StringRef` and `String`.

Sparrow does not allow implicit conversion (called *type coercions*) between numeric types. However, explicit conversions can be made between any numeric types. With the assumption that most of the times `Int` will be used for integers and `Float` for floating-points, having implicit conversions has more downsides than positives.

3.2.5 References

Sparrow supports references as a method of referring to a memory location. Although Sparrow references resemble C++ references more closely, one can think of them as being pointers.

A reference can be declared in Sparrow using the `@` operator applied to a type, as shown in the following example:

```
var i: Int = 1
var ri: @Int = i        // We need to initialize the reference

cout << ri << endl      // prints 1
i = 22                  // also changes ri
cout << ri << endl      // prints 22
ri = 33                 // also changes i
cout << i << endl       // prints 33
```

As can be seen in this example, having a reference (`ri`) to a particular value associated with a regular variable (`i`), any change to the original variable will be reflected in the reference variable, and vice-versa. Otherwise, the reference variable can be used just like a regular variable.

3.2.6 Control structures

Like most imperative programming languages, Sparrow supports control structures like `if`, `while`, and `for`. The structure for `if` statements is identical to the one in C++:

```
if n % 2 == 0
    cout << n << " is even" << endl
else
    cout << n << " is odd" << endl
```

The `while` statement is a combination of C++'s `while` and `for` statements. In the most basic form, the `while` statement executes a command (or a list of commands) as long as a condition is true. For example, the following code computes the length of the Collatz sequence that starts with a given number `n`. A Collatz sequence is a sequence of numbers that, starting with a given number we continuously apply a special transformation to the given number to produce more values, until we reach value 1. The used transformation is the following: if the number is even, divide it by two; otherwise multiply it by three and add 1. For example, the Collatz sequence that starts with the number 13 is: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. This sequence has the length 10. Here is the code:

```
var len = 1
while n > 1
    ++len
    if n % 2 == 0
        n /= 2
    else
        n = n*3 + 1
```

In Sparrow, we allow adding a *step* action to a `while` statement. For example, summing the squares of all natural numbers up to a certain value can be written like this:

```
var res = 0
while n > 0 ; --n
    res += n*n
```

Note that this form is somewhat similar to the `for` instruction from C++. The main difference is that the initialization statement needs to be placed before the `while` statement, and not inside it.

The `for` structure from Sparrow is similar to range-based for loops from C++ and other languages. Instead of providing an initialization statement, a condition expression, and a step statement, the user needs to provide a range that can produce values. Here is one example:

```
var numbers: Int Vector = getNumbers()
for val = numbers.all
    cout << val << endl
```

The `numbers` object is a vector of integers. Like all the standard containers it exposes an associated function named `all`, which returns a range that we can use to iterate over the values in the vector. In our example, `val` is a variable introduced with the `for` structure, and will have the type `Int`.

To iterate over a set of numbers (e.g., from 1 to `n`), one can use the following code:

```
for x = 1..n
    cout << x << endl
```

Here, `..` is an operator that generates a range that will yield values between 1 and `n` (open range). To indicate a closed range, one needs to use three dots (`...`) instead of two. One can also iterate with a given step:

```
for x = 1...n ../ 2 // odd numbers in range [1, n]
    cout << x << endl
```

Here, `..`, `...`, and `../` are all infix operators that act on numbers. We will see that ranges represent an important concept in Sparrow, and there are many other range constructors.

Sparrow does not support the `goto` statement.

3.2.7 Function definitions

Functions are the main method of abstracting computations. The following example presents a function definition in Sparrow that can compute the `n`'th Fibonacci number:

```
fun fib(n: Int): Int
    if n <= 1
        return 1
    else
        return fib(n-1) + fib(n-2)  // recursive; not optimal
```

If the function does not return a value, it can return the `Void` type, or it can omit the return type completely:

```
fun greet1(name: String):Void
    cout << "Hello, " << name << endl
fun greet2(name: String)
    cout << "Hello, " << name << endl
```

If the function does not take any arguments, the arguments list can be omitted:

```
fun geetTheWorld
    cout << "Hello, world!" << endl
```

Sometimes, when a function is simple enough the user can define the function with an alternative syntax that puts emphasis on the returned value rather than the actual instructions involved. Here is one example:

```
fun sum(x, y: Int) = x+y
```

This has exactly the same semantics as writing the function in a slightly more complicated way:

```
fun sum(x, y: Int): Int
    return x+y
```

The functions defined so far can be used as follows:

```
greet1("Alice");     // prints "Hello, Alice"
greet2("Bob");       // prints "Hello, Bob"
greetTheWorld();     // prints "Hello, world!"
greetTheWorld;       // parenthesis can be omitted here
cout << sum(2, 4) << endl;  // prints 6
```

The reader should note that if a function does not take any parameters the parentheses can be omitted when calling the function. This provides an interesting property of the language in that we can use function and variable names interchangeably, without changing the code that uses the variable/function.

So far, we have shown function definitions that operate on concrete data types. In addition to those, the Sparrow programming language allows definitions of *generic* functions that have parameters of *concept* types. For example, the previously defined `sum` function can work on all numeric types, not just on values of type `Int`. The following function definition is able to work on all numeric types:

```
fun sum(x, y: Numeric) = x+y;
```

The `Numeric` name refers to a *concept* defined in the standard library that accepts any numeric type (e.g., `Int`, `UInt64`, `Float`).

There is a special concept in Sparrow called `AnyType` that is compatible with any type. Here is an example of a function that prints to the console the value given as parameter:

```
fun writeLn(x: AnyType)
    cout << x << endl

writeLn(10);                    // prints an Int value
writeLn(3.14);                  // prints a Float value
writeLn("Pretty cool, huh?");   // prints a StringRef value
```

Both the `sum` function above and this `writeLn` function are generics, template functions, just like C++ template functions. This means, that the compiler will actually generate three `writeLn` functions for the three instantiations shown here: one with a `Int` parameter, one with a `Float` parameter, and one with a `StringRef` parameter. All these three functions will be compiled independently of each other.

In cases where all parameters are `AnyType`, the parentheses and the type specifications can be omitted:

```
fun writeLn x
    cout << x << endl
fun sum x, y = x+y;
```

As can be seen from the definition of the `sum` function, this form can be very compact.

A function is not just a definition that can be invoked directly; we can store a function in an variable. This allows us to separate the binding of the function name from the actual function call. Here is one example, in which we pass a function as a parameter to another function:

```
fun applyFun(n: Int, f: AnyType)
    for x = 0..n
        cout << f(x) << ' ' << endl;
fun mul2(x: Int) = 2*x;
fun sqr(x: Int) = x*x;

var f = \mul2;       // type: FunctionPtr(Int, Int)
applyFun(10, f);     // 0 2 4 6 8 10 12 14 16 18
f = \sqr;
applyFun(10, f);     // 0 1 4 9 16 25 36 49 64 81
```

At line 8 we create a variable and initialize it with a reference to the `mul2` function. To take the reference of a function, Sparrow uses the backslash operator. The type of this function will be `FunctionPtr(Int, Int)` (a function that returns an `Int` and takes one `Int` as parameter). This variable can then be passed as the second argument to the `applyFun` function. Note that instead of using `AnyType` for the second parameter we could have used `FunctionPtr(Int, Int)`.

