Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

A functional programming language (suitable for education of functional programming)

Bakalárska práca

2018
Michal Štrba

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

A functional programming language (suitable for education of functional programming)

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Richard Ostertág PhD.

Bratislava, 2018
Michal Štrba

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname: Michal Štrba
Study programme: Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study: Computer Science, Informatics
Type of Thesis: Bachelor´s thesis
Language of Thesis: English
Secondary language: Slovak

Title: A functional programming language (suitable for education of functional programming)

Annotation: Design and implementation of a simple statically typed pure functional language with built-in types for Char, Int (with arbitrary-precision) and Float with support of function overloading. Implementation of multiple „external“ side effect interpreters for this pure functional language.

Supervisor: RNDr. Richard Ostertág, PhD.
Department: FMFI.KI - Department of Computer Science
Head of department: prof. RNDr. Martin Škoviera, PhD.

Assigned: 08.11.2017

Approved: 08.11.2017 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Michal Štrba
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: A functional programming language (suitable for education of functional programming)
Funkcionálny programovací jazyk (vhodný pre výučbu funkcionálneho programovania)

Anotácia: Návrh a implementácia jednoduchého staticky typovaného čistého (pure) funkcionálneho jazyka so vstavanými typmi pre Char, Int (bez obmedzenia veľkosti) a Float s podporou preťaženia funkcií. Implementácia viacerých „externých“ interpreterov vedľajších efektov pre tento čistý funkcionálny jazyk.

Vedúci: RNDr. Richard Ostertág, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 08.11.2017

Dátum schválenia: 08.11.2017 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgement

I'd like to thank my supervisor for always promptly spraying my pages with red ink.

Abstract

In this thesis, we introduce a new purely functional programming language called Funky. The motivation for its creation arose from the joy experienced when playing with the pure λ-calculus and from the frustration with other functional languages. The result is a fairly unique language. It's simple enough to be learned in a matter of hours, while its subtle design choices make it very expressive. Most importantly, Funky's syntax makes it possible to write vertical code – code that reads top to bottom, without introducing any major syntactic sugar (like Haskell's `do` notation), or violating functional purity. Funky's approach to side-effects is also one of its most distinguishing features. Instead of building side-effects into the language, Funky makes it possible to write custom side-effect interpreters, that take the Funky code as a data structure and interpret is in some way. This makes it possible to easily penetrate any programming domain simply by creating appropriate side-effect interpreters.

Keywords: programming language, functional programming, vertical code, side-effects, side-effect interpreters, function overloading, λ-calculus

Abstrakt

V tejto práci predstavujeme nový čistý (pure) funkcionálny programovací jazyk Funky. Motivácia na jeho tvorbu prišla z pôžitku prežítého pri hraní sa s čistým λ-kalkulom a tiež z frustrácie z ostatných funkcionálnych jazykov. Výsledkom je pomerne jedinečný jazyk. Je dostatočne jednoduchý, aby sa ho človek zvládol naučiť za pár hodín, pričom, na prvý pohľad nenápadné, črty jeho dizajnu ho robia veľmi expresívnym. Asi ich najdôležitejším dôsledkom je možnosť písania vertikálneho kódu - kódu, ktorý sa číta zhora dole, a to bez zavedenia akéhokoľvek väčšieho syntaktického cukru (ako napríklad `do` notácia v Haskelli), alebo porušenia funkcionálnej čistoty. Prístup k vedľajším efektom je tiež jednou z najjedinečnejších vlastností jazyka Funky. Namiesto vstavania vedľajších efektov priamo do jazyka, Funky umožňuje vytváranie vlastných interpreterov vedľajších efektov, ktoré zoberú kód Funky ako dátovú štruktúru a interpretujú ju nejakým spôsobom. Toto umožňuje ľahké vniknutie Funky do akejkoľvek sféry programovania – jediné, čo treba spraviť je naprogramovať vhodný interpreter vedľajších efektov.

Kľúčové slová: programovací jazyk, funkcionálne programovanie, verikálny kód, vedľajšie efekty, interpretery vedľajších efektov, preťaženie funkcií, λ-kalkulus

Table of contents

Introduction

In the beginning God created λ-calculus... or at least Church did.

We took note of that and started experimenting. That was the beginning of this project. We implemented many λ-calculus interpreters, played with numerous algorithms and data structures in it, and that gave us insights. The most important one is here:

Pure λ-calculus extended with the ability to create global, recursive definitions makes for a *very decent programming language*.

The key is that creating global definitions – assigning names to expressions – in combination with λ-calculus' substitution semantics, makes it possible to easily elevate abstractions to a nearly arbitrary level, all in a very natural, human-friendly way.

But, the pure λ-calculus has its downsides. No type system, no built-in numbers, no obvious way to implement side-effects, and so on. We were aware of the existing purely functional programming languages, most notably Haskell. However, we found that all of them introduce many complicated features, that ruin the experience of harmony we enjoyed in pure λ-calculus.

Hence, we set forth to build a language that stays true to those feelings. A question arised:

What's the minimum amount of features we need to add to λ-calculus to make it usable for real-world problems?

After many design iterations, a lot of trying and failing, we believe we figured it out. In this work, we present the Funky programming language – the result of our efforts.

The design process of Funky was one without a strict set of features, or goals to reach. In fact, we believe that such an approach would be detrimental to the final result. Instead, we tried to discover the language, in a way. Starting from pure λ-calculus, adding features that seemed right, one by one, exploring them as deeply as possible to find all of their uses. This led to a simple, yet powerful language, with a small number of features that find a wide range of use.