We could have passed the function references directly to the function call, but we wanted to show that we can store function references in variables, just like any other values.

There is an easier method of achieving the same result, without defining the `mul2` and `pow` function prior to the call to `applyFun`. Instead, we could have used lambda functions (or anonymous functions):

```
applyFun(10, (fun x = 2*x));    // 0 2 4 6 8 10 12 14 16 18
applyFun(10, (fun x = x*x));    // 0 1 4 9 16 25 36 49 64 81
```

The form `(fun ...)` does the following: creates a function-like structure that can be called with the written computation, and then instantiates this functor to produce an object that can be called under the specified conditions. Note that this object is of an unspecified type, and cannot be placed inside a `FunctionPtr(Int, Int)`.

This expression can become a *closure* if it refers to variables declared in the scope in which it is used. In Sparrow, one needs to explicitly declare all the variables that are used by the closure. For example, if we generate the first lambda function to parameterize the factor we are multiplying with, we can write:

```
var k = 2;
applyFun(10, (fun.{k} x = k*x));     // 0 2 4 6 8 10 12 14 16 18
k = 3;
applyFun(10, (fun.{k} x = k*x));     // 0 3 6 9 12 15 18 21 24 27
```

3.2.8 Operators

Like in most programming languages, there are three types of operators in Sparrow, depending on the placement of the operator relative to its argument(s): prefix, infix and postfix. Prefix and postfix operators are unary, whereas infix operators are binary. An operator can be either a set of symbols or a regular function name. Moreover, Sparrow does not limit the names of operators formed by symbols to a fixed set (like C++ for example).

Here are some basic examples of operators in Sparrow:

```
-10                // '-' is a prefix operator
--k;               // prefix operator
k++;               // postfix operator
a + b * c;         // '+' and '*' are infix operators
a + -b * c;        // '+' and '*' are infix operators, '-' is prefix
```

Defining an operator is very similar to defining a function. Here is an example of defining an operator to raise a number to an integer power:

```
fun **(x: Float, p: Int): Float
    var res = 1.0
    for i = 0..p
        res *= x
    return res

cout << (3 ** 2) << endl        // 9.0
cout << (3.2 ** 2) << endl      // 10.24
```

Defining unary operators is as easy as defining infix operators; we just need to specify whether we need prefix or postfix operators:

```
fun pre_**(x: @Int): @Int
    x = x*x
    return x
fun post_**(x: @Int): Int
    var old = x
    x = x*x
    return old

var a, b = 5
cout << **a << endl     // writes 25, a becomes 25
cout << (b**) << endl   // writes 5, b becomes 25
```

If the `pre_` and `post_` prefixes are missing, then the operators can be used as both prefix and postfix operators.

Note that for prefix operators the parentheses are not needed, whereas for the postfix operators they are. In both cases, the first occurrence of << needs to be an infix operator; after it we can have a *primary expression* or a prefixed primary expression. In the first case, it is clear to the compiler that we have a prefix operator (because `**` cannot be an operand), while in the second case the compiler will treat `b` as the second operand of `<<`, and `**` as the next infix operator.

In general, in an expression that is separated by spaces, the terms on the even positions are infix operators and the terms on the odd positions are operands. This rule changes if an operand has one or more prefix operators applied to it, in which case we collapse the prefix operators first. If the expression has an even number of terms, the last term is a postfix call. Here is an example:

```
a + - - b * c !!!
```

In this expression, `- - b` will be treated as one operand, `+` and `*` as infix operators, while `!!!` will be treated as a postfix operator.

Beside operators that are formed by symbols, Sparrow allows operators to have alphanumeric names. For example, the `pow` and `sqr` functions previously defined can be used in the following way:

```
2 pow 3             // 8
2 sqr               // 4
`sqr` 3             // 9
2 pow 3 sqr         // 64 = (2^3)^2
```

Note that, in order to distinguish prefix name operators from name operands, Sparrow requires the placement of these prefix operator names in backquotes.

This is an important feature of Sparrow that allows writing concise programs, without losing performance.

3.2.9 Ranges

In Sparrow, a range is a collection (not necessarily finite) of elements that can be iterated through in an well-defined order. From an implementation point of view, ranges need to support three operations: *is the range empty?*, *get the current value*, and *move to the next value*. With these three operations one can extract all the values from the range.

We have already seen that `1..n` is a range. This is a range which will produce `Int` values starting from `1` and ending with `n` (without actually yielding `n`).

All the standard containers provide associated functions for accessing their values through ranges. In addition, Sparrow provides methods of generating ranges. Here are some of them:

```
repeat(13)                // infinite range with value 13
repeat(13, 5)             // 13 repeated 5 times
generate( (fun = 13) )    // infinite range with value 13
generate(\getNextRand)    // infinite range with values of getNextRand
generate1(2, (fun x = x*x))    // 2, 4, 16, 256, ...
```

In addition to those, there are a lot of functions that apply transformations to existing ranges. The most common is the `map` operation. It applies a functor to the given range to produce a new range:

```
1..10 map (fun x=x*x)   // 1, 4, 9, 16, 25, 36, 49, 64, 81
1..10 map (fun x=x/2)   // 0, 1, 1, 2, 2, 3, 3, 4, 4
1..5 map (fun x=repeat(2*x, x)) // range of ranges:
                                // (2), (4,4), (6,6,6), (8,8,8,8),
                                // (10,10,10,10,10)
```

Another important range operation is `filter`. It skips elements in the input range if they do not satisfy a predicate:

```
1..10 filter (fun x = x%2==1)   // 1, 3, 5, 7, 9
1..10 filter (fun x = x<5)      // 1, 2, 3, 4
```

Here are some examples of other range functions:

```
(1..) take 5                    // 1, 2, 3, 4, 5
(1..) takeWhile (fun x=x<=5)    // 1, 2, 3, 4, 5
(1...3) ++ (10...12)            // 1, 2, 3, 10, 11, 12
(1...3 cycle) take 8            // 1, 2, 3, 1, 2, 3, 1, 2
```

To illustrate the power of ranges we would like to solve the following problem: *the sum of the first 10 Fibonacci numbers that are greater than a given number*. Here is the Sparrow solution using ranges:

```
var res = (1..) map \fib filter (fun.{n} x = x>n) take 10 sum
```

Our solution is simple, and yet efficient. One can easily check that this solution does what it is supposed to. Starting from the range of natural numbers, we map them to Fibonacci numbers, we take only the values that are greater than the given `n`, we get the first 10 such elements, and finally we sum them.

All the ranges, including `..`, are simple functions defined in standard library. That means, that the user can implement new types of ranges easily.

3.2.10 Data types and object-oriented programming

Sparrow does **not** support Object-Oriented-Programming (OOP). We believe that the benefits that OOP provides does not justify the complexity added to the language. Moreover, using OOP tends to produce suboptimal designs. So what to use instead?

Most of the benefits of OOP can be achieved using packages and simply grouping data and code together (a notable exception to this is subtype polymorphisms). Let us take an example:

```
datatype MyItem
    id: Int
    name: String
    description: String
    _borrower: String

fun ctor(this: @MyItem, id: Int, name, description: String)
    this.id ctor id
    this.name ctor name
    this.description ctor description
fun dtor(this: @MyItem)
    cout << "Destroying item " << id
fun borrowTo(this: @MyItem, borrower: String)
    _borrower = borrower
fun restore(this: @MyItem)
    _borrower = ""
fun isAvailable(this: @MyItem) = _borrower.empty
fun borrowerName(this: @MyItem) = _borrower
```

Similar to a C `struct`, Sparrow uses `datatype` declarations to define data structure. This allows the user to add more data types to be used along the primitive types. Unlike the OOP languages (C++, Java, C#, etc.), Sparrow does not allow functions to be placed inside datatypes. But, just as well they can be placed after the data type. Please note that, our functions have a parameter named `this` that allows the body of the functions to use fields from the datatype directly.

The usage of this datatype and its associated functions is straightforward:

```
var item = MyItem(1, "pen", "a nice, blue color pen")
item.borrowTo("Alice")
```

(continues on next page)

(continued from previous page)

```
if !item.isAvaiable
    cout << " Item" << item.name
    cout << " is lent to " << item.borrowerName << endl
```

We can define a variable of the new type, and then we can access the fields of the datatype, using the `.` syntax; in our case, we accessed the `name` field. What is somehow surprising is that we can access functions defined near the datatype the same way we access fields. It looks extremely similar to other OOP languages.

The way the functions are defined, we can also use the traditional function call notation:

```
borrowTo(item, "Alice")
```

Furthermore, we can write this code using operator notation (see above):

```
item borrowTo "Alice"
if !(item isAvaiable)
    cout << " Item" << (item name)
    cout << " is lent to " << (item borrowerName) << endl
```

Accessing data from the data structure is done using the `.` syntax; in this example, we accessed the `name` part.

What is worth mentioning about datatypes is the two special associated functions: `ctor` and `dtor`. A `ctor` (constructor) is a function responsible for creating a valid instance of the given type. The `this` object passed to this function is uninitialized, and the function promises to create a valid, initialized object. Conversely, the `dtor` (destructor) associated function is responsible for all the actions needed for cleaning up the object before the object is completely destroyed. A destructor main purpose is to release any resources (e.g., memory) that the object may hold.

In our example we defined a constructor that creates an object with the given id, name, and description. A default constructor is one that takes no parameters, and initializes the object with some default state. By default, if no default constructor is provided, the language will generate one. The same applies for a copy constructor (a constructor that can create an object by copying the data from another object of the same type). Like in the case of constructors, if the user doesn't supply a destructor, the language will automatically create one, by calling the destructors for all the data members.

Sparrow follows the C++ tradition and expects manual memory management; it does not provide garbage collection. In such a language, constructors and destructors pay a very important role.

Just like functions, datatypes can be generics as well. While a regular datatype does not take any parameters, any datatype that has parameters is a generic. Here is an example of a datatype generic:

```
[initCtor]
datatype Pair(t1, t2: Type)
    first: t1
    second: t2
```

In this example we defined a datatype parameterized by two types. We used the given parameters for the types of our two fields: `first` and `second`.

All the parameters to a datatype need to be compile-time. For example, a datatype cannot have an `Int` parameters, but can have an `Int ct` parameter. To be able to use this generic, one needs to *instantiate* it; this is the process that transforms a datatype generic into a proper datatype. Datatype instantiation is just like function application:

```
var p1: Pair(Int, Float32)             // call default constructor
var p2 = Pair(Int, Float)(1, 3.14)  // call initialization constructor
p1.first = 10
p1.second = 2.34
cout << "(" << p2.first << ", " << p2.second << ")" << endl
```

On the first line we are telling the compiler that `t1` is `Int` and `t2` is `Float`, and we ask it to instantiate a `Pair` with these two types. This is not a constructor call, it's a generic instantiation.

On the second line, we ask the compiler to generate another type, one that is parameterized with valued `Int` and `Float32`. But this, time, after specifying the parameter values for the generic, we are specifying arguments for a constructor call (`1` and `3.14`).

In our case, we haven't manually created a constructor associated with our generic datatype. But, we specified the `[initCtor]` modifier. This will tell the compiler to generate a constructor withe the right number of parameters to initialize all the fields.

3.2.11 Standard library

Sparrow provides a minimal standard library that provides the user with the basic abstractions for writing programs. The Sparrow standard library was influenced to some degree by the C++ standard library.

One of the most important abstractions in a standard library are the containers. Sparrow provides the following general-purpose containers: * `Vector` - a dynamic size array, holds the elements contiguously in memory * `List` - a generic double-linked list that allows constant time insertion and removal in any place of the container * `Set` - associative containers that ensures that objects are uniquely stored in the set; the search, insertion, and removal operations have average constant-time complexity; implemented using hash tables * `Map` - provides a mapping from a set of keys to a set of values; the search, insertion, and removal operations have average constant-time complexity; implemented using hash tables

The containers are implemented as generic datatypes. Any container has an associated function called `all` that returns a range of the elements in the container. They all provide functions for accessing elements, inserting, and removing elements from the container. Example:

```
var v: Vector(Int) = 0..100  // vector of integers
for x = v.all                // iterate over all elements
    cout << x << endl
v(0) = 12                    // change the first element
v.pushBack(42)               // append at the end of the vector
v.subrange(5, 10) sort       // sort 10 el. starting at index 5
v.insertBefore(0..10, v.all) // insert 10 numbers at start
v.remove(v.subrange(3, 12))  // remove 12 elements
```

Following C++ STL principles the algorithms that operate on data are separated from the containers holding the data. Unlike C++ which uses iterators as a bridge between containers and algorithms, Sparrow uses ranges. Here is a short example:

```
var v: Int Vector = 0..100     // use postfix operator notation
var l: Int List = 0..100
replace(v.all, 10, 110)
replace(l.all, 10, 110)
(v.all find 50) size           // range starting with 50 -> size
(l.all find 50) size
v.all map \fib sum
l.all filter (fun x = x%10 < 5) maxElement
v.all copy l.all               // copy list elements into vector elements
```

Beside containers, ranges, and algorithms, the Sparrow standard library also provides utilities, such as: strings, pointers (raw, scoped, shared), memory allocation, pairs, optionals, bitsets, math functions, etc.