In the first chapter, the language is described. Many of its design decisions are rationalized in this chapter. The most important design decisions are, however, very subtle and hard to appreciate at the first sight. The second and the third chapter try to do them justice.

Chapter 1

The Funky Programming Language

In short, Funky is vanilla λ-calculus with a type system on top, one special `switch`/`case` control structure, and three magical built-in types: `Char`, `Int`, and `Float`.

Three aspects distinguish it most from other functional languages:

- Side-effects are handled by special programs called interpreters. An interpreter takes your program expressed as a transparent (no hidden information) data structure describing what should be done.

- Few clever syntax tweaks that enable writing code that reads top to bottom, without any use of a special `do` notation, as seen in Haskell.

- Type system, which allows function overloading (having multiple functions of the same name but different types), but disallows higher-kinded (not the same as higher-order) types.

The first two of these aspects each has a dedicated chapter discussing their impact in its full extent. In this chapter, we will describe the Funky programming language in general, its syntax, semantics, and how we can use them.

1.1 Top-level code structure

```
record Point = x : Int, y : Int

union Line = blank | Point -- Line  # -- is an infix constructor

# length^2 calculates total squared length of a line
func length^2 : Line -> Int =
    \line
    switch line
    case blank 0
    case (--)  length^2-from

func length^2-from : Point -> Line -> Int =
    \pt0 \line
    switch line
    case blank
        0
    case (--) \pt1 \rest
        let (x pt0 - x pt1)       \dx
        let (y pt0 - y pt1)       \dy
        let ((dx ^ 2) + (dy ^ 2)) \delta
        delta + length^2-from pt1 rest

func main : IO =
    let (Point 1 2 -- Point 7 8 -- Point -4 0 -- blank) \line
    println (string (length^2 line));  # prints 257
    done
```

Here's a nice, short, example of Funky code. Just looking at it, we can already discern something about the language. Comments start with #, identifiers may contain non-alphanumeric symbols, there are some backslashes that seem to introduce variables, and the top level consists of a set of definitions, either of types or functions, each of which starts with a keyword (`record`, `union`, `func`, ...).

In total, there are four keywords that signify a top-level definition, three of which were used in the example above:

- `func` – a function definition with a mandatory type and a body
- `record` – a compound data type, similar to structs from C or records from Pascal
- `union` – a type consisting of several alternative forms, similar to `data` from Haskell
- `alias` – a name taken as an alias for another type

Nothing, but these four kinds of definitions, is allowed on the top level.

The order of definitions is irrelevant. All definitions "see" each other.

In the future, two more top-level keywords will be added: `package` and `import`. They will be used for package and dependency management, which is currently dealt with in a particularly spartan way by simply listing all the needed files to the compiler.

Before we examine each of the top-level keywords individually, we'll briefly describe general syntax rules.

1.2 General syntax rules

In every programming language, the source code, represented by characters, is first split into the smallest, semantically indivisible units called tokens. Examples of tokens are:

```
func, IO, length^2, 42, "Hello, world!", '%', (, +, =, |, \
```

In Funky, tokens are generally separated by whitespace characters, except for a few special characters that are always parsed as individual tokens, even when not separated from their surroundings by whitespace. These characters are:

```
( ) [ ] { } , ; \ #
```

String and character literals are one more thing that does not follow splitting by whitespace. Although the `"` and `'` characters can be found inside other tokens, when a token starts with a quote (single or double), everything until the closing quote is taken as a single token.

For example, the following (not idiomatically formatted) sequence of characters:

```
println("Length: " ++ (string;length^2 line))
```

gets split into these tokens:

```
println
(
"Length: "
++
(
string
;
length^2
line
)
)
```

Comments start with `#` and continue until the end of line. Everything inside a comment is completely ignored by the language and has no semantics.

Funky's syntax is not layout or indentation sensitive. Every time a whitespace is needed, one space is always equivalent to any sequence of whitespace characters.

1.2.1 Kinds of tokens

Some tokens have special meaning in Funky. Others can be used as names for variables, functions and types.

Here's a list of tokens that act as keywords in Funky and can't be used as identifiers:

`( ) [ ] { } , ; \ # : | = record union alias func switch case`

However, except for the underlined ones, you can use them inside your identifiers. For example, the function for prepending an element to a list is called `::` and a logical "or" is `||`. Also, there's a function called `|>`, and so on. The underlined can't be used inside identifiers, because they are always parsed as separate tokens.

The other class of special tokens is literals. From the total of five kinds of literals, instances of four are represented by single tokens. These are: integers, floating-point

reals, characters, and strings. The fifth kinds of a literal is a list literal, but these are represented by multiple tokens, so they don't concern us here.

The parser splits tokens into their respective categories by the rules described below. In the description of the rules, *<digit>* stands for any decimal digit, `...` stands for any sequence of non-whitespace characters, `|` separates multiple alternatives.

- *<digit>*`...` `|` `+`*<digit>*`...` `|` `-`*<digit>*`...` – tokens of this form are considered numbers. If the `...` consists solely of digits, the token is taken as an integer literal. Otherwise it's taken as a floating-point literal. Floating-point literal may contain digits, a single decimal point, and an exponential notation. If the token can't be understood as a floating-point literal, a compilation error is raised.

- `'...'` – character literals. The content between quotes may be escaped using back-slash characters, just as expected. If the content consists of multiple or zero characters, a compilation error is raised.

- `"..."` – string literals. Same as character literals, except they can contain arbitrarily many characters.

All tokens that are not keywords, nor literals, can be used as identifiers.