This section is still under development.

CHAPTER 4

Language features

4.1 Modules

Modules represent Sparrow source code files to be compiled; to some degree, they resemble packages. Besides representing the content of a source file, a module has a name. Modules can be grouped together in hierarchies, just like packages do.

To indicate the module name, the source code would start with a module declaration:

```
module myModule
// actual source code content
```

If we were to image the source code without modules, this would be equivalent with:

```
package myModule {
    // actual source code content
}
```

The name of the module might be qualified, indicating the hierarchy in which the module lies:

```
module company.sysA.subSysA1.myModule2
```

This would be equivalent with:

```
package company { package sysA { package subSysA1 { package myModule2 {
    // actual source code content
} } } }
```

If a source code does not contain module declaration, the name of the module is computed from the name of the file, removing the extension and any path.

In practice we often construct programs with more than one modules, i.e., split the code into several files. Therefore, the declarations of the program are split across different modules. One can make the declarations of one module available to other modules by using import declarations:

```
package myOtherModule
import myModule
import company.sysA.subSysA1.myModule2
```

4.1.1 Syntax

```
SourceFile      = [Module] {TopLevel} ;
Module          = 'module' QualifiedId ';' ;
TopLevel        = ImportList (* | ...*) ;
ImportLine      = [AccessSpec] 'import' ImportName {',' ImportName} ';' ;
ImportName      = [[Id] '=' ] (QualifiedId | String) ['(' {Id} ')'] ;
QualifiedId     = Id {'.' Id} ;
```

4.1.2 Modules semantics

A module is always created for each source-code. The name of the module can be specified with the `module` syntax, at the beginning of the source file.

The module name is used only to organize the code inside the module; it has no significance for any other module that may import that module. Having multiple modules with the same module name can lead to name clashing for the symbols defined in those modules; these clashes can be linker errors.

The module name can be completely independent of the filename or its position on disk.

The code generation for a module is independent of other modules that the program might have.

4.1.3 Import semantics

As part of the body of a module, a Sparrow source code might contain a series of import lines, each containing one or multiple import names. Having all import names in only one import line, or having them spread our across multiple lines doesn't make any difference. If a module is imported several times into the same module, in the same package, the compilation behaves as if the module was imported only once.

For most practical uses, the order of the import lines don't matter, if they are placed in the same package. When the imported modules have associated compile-time functionality that should run when the module is imported, then, of course the order of import lines matter.

An import name, must correspond to an existing module. The purpose of the import name is to tell the compiler how to find the file containing the module to be imported. If the compiler cannot find the corresponding file, an error is issued.

Depending on the import name specification, the compiler will perform different types of file/directory search:

- If the import name is a string, a file with that exact name is searched for; the file extension is assumed to be present in the string; relative or absolute paths might also be present.
- If the import name is a qualified id with only one name, then the compiler assumes this is the name of the file without extension, and, adding the default extension, will search for the file.
- If the import name is a qualified id with multiple names, then the compiler will assume that the first names will correspond to a directories path, while the last name corresponds to the file name (again without extension)

After the import line, the actual import name is not used anymore.

If we have three modules like the following:

```
module he;
fun hello = "Hello, "
```

```
module wo
fun world = "world!"
```

```
module greet
import he, wo
fun greet
    cout << hello << world << endl
```

The effect of compiling the *greet* module would be similar to:

```
package toplevel_anon_1
    package he
        fun hello  // not implemented in greet module
package toplevel_anon_2
    package wo
        fun world  // not implemented in greet module
package toplevel_anon_3
    package greet
        using toplevel_anon_1.he.*
        using toplevel_anon_2.wo.*
        fun greet
            cout << hello << world << endl
```

As shown in the above example, the compiler may create anonymous top-level packages to make sure that independent modules are not affecting one another. Also, the compiler will not redefine in the *greet* modules the functions that are already defined in the *he* and *wo* modules.

4.1.4 Named imports

By default, all the imported declarations from one module can be referred directly in the module that does the import. However, importing declarations from multiple modules can lead to name clashes. We can solve this by specifying a name at the import line, name that would be later be used for accessing the declarations.

Here is an example of doing a named import:

```
module dbTest
import db = storage.sql.database
...
db.open(...)
```

In the above example, the import line corresponds the following expansion:

```
package toplevel_anon_1
    package storage
        package sql
            package database
                ...
package toplevel_anon_2
    package dbTest
        using db = toplevel_anon_1.storage.db.database
        ...
        db.open(...)
```

The same effect can be achieved by placing the import line into a package:

```
module dbTest;
package db
    import storage.sql.database
...
db.open(...)
```

4.1.5 Private and public imports

By default, if module *A* imports module *B*, which in turn imports module *C*, then *A* will not see the declarations from *C*. We say that the *imports are private* to that module.

There are however cases in which we want to make all the exports of a module to be public to the modules that import that module, i.e., making the *import public*. A good example is when we create a module as an interface for a component that spreads across different modules.

To make an import line public, one must add the *public* access specifier:

```
module MyComponent
[public] import Foo
[public] import Bar
```

4.1.6 Selective imports

It is often considered a best practice to import only the needed declarations from a module, and not all the declarations in that module. One can follow this practice and specify the names of declarations to be imported, as suggested by the following example:

```
module foo
fun f = 1
fun g = 2
fun h = 3
```

```
module bar
import foo(f, g)
...
f()     // ok
g()     // ok
h()     // ERROR
```

The import line from the above example would be translated similar to:

```
package toplevel_anon_1
    package foo
        fun f = 1
        fun g = 2
        fun h = 3
package toplevel_anon_2
    package bar
        using f = toplevel_anon_1.foo.f
        using g = toplevel_anon_1.foo.g
        ...
        f()     // ok - resolves to toplevel_anon_1.foo.f
```

(continues on next page)

(continued from previous page)

```
        g()     // ok - resolves to toplevel_anon_1.foo.g
        h()     // ERROR
```

Note: The compiler might compile all the declaration from *foo*, to make sure that everything functions properly; think of auxiliary types and using declarations. This syntax just makes sure that *bar* cannot access the non-specified declarations directly by name. Some compilers might also use this information to reduce processing.

4.2 Declarations

UNDER CONSTRUCTION

4.2.1 Syntax

```
Declaration      = UsingDecl
                 | PackageDecl
                 | DatatypeDecl
                 | ConceptDecl
                 | VarDecl
                 | FunDecl
                 ;

UsingDecl        = 'using' [Mods] QualifiedIdStar ';'
                 | 'using' [Mods] Id '=' Expr ';'
                 ;
PackageDecl      = 'package' [Mods] Id [Params] '{' {Stmt} '}' ;
DatatypeDecl     = 'class' [Mods] Id [Params] IfClause '{' Stmts '}'
                 | 'datatype'  [Mods] Id [Params] IfClause '{' {DatatypeItem} '}'
                 | 'datatype'  [Mods] Id [Params] '=' Expr IfClause ';'
                 ;
ConceptDecl      = 'concept' [Mods] Id '(' Id [Type] ')' IfClause ';' ;
VarDecl          = 'var' [Mods] IdList [Type] ['=' Expr] ';'
                 | 'var' [Mods] IdList '=' Expr ';'
                 ;
FunDecl          = 'fun' [Mods] FunName [Params] [Type] IfClause FunBody
                 | 'fun' [Mods] FunName [Params] [Type] '=' Expr IfClause ';'
                 ;


Mods             = '[' Expr {',' Expr} ']' ;
IdList           = Id {',' Id} ;
Type             = ':' ExprNE ;
DatatypeItem     = UsingDecl | FieldsLine ;
FieldsLine       = IdList Type ['=' Expr] ';' ;
FunName          = IdOrOperator | '(' ')' ;
FunBody          = '{' {Stmt} '}' | ';' ;
Params           = '(' [Formal {',' Formal}] ')'
                 | IdList
                 ;
Formal           = IdList Type ['=' Expr] ;
IfClause         = ['if' Expr] ;
```

(continues on next page)

(continued from previous page)

```
Operator         = Oper | '=' ;
OperatorNE       = Oper ;
IdOrOperator     = Id | Operator ;
IdOrOperatorNE   = Id | OperatorNE ;
```

Please note that tokens like `{`, `}` or `;` can be introduced automatically by the lexer, depending on the layout of the program.

4.2.2 Using declarations

Using declarations introduce declaration names in a new context (package).

There are three forms of the using declaration:

- `using QualifiedId` This will just add a new symbol in the current context, referring to the declaration indicated by `QualifiedId`
- `using QualifiedId.*` This assumes that QualifiedId is a package or module name, and will add all the declarations from that package to the current context
- `using NewName = QualifiedId` This creates an alias of the declaration indicated by `QualifiedId` and adds it with a new name to the current context.

Example:

```
package Foo
    package Bar
        fun f = 1
        fun g = 2
fun test1
    using Foo.Bar.f
    cout << f() << endl
fun test2
    using Foo.Bar.*
    cout << f() << endl
    cout << g() << endl
fun test3
    using ff = Foo.Bar.f
    using FB = Foo.Bar
    cout << ff() << endl
    cout << FB.g() << endl
```

4.2.3 Package declarations

Package declarations are used to group code.

Example:

```
package Grp
    datatype MyType = Int

    fun f(this: @MyType) = this.data
    fun print(this: @MyType) {...}
```

(continues on next page)

(continued from previous page)

```
fun caller
    var x: Grp.MyType
    cout << Grp.f(x) << endl
    Grp.print(x)
    x print             // OK, we are searching near the class first
    print(x)            // ERROR, cannot find 'print'
```

Packages can be generics. Please see *Generics* for more details.

4.2.4 Datatype declarations

As its name suggests, a datatype declaration introduces a new data type. In Sparrow, a datatype can only contain fields (variables) or other using declarations.

There are two main forms of declaring datatypes: an explicit one, and a simple one. The explicit form is exemplified by the following:

```
datatype MyType
    x: Int
    y: Float
    name: String
    using BaseType = Int
```

In this example, we introduced a new composite type `MyType` that contains three fields: `x`, `y` and `name`. It also contains an using name `BaseType` that expands to `Int`; there is no memory reserved for any using declarations; they are used for type introspection, especially in the context of generics.

The simple form of declaring datatypes is illustrated by the following example:

```
datatype Type1 = Int
datatype Type2 = Int*Float // Pair of Int & Float
```

This is a shortcut for the following code:

```
datatype Type1
    data: Int
datatype Type2
    data: Int*Float
```

In addition to this, the compiler will also generate a constructor that can covert the type given after = to the new type.

For any datatype declared, the compiler will also attempt to auto-generate several constructors, a destructor, a = and a == operator. For more details see *Generated associated functions*.

Datatypes can be generics. Please see *Generics* for more details.

4.2.5 Concept declarations

Formally, a concept is a predicate on types, or from a different point of view, a set of types. We use it in generic programming to be able to operate on set of types.

Example:

```
concept Swappable(x) if isValid(x.swap(x))
```

The above line can be read as: a type is *Swappable* if for a value `x` of that type, the expression after `if` (the *if-clause*) is fulfilled – that is, `x.swap(x)` is a semantically valid construct.

As an if-clause, there can be any compile-time expression that evaluates to `Bool`. If, for a type, the given if-clause will result in errors, the type will not model the concept.

Such a concept can be then use in generic programming, in the following way:

```
fun doSwap(x: @Swappable, y: typeOf(x))
    x swap y
```

Please see *Generics* for more details on generics.

TODO: base concepts

4.2.6 Variable declarations

TODO

4.2.7 Function declarations

TODO

4.2.8 Access modes

Declarations in Sparrow can have three access modes: `public`, `protected` and `private`. By default, the `public` mode is assumed. If the declaration starts with _, then `private` is assumed instead. If the declaration is a ctor, dtor or one of the =, == operators, by default the access mode is `protected`.

The user can force an access mode by using the `public`, `protected` and `private` modifiers. This is especially useful for public imports, where we cannot control the name of the symbol imported.

A public declaration means that everybody can access and use that declaration.

A private declarations means that only code from the **same module** can use that declarations. Code that imports the module cannot use the declaration.

Example:

```
module A
fun f = 1
fun _g = f() + 1
fun h = g() + 1
```

```
module B
...
A.f()    // ok, 1
A.g()    // ERROR
A.h()    // ok, 3
```

Note: Private declarations of one package can be seen from another package in the same module.

This rule simplifies *friendship* relations between entities that are closely related. See the following example:

```
module goodFamily
package mom
    fun publicAttitude {...}
    fun _secrets {...}
package dad
    fun publicAttitude {...}
    fun _secrets {...}
package child
    fun schoolBehaviour {...}
    fun _secrets {...}

fun discussion
    // No secrets between the members of the module
    mom._secrets
    dad._secrets
    child._secrets
```

A protected declaration is accessible by everybody. The difference from a public declaration is that protected declarations are not considered in `using` clauses without explicit names given. This way, one can hide the declaration from general name lookups. This hiding occurs across different modules as well as within the same module. Protected declarations can be still be accessed whenever we access them indirectly by searching near a class at the operator call.