1.3 Functions

Every righteous guide of a functional programming language has to include the factorial function, so we start with that.

```
func n! : Int -> Int =
    \n
    if (n == 0)
        1;
    n * n! (n - 1)
```

This is the classical recursive definition of the factorial function. We named the function `n!`, because we can, although, too short and cryptic names are generally discouraged in Funky. The type of the function is `Int -> Int`, which means that this is a function which takes an argument of type `Int` and returns a value of type `Int`. The `=` symbol

is followed by the body of the function, which is simply an expression in the Funky language which must conform to the type of the function.

The backslash on the first line of the body is supposed to look like a lambda: λ. This is a legacy from the ingenious λ-calculus formal system created by Alonzo Church[4]. What it means is: this expression is a function which takes an argument called `n` and returns everything that follows after.

The next line contains an `if`. In Funky, `if` is a regular function with three arguments: condition, then-value, else-value. Its type is: `Bool -> a -> a -> a`. If the condition is `true`, `if` returns its first argument (then-value), otherwise it returns its second argument (else-value). In our case, the condition is the expression `(n == 0)`, which does the obvious. The then-value argument is `1`.

Now, there's a semicolon. Semicolon is very simple – all it does is it puts everything that follows after it inside parentheses. If we put the last line in parentheses and removed the semicolon, the result would be the same. Thus, the else-value argument is the whole last line: `n * n! (n - 1)`, a recursive evaluation of the `n!` function multiplied by `n`.

The whole expression looks similar to what an imperative program in C would look like:

```
if (n == 0)
    return 1;
return n * fac(n - 1);
```

Of course, we're not trying to look like C. We're trying to write code that reads top to bottom. This is something that imperative languages do very well, but as it turns out, functional programming can do it just as fine. We'll discuss this topic in a great detail in a dedicated chapter.

1.3.1 Function body – expressions

In our describing of functions, we'll first focus on the function bodies – expressions. We'll intentionally omit the `switch`/`case` construct, and leave for the section about unions.

The expression syntax adds very little to the pure λ-calculus. All that's added is the literals, the `switch`/`case` structure, the semicolon, and type annotations.

Here's the complete expression grammar:

```
<expression> ::= <literal>                          |
                 <identifier>                       |        reference
                 <expression> <expression>          |        application
                 \ <simple-variable> <expression>   |        abstraction
                 <switch-expression>                |
                 ( <expression> )                   |
                 <expression> ; <expression>        |
                 <expression> : <type>
```

Three non-terminals from the above definition are not yet defined. These are: *<simple-variable>*, *<switch-expression>*, and *<type>*. Both *<switch-expression>* and *<type>* will be defined in later sections.

The *<simple-variable>* non-terminal is an *<expression>* that only makes use of the rules underlined in the definition, and none others. This non-terminal will be used a few more times in the definitions, so it's good to remember it.

The three rules with the names on the right (reference, application, abstraction) will be referenced by these names later in the text.

Now we'll clarify some ambiguities with the grammar, because it's quite ambiguous in itself. We'll clarify it by showing examples of expressions without parentheses, then putting parentheses in a wrong and in the right way. I believe this is easier to understand than a rigorous formal definition.

The first source of ambiguity is the application syntax used more than once. As with all examples here, the first line is without parentheses, the second line shows an expression that's semantically different from the expression without parentheses, and the third line shows an expression that's semantically same:

```
<expression> <expression> <expression>
<expression> (<expression> <expression>)                  different
(<expression> <expression>) <expression>                  same
```

Another ambiguity comes with abstractions:

```
\ <simple-variable> <expression> <expression>
(\ <simple-variable> <expression>) <expression>           different
\ <simple-variable> (<expression> <expression>)           same
```

A few come with the semicolon:

`<expression> ; <expression> <expression>`	
`(<expression> ; <expression>) <expression>`	*different*
`<expression> ; (<expression> <expression>)`	*same*

`<expression> <expression> ; <expression>`	
`<expression> (<expression> ; <expression>)`	*different*
`(<expression> <expression>) ; <expression>`	*same*

And the last one with type annotations:

`<expression> <expression> : <type>`	
`<expression> (<expression> : <type>)`	*different*
`(<expression> <expression>) : <type>`	*same*

That'll do for ambiguities.

We will not describe the expression evaluation process in Funky, because it's the same as any other purely functional language. The main evaluation mechanism is β-reduction. All we need to know is that Funky is lazy, with implicit support for tail-recursion, and call-by-name, just like Haskell.

Now we'll move on to describe some of the expression syntax peculiarities in a greater detail and answer some semantics questions, like "what does the semicolon do?".

1.3.1.1 Literals

As we've learned already, some tokens can be used as identifiers, others act as special syntax symbols, while the rest represent literals – direct values of built-in types and strings (`String` is not a built-in type). String literals are each a single token, although they can contain whitespace. There's one more kind of literals – list literals. They are composed of more tokens, though – initial `[`, list of values separated by `,` and a closing `]`.

Let's examine all kinds of literals, one by one:

- Character literals – these are enclosed in single quotes (`'`) and represent a single Unicode character. Their type is `Char`. Escaping special characters with backslash works as expected. Examples: `'a'`, `'\n'`, `'č'`, `' '`, `'\''`, etc.
- Integer literals – start with a digit or a `+`/`-` sign followed by a digit subsequently only contain digits (no decimal point or scientific notation). They represent integers of arbitrary precision. Their type is `Int`. Examples: `42`, `-1`, `+1`, `19237489124398124891324`, etc.