Example:

```
package ShySpace
    datatype Foo = Int
    [protected] fun print(this: Foo) { cout << this.data }

using ShySpace.*
var x: Foo = 0
x print     // ok print searched indirectly through Foo
x.print     // also ok
print(x)    // ERROR: print is not visible here
```

4.3 Generated associated functions

UNDER CONSTRUCTION

For each type, the compiler will attempt generate the following associated functions:

- initialization constructor (if required)
- default constructor
- copy constructor
- constructor to convert from compile-time to run-time
- destructor
- the = assignment operator
- the == equality check operator

TODO

4.4 Generics

UNDER CONSTRUCTION

There are three types of generics in Sparrow:

- package generics
- datatype generics
- function generics

TODO

4.5 Basic expressions

- lambda expressions
- parenthesis expression
- identifiers
- literals: null, true, false, integers, floating point, char, string
- function applications
- compound expressions

TODO

4.5.1 Syntax

TODO

4.6 Operators

There are 3 types of operators in Sparrow, based on their form:

- prefix operators: `oper_name arg`; example: `++a`
- postfix operators: `arg oper_name`; example: `a++`
- infix operators: `arg1 oper_name arg2`; example: `a + b`

All operators in Sparrow are user-defined. However, compilers may choose to implement some of them directly, without relying on user-code, for optimization purposes.

Sparrow does not have a fixed set of operators. Almost all combination of symbols are valid operator names. Moreover, alphanumeric identifiers can also be operator names.

4.6.1 Syntax

```
Operator          = Oper | '=' ;
OperatorNE        = Oper ;
IdOrOperator      = Id | Operator ;
IdOrOperatorNE    = Id | OperatorNE ;

Letter            = 'a'-'z' | 'A'-'Z' | '_' ;
Digit             = '0'-'9' ;
OpChar            = '~' | '``' | '@' | '#' | '$' | '%' | '^' | '&'
                  | '-'| '+' | '=' | ' | ' | '\' | ':' | '<' | '>'
                  | '?' | '/'| '*' | '.' ;
Oper              = (OpChar, {OpChar}) - '=' - '.' - ':' ;
Id                = Letter, {Letter | Digit}, ['_', Oper] ;

Expr              = PrefixExpr, {IdOrOperator, [PrefixExpr]} ;
ExprNE            = PrefixExprNE, {IdOrOperatorNE, [PrefixExprNE]} ;
PrefixExpr        = {'`', Id, '`' | Oper} SimpleExpr ;
PrefixExprNE      = {'`', Id, '`' | Oper} SimpleExprNE ;
SimpleExpr        = "..." ;
SimpleExprNE      = "..." ;
```

According to the above grammar, `.` and `:` are never operator names, but can be part of operator names. In addition, = can be part of operator names and, but sometimes can be by itself an operator name (i.e., `a = 10`), and sometimes it is not. The cases in which = is not an operator name, are the cases in which = cannot be part of an expression; this is the case with type expressions; i.e., `a: Int = 3` – here = is a separator between a type expression and an initializer expression.

The following are valid examples of operators: `+`, `-`, `++`, `**`, `#$`, `=/=/=/=`. Valid operator names can be also: `foo`, `bar123`, `_123`, `oper_$#@`.

As can be seen from grammar, one can use identifiers as prefix operator names if they are enclosed into backticks.

Here are some more examples of operators, this time in context:

```
a + b         // add a and b
a + b * c     // add a to the product of b and c
++a++         // prefix ++ on a, then apply the postfix ++
v1 dot v2     // dot product between v1 and v2
`length` a    // calling length on variable a with prefix call notation
1..100        // the numeric range between 1 and 100, exclusive
@Int          // type representing a reference to Int
```

A nice property of the grammar, is the chaining of name operators. Here is an example of this technique:

```
(1..) map \collatzSeq map \rangeSize takeWhile (fun s = s < 500) rootMeanSquare
```

In this example, `..` and `rootMeanSquare` are postfix operators. On the other hand, `map`, `takeWhile` and < are infix operators. The above line has the following meaning: take the range of integers starting from 1 (pseudo-infinite range), map it with the `collatzSeq` function, then map the result to `rangeSize` function, then apply `takeWhile` operator with a lambda function, and finally apply `rootMeanSquare` as a postfix operator.

If we were not to have prefix operators, then reading an expression of the form `a b c d e f` is straight forward:

- all the elements on odd positions are operands (`a`, `c` and `e`)
- all the elements on even positions are operator names (`b`, `d` and `f`)
- if the expression has an odd number of elements, the last operator is a postfix operator; all the rest are infix operators

Prefix operators can be identified by observing pure operator names (symbols only) on odd positions – symbols only identifiers cannot ever be operands. Whenever a prefix operator is encountered, the odd/even rule above is shifted with one position. For example, `a b ++ c` is interpreted as `a b (++ c)`, and cannot be interpreted as `(a b ++) c`, with ++ being an operand.

Defining operators

Defining an operator is identical to defining a regular function. Example:

```
datatype Complex {re, im: Float}
fun + (x, y: Complex) = Complex(x.re+y.re, x.im+y.im)
fun - (x: Complex) = Complex(-x.re, -x.im)
```

In this example, the + operator is a binary operator (can be used in infix operations) and − is an unary operator; this is inferred from the number of parameters. In this example, the − function can be used both for infix and prefix operator calls. To distinguish between prefix and postfix operations, user can add the `pre_` or `post_` prefix to the function name. Examples of prefix-only and postfix-only functions:

```
fun pre_++ (x: Complex): @Complex { x+=1; return x; }
fun post_++(x: Complex): Complex { var old=x; x+=1; return old; }
```

Please note that Sparrow does not impose a limit on the number of parameters to the functions with operator names. If such a function has too many parameters, it won't be able to be called with operator notation.

4.6.2 Operator lookup

A prefix operator `oper_name arg` can be reduced to an infix operator with a null operand: `null oper_name arg`. Similarly, a postfix operator `arg oper_name` can be reduced to an infix operator with a null operand: `arg oper_name null`.

We explain in this section the semantics of a simple infix operator call `arg1 oper_name arg2`, possibly with a null argument.

To simplify the explanation of the operator lookup, it is convenient to separate out the processes involved in the operator lookup:

1. overall operator resolving
2. operator selection
3. overload procedure

Overall operator resolving may invoke operator selection once or multiple times; similarly, operator selection can invoke overload procedure once or multiple times.