- Floating-point literals – start with a digit or a +/- sign followed by a digit and are distinguished from integer literals by containing a decimal point or the exponential scientific notation. They represent 64-bit precision floating-point numbers. Their type is `Float`. Examples: `4.0`, `-1.5`, `1e-9`, `7.14e5`, `3.14159265358979`, etc.

- String literals – are enclosed in double quotes (") and represent a string of Unicode characters. It's worth noting that the type `String` is just an alias for `List Char` – a list of characters. String literals are thus just a syntactic sugar for a list of characters constructed by using the `(::)` and `empty` functions. We'll learn more about those later. Here are a few examples:

```
"hello"   # ('h' :: 'e' :: 'l' :: 'l' :: 'o' :: empty)
"\n\"\n"  # ('\n' :: '"' :: '\n' :: empty)
```

- List literals – are enclosed in square brackets, with list elements separated by commas. Just as strings, list literals are just a syntactic sugar that expands to a series of `(::)` and `empty` applications. Examples:

```
[1, 2, 3, 4, 5]   # (1 :: 2 :: 3 :: 4 :: 5 :: empty)
[[a, b], [c]]  # ((a :: b :: empty) :: (c :: empty) :: empty)
```

1.3.1.2 Semicolon

Oftentimes, an argument to a function is a non-trivial expression itself and has to be put inside parentheses. But too many parentheses can harm readability and make programs less fun to write. This is where the semicolon comes handy:

```
extract (int (strip-whitespace number))
```

can be instead written as:

```
extract; int; strip-whitespace number
```

All semicolon does is it puts everything that follows after it into one big pair of parentheses. It's very similar to Haskell's `$` operator, except, the semicolon is more concise. That's very important. Semicolon is used all the time in Funky. It's one of the two crucial features that make it possible to write code that reads top to bottom. The other crucial feature is the syntax of abstractions. We'll discuss this advantage of Funky in a dedicated chapter.

1.3.1.3 Infix functions

We've seen use of `+`, `-`, `^`, and other operators in previous code examples and you might've wondered if those receive special treatment from the compiler or they're just regular functions. The truth is, they're just regular functions. The only special thing about them is that they're composed solely of special symbols (no letters or numbers) and that makes them *infix*, as opposed to prefix. Prefix functions come before their argument, infix functions come after it instead.

Infix functions have lower precedence than prefix function application and are all right-associative. Among themselves, all infix functions have the same precedence.

```
x * y + z
```

is equivalent to:

```
x * (y + z)
```

Similarly:

```
sqrt x + log y ^ n ^ 2
```

is equivalent to:

```
(sqrt x) + ((log y) ^ (n ^ 2))
```

And lastly:

```
"your number is: " ++ string; n * 2
```

is same as:

```
"your number is: " ++ (string (n * 2))
```

The reason for making all infix functions same precedence and same associativity is consistency and simplicity. We could add a possibility of specifying the precedence index and associativity of infix functions, but this adds burden on programmers, and only resulting in little benefit. Alternatively, we could hard-wire precedence and associativity for specific operators known to the compiler. This approach would however diminish seamless extensibility of the language – functions defined by the programmer would be inferior to the built-in ones in this regard. The last option is to go for maximum consistency at the cost of "looking less like math". I believe this cost is not severe enough to rule out this option, and so it has been chosen.

1.3.1.4 Variable shadowing

Introducing a local variable in an abstraction "shadows" all other uses of that name. Let's say there's a global function called `reality`. The expression

```
reality exists
```

applies that global function with an argument named `exists`. However, if this expression is located inside an abstraction which binds the name `reality`, the result is different:

```
\reality
reality exists
```

This time, the name `reality` on the second line no longer refers to a global function, instead it refers to the local variable introduced by the abstraction. In general, if there is a local variable of the referenced name, it will be used every time, all global functions of that name are automatically ruled out.

A nested abstraction can overshadow a previous binding:

```
let (just you) \reality
let nothing    \reality
explain reality
```

Introducing a second variable of the same name shadows the previous instance, which is no longer accessible. The `reality` reference on the line 3 refers to the variable introduced on the line 2.

1.3.2 Type of a function

Funky is a strongly typed language. Everything has a known type at compile-time, no implicit conversions between types occur to confuse programmers. However, Funky offers a convenient type inference mechanism, so programmers rarely need to type-annotate variables.

Explicitly stating types of global functions is required, though. This is because, in contrast to most purely functional languages, Funky supports function overloading. There may exist multiple functions of the same name with different types. Without explicit type information, selecting the right version to use based on context would be impossible, or at least very confusing.

Funky bases its type system on the standard Hindley-Milner model[1], which enables all the properties described above. The system is extended only by the support for function overloading (ad-hoc polymorphism), and recursive aliases.

Function types are specified using this grammar:

```
<type> ::= <type-variable>     |
           <type-application>  |
           <type> -> <type>    |
           ( <type> )

<type-application> ::= <type-name> | <type-application> <type>
```

Two non-terminal from the above grammar aren't defined yet: `<type-variable>` and `<type-name>`. Type variables, as we'll learn soon, stand for an arbitrary other type. They are just identifiers, that additionally must start with a lower-case letter. Type names, on the other hand, stand for defined types, possibly higher-order types. They are also just identifiers, but contrary to type variables, these must start with an upper-case letter.

We need to clear up some ambiguities, just as we did with the expression grammar. The only ambiguities are introduced by the arrow, which is used to express function types:

```
<type> -> <type> -> <type>
(<type> -> <type>) -> <type>                    different
<type> -> (<type> -> <type>)                    same

<type-application> <type> -> <type>
<type-application> (<type> -> <type>)           different
(<type-application> <type>) -> <type>           same

<type> -> <type-application> <type>
(<type> -> <type-application>) <type>           error
<type> -> (<type-application> <type>)           same
```

Now we'll walk through the Funky's type system, starting from the easiest – the built-in types – and finishing with the hardest, but still very easy – the higher-order types.