Overall operator resolving

This process will handle any operators that the compiler wants to handle directly, and it will provide the fallback cases for when the raw operator selection didn't succeed. The process is summarized by the following table:

Table 1: overal operator resolving

Condition	Action
op= `===`, infix	Handle reference equality directly
op= `!==`, infix	Handle reference inequality directly
op= `:=`, infix	Handle reference assignment directly
op= `\`, prefix	Handle function pointer operator directly
op= `construct`, prefix	Handle construct calls directly
not handled yet	**apply operator selection**
not handled yet, `a != b`	attempt to transform into `!(a == b)`
not handled yet, `a > b`	attempt to transform into `b < a`
not handled yet, `a <= b`	attempt to transform into `!(b < a)`
not handled yet, `a >= b`	attempt to transform into `!(a < b)`
not handled yet, `a <op>= b`	attempt to transform into `a = a <op> b`
not handled yet, infix+postfix	attempt to transform first arg into `(a .)`

The first entries in this table force the compiler to deal directly with the operators, rather than executing user-defined code; this is for speeding up certain common operations. Then, if these operations are not applied, the compiler will attempt to use the operator selection process to resolve the operator call; the vast majority of operator calls should be handled here. If this does not succeed, the compiler will attempt to handle some fall-back cases. It tries to infer inequality based on equality, it tries to infer other relational operators based on the definition of <, and it tries to resolve oper-equals operators. The last rule is the application of the dot operator, in case nothing worked; this is only applied to the first argument of the operator.

If no action is successful, i.e., cannot find a proper operator or sequence of operators to call, an error is reported.

Operator selection

Given two arguments (`<arg1>` and `<arg2>`), at least one non-null, and an operation name (`<oper>`), this process will attempt to find a way to call the operation with the given arguments.

We call the *base argument* the first non-null argument. This process will search around the datatype of the base arguments for matching functions.

If the operation is prefix, it will also consider searching with `pre_<oper>`. If the operation is postfix, this will also consider searching with `post_<oper>`. We call these search names `<operWithPrefix>` – note, this may not be valid.

There are multiple contexts in which the selection process can search, and the compiler can search both with `<oper>` and `<operWithPrefix>`. The compiler will attempt to perform the search, in order, according to the following list:

1. in the datatype of the base argument, using `<operWithPrefix>` (if valid)
2. in the datatype of the base argument, using `<oper>`
3. in the package that contains the datatype of the base argument, using `<operWithPrefix>` (if valid)
4. in the package that contains the datatype of the base argument, using `<oper>`
5. upward from the context of the operator call, using `<operWithPrefix>` (if valid)
6. upward from the context of the operator call, using `<oper>`

At each step, the compiler will perform a name search in the appropriate context for the given name. If declarations are found, the overload procedure is invoked, trying to select the appropriate declaration that can be called. If at one step a match is found, the compiler will not continue with the rest of the steps. If valid names are found, but no match is found, the compiler will continue with the next step. Please note that it may be ok for this process to fail; a follow-up step in the overall operator resolving process may succeed.

Example:

```
package A
    datatype Foo
        a, b, c: Int

    fun f1(this: @Foo) {}
    fun f2(this: @Foo) {}

fun oper(this: @A.Foo)
    cout << 'this will be selected' << endl

package B
    fun g1(this: @A.Foo) {}
    fun g2(this: @A.Foo) {}

    fun test
        var x: A.Foo

        x oper  // searches 'post_oper' inside A.Foo (a,b,c)             -> FAIL
                // searches 'oper' inside A.Foo (a,b,c)                  -> FAIL
                // searches 'post_oper' inside A (f1,f2)                 -> FAIL
                // searches 'oper' inside A (f1,f2)                      -> FAIL
                // searches 'post_oper' up from current context (B, global) -> FAIL
                // searches 'oper' up from current context (B, global)      -> SUCCESS
```

Overload procedure

TODO: add more details

4.6.3 Precedence and associativity

For infix operators, we also need to consider precedence and associativity. Precedence determines the order in which different infix operators inside the same expression are called. Associativity determines whether for an expression containing only operators of the same type the order of applying the operator is from left to right, or from right to left.

For each infix operator we can associate a numeric value such that we can compare the precedence of two operators. Let us denote by p_1 the precedence of the operator `op1` and by p_2 the precedence of the operator `op2`. Then, the expression `A op1 B op2 C` would be interpreted as `(A op1 B) op2 C` if $p_1 \geq p_2$, and as `A op1 (B op2 C)` if $p_1 < p_2$. For example, multiplication and division have higher precedence than addition and subtraction.

For an infix operator `op`, an expression like `A op B op C` would be interpreted as `(A op B) op C` if `op` has left associativity, and `A op (B op C)` if `op` has right associativity. Most of the mathematical operators have left associativity, but an operation like assignment makes sense to have right associativity. Also, if one were to define an exponentiation operator, it should also have right associativity.

As Sparrow operators are defined in the library, precedence and associativity can also be defined in the library – with `using` directives:

```
using oper_precedence_default = 100
using oper_precedence_+       = 500
using oper_precedence_*       = 550
using oper_assoc_=            = -1
```

Whenever the compiler needs to know the precedence of an operator `<op>` it will search for the using with the name `oper_precedence_<op>` and use its value. If this cannot be found, it will use the `oper_precedence_default` value.

Whenever the compiler needs to know the associativity of an operator `<op>` it will search for the using with the name `oper_assoc_<op>`. If we found a value and it's negative, the compiler will use right associativity; otherwise it will use left associativity.

Example of using precedence and associativity for a new operator:

```
fun **(x, y: Float)        = Math.pow(x, y)
using oper_precedence_** = 1 + oper_precedence_* // higher precedence than␣
↪multiplication
using oper_assoc_**      = -1                    // right associativity

cout << 4 * 3 ** 2 << endl       // 36 == 4 * (3**2)
cout << 4 ** 3 ** 2 << endl      // 262144 == 4 ** (3**2)
```

4.6.4 The dot operator

The compiler uses a special `.` operator to ease the access for some datatypes. Consider the following example:

```
package A
    [initCtor]
    datatype Ptr(type: Type)
        using ValueType = type

        _ptr: @ValueType
    fun get(this: Ptr): @ValueType = _ptr
    fun .(this: Ptr) = _ptr

[initCtor]
datatype MyObj
    x: Int
fun print(this: @MyObj)
    cout << "MyObj.print: " << x << endl

var p: A.Ptr(MyObj) = ...
p.x = 10                    // OK
p.print                     // OK
p print                     // OK
```

If the dot operator would not be present, then the last 4 lines would not be valid anymore. The user would have been supposed to write:

```
p.get().x = 30              // UGLIER
p.get().print               // UGLIER
p.get print                 // UGLIER
```

Defining a dot operator allows the overall operator resolving process to use the value returned by the dor operator to resolve the above expressions.

Formally, if `X` has a dot operator (i.e., `(X .)` would be valid), then if `x` is of type `X` (or derived), then `x <op> y` would fall back to `(x .) <op> y`.

Please note that the syntax of Sparrow does not allow to write `(x .)`. In general, `.` is not an operator name. However, the syntax of Sparrow was extended for dot operator to allow users to write `fun ..`

In the above example we shown how dot operator works with operator calls, but also with compound expressions (things of the form `a.b`). The main idea is the same, but the details are somehow different. In compound expressions we don't have overload resolution, so if find a name, the selection process stops successfully; this means the dot operator will not be tried.