1.3.2.1 Built-in types

Funky has three built-in types: `Char`, `Int` and `Float`. `Char` represents all Unicode characters. `Int` represents arbitrary-precision integers. `Float` represents 64-bit precision floating-point decimal numbers. Literals and built-in functions are used to create values of these types.

Here are a few examples of functions that have a built-in type:

```
func newline : Char = '\n'
func seconds-in-a-day : Int = 86400
func pi : Float = 3.14159265358979

func one-plus-two : Int = 1 + 2
func ln-of-pi : Float = ln pi
```

As these functions don't take any arguments, they simply represent a single value of a built-in type. Functions like these are called constants.

Note, that `Float` literals must contain a decimal point. Number `42` is an `Int` literal, not a `Float` literal. To make it a `Float` literal, we need to add the point: `42.0`.

1.3.2.2 Function types

A function that takes an argument of type `A` and returns a value of type `B` has type `A -> B`. For example:

```
func double : Int -> Int = \x 2 * x

func round : Float -> Int =
    \x
    int (x + 0.5)
```

If a function takes multiple arguments, we simply make a function taking the first argument that returns a function taking the second argument, like this:

```
# gcd returns the greatest common divisor of numbers x and y
func gcd : Int -> (Int -> Int) =
    \x \y
    if (x < y)
        (gcd y x);
    if (y == 0)
        x;
    gcd y (x % y)  # % is modulo
```

And, since the -> operator is right-associative (as all infix operators are), we can omit the parentheses:

```
# gcd returns the greatest common divisor of numbers x and y
func gcd : Int -> Int -> Int =
    \x \y
    if (x < y)
        (gcd y x);
    if (y == 0)
        x;
    gcd y (x % y)  # % is modulo
```

Function can take functions as arguments:

```
# differentiate returns approximate derivative of a function f in x
func differentiate : (Float -> Float) -> Float -> Float =
    \f \x
    let 1e-8 \eps
    ((f (x + eps) - f (x - eps)) / (2.0 * eps)

func linear-2x : Float -> Float = differentiate (^ 2.0)
```

The last line may be difficult to understand for those unfamiliar with functional programming. Remember, a function accepting two arguments is actually a function accepting one argument and returning another function that accepts the second argument. Exploiting this property by supplying some, but not all, arguments to a function is called partial application. In the last line, we use it twice. For better understanding:

```
differentiate (^ 2.0)
```

is equivalent to:

```
\x differentiate (\y y ^ 2.0) x
```

1.3.2.3 Type variables

Red roses, red sofa, red shoes, red car. The word "red" is not owned by roses, nor sofa, nor shoes, nor car. "Red" is used with many words, and will be used with words that don't yet exist. "Red" is a naturally general concept. Not as general as, say, "good"

or "bad", but quite general.

Many words in programming – functions – are similarly general by nature. We, humans, are easily capable of conceiving such words and if we were prevented from defining them, we'd feel trapped. The language could not grow[7] as well.

Type variables are a way to define such general functions in Funky. As a rule, all concrete types, such as `Char`, `Int`, and so on, must start with an upper-case letter. On the other hand, identifiers starting with a lower-case letter stand for type variables (all identifiers in types must start with a letter).

Let's look at the simplest example:

```
func id : a -> a = \x x
```

Here's the definition of the `id` function from the standard library. The name `id` stands for "identity". It takes one argument and returns it back. Looking at its type, we can see the lower-case `a` there. That is a type variable.

A type variable means that a function works for any type substituted for that variable. Therefore, the `id` function actually has all kinds of types: `Int -> Int`, `Float -> Float`, `(Char -> Int) -> (Char -> Int)`, and so on. In fact, it has an infinite number of types. And, whenever we define a new type, it works with that one, too.

Note, though, that all occurences of the type variable must be replaced by the same type. The `id` function does not have the type `Int -> Char`, or anything similar.

Type variables introduce a lot of expressive power. Let's examine a few more general functions from the standard library.

```
func let : a -> (a -> b) -> b = \x \f f x
```

Here's the `let` function. It takes a value and a function accepting that value and simply returns what the function returns.

```
func if : Bool -> a -> a -> a =
    \bool \then \else
    switch bool
    case true  then
    case false else
```

The body of `if` contains the `switch`/`case` structure, which we haven't discussed yet, but notice the type. It accepts a `Bool` and two values of any, but the same, type and returns one of them.

Here's an interesting function from the standard library:

```
func flip : (a -> b -> c) -> b -> a -> c =
    \f \x \y
    f y x
```

What `flip` does is it takes a function of (at least) two arguments and swaps them – the second argument becomes the first and the first becomes the second. For example:

```
(flip (-)) 3 5  # results in 2, because 5 - 3 = 2
```

One of the most useful general functions is `.` – function composition.

```
# (f . g) x = f (g x)
func (.) : (b -> c) -> (a -> b) -> a -> c =
    \f \g \x
    f (g x)
```

It's especially useful when applied partially:

```
map ((* 2) . (+ 1)) [1, 2, 3]  # => [4, 6, 8]
```

Function composition operator has a wide-spread application in functional programming.

1.3.2.4 Higher-order types

Just as some functions happily operate on many types, some types are more general, too. A good example of such a type is a list. We can have a list of integers, a list of characters (a string, in fact), a list of apples, or a shopping list. We'll learn how to create such types later, here's how we use them.

Types that can be parametrized by other types are called higher-order types. Each high-order type has a specific number of type arguments, partial application doesn't work here. For example, the `List` type from the standard library takes one type argument, while the `Dict` type (a key-value storage) takes two – one for keys, one for values.

Here's how it looks like in code:

```
func some-numbers : List Int = [1, 2, 3, 4, 5, 6, 7]
```

The function `some-numbers` is a constant of type `List Int`, which is a list of integers.

Here's a list constant for any type, the empty list from the standard library:

```
func empty : List a = []
```

The function `empty` is not defined this way in the standard library. We'll see how it's defined in the section about unions.

```
func sum : List Int -> Int =
    \numbers
    if (empty? numbers)
        0;
    first numbers + sum (rest numbers)
```

Here's a recursive function `sum` that takes a list of integers and adds them up. It uses several list functions from the standard library, here are their types (we'll omit the bodies this time):

```
func empty? : List a -> Bool    # checks if a list is empty
func first  : List a -> a       # the first element in a list
func rest   : List a -> List a  # the list without its first element
```

Type variables cannot have arguments, so something like `m a -> (a -> m b) -> m b` is not a valid type (this is how Funky prevents generalization of monads...). Of course, it's possible to substitute a higher-order type (with all of its arguments) for a type variable, so `id [1, 2, 3]` works just fine.

1.3.2.5 Type inference

Functional languages tend to lie in a sweet spot by being statically typed and thus safe, yet avoiding verbosity by providing type inference – automatically deducing types of variables and expressions.

Type inference is a non-trivial algorithmic task, especially when function overloading is supported, as it is in Funky. It's fairly easy to understand it on an intuitive level, though.

In this short explanation, we will use the term "expression *`E`* *can* have a type *`A`*", meaning that *`A`* is one of the possible types of *`E`*.

If a variable expression `x` refers to a global function and there is a global definition of *`x`* with type *`A`*, then *`x`* can have type *`A`*. If `x` refers to a variable bound by an abstraction, then `x` can have type of that bound variable.

An application *`f x`* can have type *`B`* if *`f`* can have type *`A -> B`* and *`x`* can have type *`A`*.

An abstraction `\x` *`expr`* can have type *`A -> B`* if *`x`* can have type *`A`* and *`expr`* can have type *`B`*.

The `switch`/`case` structure will be discussed in the section about unions.

This description of type inference is not complete, but is sufficient for reasoning.

Sometimes, as will be discussed shortly in the section on ambiguity, type inference algorithm can't uniquely infer types of all expressions. In such cases, the programmer must manually supply type information in form of type annotations.

Any expression in Funky can be enriched with a type annotation using the `:` symbol:

```
<expression> : <type>
```

Everything to the left side of `:` is an expression and everything to the right of `:` is the intended type. Parentheses are often required when type annotating expressions:

```
g . (f : Int -> Float)
map (\(x : Int) x * 2) [1, 2, 3, 4]
```

With infix functions, the situation is more peculiar.

```
< : a -> a -> Bool
```

The `:` symbol gets parsed as the right argument to the `<` function, which is not what we meant, and we get a compilation error. With infix functions, we need to put the name inside parentheses:

```
(<) : a -> a -> Bool
```

The type inference algorithm is implemented in the compiler. The source that helped the most during the implementation was the paper "Algorithm W Step by Step"[2].

1.3.3 Function definition

Now, after describing function bodies and types in a great detail, it's time to put that all together in actual, fully working, function definitions.

Here's the grammar:

```
<function> ::= func <simple-variable> = <expression>
```

Where's the type? Well, the simple variable right after the `func` must have a type annotation.

Why choose this definition over a simple `func <identifier> : <type> = <expression>`? For consistency regarding infix functions. When we have an abstraction that binds an infix variable and we want to type annotate it, we have to put the variable name inside parentheses, like this: `\((<) : a -> a -> Bool)`, otherwise, the `:` symbol would be parsed as the right argument to the `<` function and we'd get a compilation error.

The same now holds when defining infix functions:

```
func < : Int -> Int -> Bool = ...    # compilation error
func (<) : Int -> Int -> Bool = ...  # OK
```

Functions can only be defined at the top level. Functions can't be defined inside other functions, except by a use of the `let` function and often with a help of the `fix` combinator (fix-point combinator function for inline recursion, more on it later).

1.3.4 Overloading

For some reason, there are almost no functional languages with full support for function overloading, otherwise known as ad-hoc polymorphism. With all its benefits, this is a mysterious phenomenon. Haskell supports restricted form of overloading, through the mechanism of type classes[9]. The Idris programming language supports function overloading with the restriction that different overloaded versions must be defined in different namespaces[3].

Funky supports nearly unrestricted function overloading. The word nearly is important here: functions with colliding types are forbidden. The precise meaning of the word *colliding* will be explained shortly.

To overload a function, we simply define multiple functions with the same name:

```
func zero : Int   = 0
func zero : Float = 0.0
```

Now, when we use the name `zero` in another function, compiler will automatically select the right version to use, based on context:

```
5 + zero    # the Int version is used
5.0 + zero  # the Float version is used
```

Here's a more complex example:

```
# map applies a function to all elements of a list
func map : (a -> b) -> List a -> List b =
    \f
    fold< ((::) . f) []

# map applies a function to the potential content of a maybe
func map : (a -> b) -> Maybe a -> Maybe b =
    \f \maybe
    switch maybe
    case nothing nothing
    case just \x just (f x)

# useless converts a list of maybe floats to a list of maybe ints
# ^ that's a useless comment
func useless : List (Maybe Float) -> List (Maybe Int) =
    map (map int)
```

In the `useless` function, the first `map` occurence refers to the first version of `map` that operates on lists. The second occurence, however, refers to the version operating on maybe's. Compiler, with the help of type inference, easily figures out the right version to use.