This can generate some strange errors. In the above example, if we remove the package (making `Ptr` be contained in the same package as `MyObj` and `print`), then the line `p.print` would result in an error. The `print` function is found near the `Ptr` class, the finding process stops, and then the result cannot be used, and thus an error is generated.

4.7 Modifiers

Modifiers allow changing the semantic of Sparrow constructs.

Example:

```
[initCtor]
datatype MyType
    a: Int
    b: Bool

var x = MyType(13, true)
```

The `initCtor` is a modifier that makes the `MyType` datatype to have an initialization constructor for the two variables.

4.7.1 Syntax

```
Mods                = '[' Expr { ',' Expr } ']' ;

Stmts               = { Stmt } ;
Stmt                = [Mods] ImportLine
                      | [Mods] UsingDecl
                      | [Mods] PackageDecl
                      | [Mods] DatatypeDecl
                      | [Mods] ConceptDecl
                      | [Mods] VarDecl
                      | [Mods] FunDecl
                      | [Mods] ExprStmt
                      | [Mods] BlockStmt
                      | [Mods] IfStmt
                      | [Mods] ForStmt
                      | [Mods] WhileStmt
                      | [Mods] BreakStmt
                      | [Mods] ContinueStmt
                      | [Mods] ReturnStmt
                      ;
```

4.7.2 Predefined modifiers

`public` modifier

Applies to all declarations

Semantics Marks the declaration as being public. A public declaration can be accessed from any module.

See *Access modes*

`protected` modifier

Applies to all declarations

Semantics Marks the declaration as being protected. A protected declaration can be accessed from any module, but it will not be considered in *using* clauses without explicit names.

See *Access modes*

`private` modifier

Applies to all declarations

Semantics Marks the declaration as being private. A private declaration can be accessed only from the current module.

See *Access modes*

`ct` modifier

Applies to all declarations, *if* statements, *while* statements, *for* statements

Semantics Marks the declaration or statement to be executed at compile-time. A *ct* declaration will not have a runtime counterpart. A *ct* statement will be executed at compile-time; used in specializing code or code generation.

Examples:

```
[ct] fun addRef(t: Type): Type // function is available only at compile-time

[ct]
if sizeOf(t) <= sizeOf(Int32)
    var storage: Int32
else
    var storage: Int64
// Depending on the size of 't', this will create different 'storage' variables

[ct]
for x = 1..5
    doStuff(ctEval(x))
// Will generate the following code:
//      doStuff(1)
//      doStuff(2)
//      doStuff(3)
//      doStuff(4)
```

`rt` modifier

Applies to all declarations

Semantics Ensures the declaration has meaning both at run-time and at compile-time. The default behavior. Can be used in `[ct]` environments to go back to default `[rt]` environments

autoCt modifier

Applies to functions

Semantics If the function is called with only with compile-time arguments, the function will be compile-time; otherwise it will be run-time.

Example:

```
[autoCt] fun printVals(x, y: Int)
    cout << x << ' ' << y << '\n'

var six = 6
var seven = 7

printVals(6, 7)          // prints '6 7' at compile-time
printVals(six, 7)        // prints '6 7' at run-time
printVals(six, seven)    // prints '6 7' at run-time
```

ctGeneric modifier

Applies to functions

Semantics Marks the function as being both a compile-time function and a generic function.

A regular function (non-`[ct]`) that has compile-time parameters will be a generic; it can change the semantics of its body based on the compile-time parameters, and has run-time presence. On the other hand, a function marked as `[ct]` is not a generic; it doesn't have any run-time presence and it cannot change the semantics of its body based on the parameters.

A `[ctGeneric]` function is a combination of both: it's a compile-time function (no run-time presence) but has a different code generated for each set of parameters.

convert modifier

Applies to `ctor` functions and datatypes

Semantics Used to indicate that a datatype can be implicitly converted from other types through constructors marked as `[convert]`. To convert a value from a type `A` to a type `B` implicitly, the following conditions must be met:

- the `B` datatype must be declared as `[convert]`
- `B` needs to have a constructor marked as `[convert]` that can construct a `B` from values of type `A`

noDefault modifier

Applies to datatypes, `ctor` and `dtor` functions

Semantics When applied to datatypes, it tells the compiler not to generate default functions for the datatype. When applied to constructors and desructors, it tells the compiler not to try to inject constructor/destructor calls for the members of the datatype.

initCtor modifier

Applies to datatypes

Semantics It instructs the compiler to generate an extra constructor to initialize all the members of the datatype.

When generating the initialized constructor, the following are applied:

- the initialization constructor will have a parameter for each field of the datatype
- the order of parameters in the initialization constructor will be the same as the oder in which the corresponding fields are declared in the datatype
- if the field of the datatype has an initializer, the corresponding parameters will have the same initializer

`bitcopiable` modifier

Applies to datatypes

Semantics It instructs the compiler that the given datatype can be safely copied bitwise (i.e., with `memcpy`). The copy constructor will be elided whenever possible.

Also, functions returning *bitcopiable* will not take the location in which the resulting value needs to be placed by (hidden) reference.

For example, a `Vector` applied to a *bitcopiable* type can avoid calling the copy constructor of the elements whenever resizing the data.

The basic numeric types are all *bitcopiable*.

`autoBitcopiable` modifier

Applies to datatypes

Semantics Instructs the compiler to detect whether the datatype can be bitcopiable or not. A datatype with `[autoBitcopiable]` modifier applied will become *bitcopiable* if all the fields are also *bitcopiable*; if at least one of the fields is not *bitcopiable* then the datatype will not be *bitcopiable*.

It is used for datatypes like `Tuple` to automatically become *bitcopiable* base on the field types.

`macro` modifier

Applies to functions

Semantics When making calls to the function, it will pass the raw AST (abstract syntax tree) nodes to the functions, without even compiling them. Used for manipulating the source code.

Example: `assert` uses macros to extract the variables out of the given condition, to be able to print them whenever the assertion fails.

`noInline` modifier

Applies to functions

Semantics Prevents the compiler from inlining the function.

`native` modifier

Applies to functions, datatypes

Form `native(` *stringLiteral* `)`

Semantics Assigns the specified name to the given function/datatype. It does not apply any name mangling rules when generating the assembly name.

Native datatypes are used to map types defined in the standard library onto machine types. I.e., `[native("i32")] datatype Int` is mapping the type `Int` to a signed 32-bit machine type.

Functions that are declared native would not change their return type (non-*bitcopiable* return types are typically transformed into hidden pointer parameters).

Typically used when interacting with other libraries.

4.7.3 User-defined modifiers

TODO: not yet implemented

This section is still under development.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search