1.3.4.1 Colliding types

Consider these two functions:

```
func repeat : a -> List a     = \x x :: repeat x
func repeat : Int -> List Int = \x x :: repeat (x + 1)
```

A defective individual who created the second `repeat` function clearly has no idea what the word "repeat" means. But that's not our concern. Here's the concern:

```
repeat 5
```

Which version of the function `repeat` should be selected? Both fit the context. Someone might argue that the second version is clearly the right one, because it's more specific. Alright, let's see how this intuition scales:

```
func repeat' : a -> List a = repeat
```

We've just created a "copy" of the `repeat` function called `repeat'`. Which version should be selected in the body? The second version doesn't fit this time, so the answer is clear: the first one.

Now, we can do this:

```
repeat' 5
```

And we get the first version of the `repeat` function do the job, instead of the second, as someone might expect.

Confusion doesn't end here, though. Let's consider these two functions:

```
func weirdo : a -> List Int = \x [1, 2, 3]
func weirdo : Int -> List a = \x []
```

And say we have a `sum` function that adds up a list of integers. Which version of the `weirdo` function should be selected here?

```
sum (weirdo 5)
```

Neither version can be said to be more specific, because both have to specialize their type variable to fit the context.

What we've seen in the examples were functions with colliding types. Two types are *colliding* if and only if we can substitute their type variables in such a way that they end up being precisely the same type. In other words, types are colliding if there exists a context of arguments/result type without any type variables (all arguments and result type have concrete types), such that both types fit the context.

Funky avoids all of the confusion of colliding types by forbidding them. If we attempt to define two functions of the same name with colliding types, we get a compilation error.

1.3.4.2 Ambiguity

Even though forbidding colliding types prevents vast majority of ambiguities, we're still not entirely spared of them. Here's the (only kind of) way to still be ambiguous:

```
func f : Int -> Float  = float
func f : Int -> String = string

func g : Float -> String  = string
func g : String -> String = id

func h : Int -> String = g . f
```

No types are colliding here. Yet, there are two ways of selecting `f` and `g` for the body of `h`. One is here: `f : Int -> Float`, `g : Float -> String`. And the other one is here: `f : Int -> String`, `g : String -> String`. The place of ambiguity is hidden in the tunnel of composition.

When this happens, we get a compilation error complaing about the ambiguity. To fix the error, we must add some type annotations:

```
func h : Int -> String = g . (f : Int -> Float)
```

Ambiguities like this are very rare, though.

1.4 Records

Now that we've mastered defining and using functions, it's time to start creating our own types. The simplest kind of type is a record – a compound data type. A record is a single value that contains multiple values inside of it called fields. The number, names, and types of fields are known beforehand, they are specified in the record type definition.

It's fascinating that such a simple concept as a record is so poorly implemented in Haskell. Anyone familiar with Haskell knows exactly what it's about[6]. Function name collisions force the users of Haskell to put records into separate modules, while the poor accessing and updating mechanisms motivated the creation of a whole, huge, Lens library[5].

Funky avoids the first problem by supporting function overloading and avoids the second one by offering composable functions for accessing and updating record fields.

Here's the grammar of record definitions:

```
<record> ::= record <type-name> <variables> = <fields>
<variables> ::= <nothing> | <variables> <type-variable>
<fields> ::= <nothing> | <simple-variable> | <fields> , <fields>
```

The simple variables standing for record fields must have type annotations. The type name, again, must start with an upper-case letter, while all the type variables must start with a lower-case one.

Here's an example that shows all the things present in the general form:

```
record Pair a b = first : a, second : b
```

We can split the definition into multiple lines, trailing comma is allowed:

```
record Pair a b =
    first  : a,
    second : b,
```

So, what happens when we define a record and how do we use one? Let's consider a more associable record:

```
record Person =
    name : String,
    age  : Int,
```

Whenever we define a record, Funky generates a set of functions for using the record. In case of `Person`, Funky generates these five functions (bodies omited, they are internal to the compiler/runtime):

```
func Person : String -> Int -> Person  # constructor

func name : Person -> String                          # getter
func name : (String -> String) -> Person -> Person  # updater

func age : Person -> Int                       # getter
func age : (Int -> Int) -> Person -> Person  # updater
```

Funky generated a constructor, which simply takes a value for each field in order and returns the record, and also generated a pair of overloaded functions per field – a getter and an updater. A getter is very simple, it just takes the record and returns the value of a field. Updater is more nuanced, though. It always takes this general form: a field `f` of type `A` in a record `R` has an updater `f : (A -> A) -> R -> R`. The main reason for this form is composability, which will be discussed shortly.

Here's what the updater does: it takes a function, takes a record, gets the current value of the field, applies the function to that value, and returns a copy of the record where the value of the field is replaced by the result of the function. In short, it maps the field through the function. This is very useful – usually we want to update the value of a field according to its previous value, not just set it to some fixed constant.

Here's an example usage (the `IO` here is not important, but real nonetheless):

```
func main : IO =
    let (Person "Michal" 22) \me      # constructs the record
    println (name me);                # Michal
    println (string (age me));        # 22
    let (age (+ 1) me) \me            # cake day!
    println (string (age me));        # 23
    let (name (++ " Štrba") me) \me   # appends " Štrba" to the name
    println (name me);                # Michal Štrba
```

```
    let (name (const "Mišo") me) \me  # sets the name to "Mišo"
    println (name me);                # Mišo
    done
```

The `const` function from the standard library on the third line from the bottom makes a function that takes one argument and always returns `"Mišo"`. Its definition is simply: `\x \y x`. It's the idiomatic way to set a field to a value independent from the previous value.

For completeness, it's possible to define a record with no fields:

```
record Unit =
```

In such a case, we can drop the `=` sign:

```
record Unit
```

This record has a single possible value, obtained through the constructor with no arguments: `Unit`. This is not particularly useful, but it's possible.

There's nothing more to records. However, there's a little more to know about how we can use them – especially, how we can compose accessors (getters and updaters) to get or update nested records and data structures.

1.4.1 Composing accessors

Notice how the right side (return type) of the type of an updater looks similar to its left side (argument type) and precisely matches the left side of another suitable updater. Similarly, the right side (return type) of one getter matches the left side (argument type) of a suitable getter. This property of getters and updaters can be exploided for composition. Let's define two records:

```
record Point = x : Int, y : Int

record Segment =
    start : Point,
    end   : Point,
```

The fields of the `Segment` record are records themselves. Say we have a variable called `seg`, which contains the value `Segment (Point 1 2) (Point 3 4)`. We could access the X coordinate of the starting point like this:

```
x (start seg)
```

And this is sufficient for many cases. However, imagine a scenario where instead of a single segment we deal with a list of segments, let's call this list `segs`. Say we want to map this list to a list of the X coordinates of the starting points of the segments. The above method is no longer so convenient. Instead, we can compose the getters:

```
map (x . start) segs
```

How about updaters? Can we compose them as well? Say we want to increase the Y coordinate of the end point of a segment by 4. We could do this:

```
end (\pt y (+ 4) pt) seg
```

Which can be shortened to:

```
end (y (+ 4)) seg
```

And that can be rearranged by composition:

```
(end . y) (+ 4) seg
```

Notice that updaters compose in the opposite order compared to getters.

The composition can, again, be used to update all segments in a list:

```
map ((end . y) (+ 4)) segs
```

Things go deeper, though. Notice, that the type of the `map` function resembles an updater: `(a -> b) -> List a -> List b`. In fact, it works like an updater on a list. Let's make one more record, which contains a list of segments:

```
record Plan = segments : List Segment
```

Using composition, we can easily update all segments in a plan and get the updated plan:

```
(segments . map . start . y) (* 2) plan
```

I think this is pretty remarkable.

1.5 Unions

A boolean may be true or false, a list may be empty or not, a Peano integer may be zero or a successor of another Peano integer, a tree may be empty or a node. Some things are naturally represented by several alternative forms. The whole type is then the union of all these forms.

In Funky, these types are defined using the `union` keyword. Here's the grammar:

```
<union> ::= union <type-name> <variables> = <alternatives>
<variables> ::= <nothing> | <variables> <type-variable>
<alternatives> ::= <nothing>     |
                   <alternative> |
                   <alternatives> "|" <alternatives>
<alternative> ::= <prefix-identifier> <arguments>    |
                  ( <infix-identifier> ) <arguments> |
                  <type> <infix-identifier> <type>
<arguments> ::= <nothing> | <arguments> <type>
```

The `|` symbol in the definition of *<alternatives>* is inside quotes to signify that it's an actual symbol in the form and doesn't mark another alternative in the grammar.

Let's get into real examples.

```
union Bool = true | false
```

This is precisely how the `Bool` is actually defined in the standard library. Note, that in Funky it's idiomatic to start the names of the alternatives with a lower-case letter instead of upper-case, as is idiomatic in Haskell. This is because the union alternatives are often used just as regular functions and this way they blend in much better.

```
union Peano = zero | succ Peano
```

Here's a definition of Peano integers, which are just natural numbers represented in a unary notation. A Peano integer may be either zero, or a successor of another Peano integer.

Whenever we define a union, Funky automatically generates a set of functions for constructing the alternative forms. In case of `Peano`, these particular functions are generated (bodies are omited and internal to the compiler/runtime):

```
func zero : Peano
func succ : Peano -> Peano
```

Generally, Funky generates a constructor for each alternative – this function takes all the arguments specified by the alternative and returns the union in question.

Now we can construct Peano integers:

```
zero                                          # 0
succ zero                                     # 1
succ (succ zero)                              # 2
succ (succ (succ (succ (succ zero))))  # 6
```

Let's take another example, the classical Maybe:

```
union Maybe a = nothing | just a
```

These functions get generated:

```
func nothing : Maybe a
func just    : a -> Maybe a
```

Another classical example:

```
union List a = empty | elem a (List a)
```

Actually, that's not how it's defined in the standard library. The `elem` form is called `(::)` there.

```
union List a = empty | (::) a (List a)
```

Or we can use the nice infix form:

```
union List a = empty | a :: List a
```

These functions get generated by Funky:

```
func empty : List a
func (::)  : a -> List a -> List a
```

And for completeness, yes, we can define a union with no alternatives, and just as with records, we can drop the = sign in such a case:

```
union Bottom
```

A union with no alternatives has precisely zero possible values. A function returning it can never halt, and a function taking it as an argument can never be called. It's fairly useless.

Now, we have records and we have unions, but why have both? Why not have them merged, like Haskell does? After all, any record, such as:

```
record Person = name : String, age : Int
```

can be made into a union:

```
union Person = Person String Int
```

All that's missing are the accessor functions.

Haskell solves this by adding a special "record syntax" to `data` (`union` in Haskell), which looks like this:

```
data Person = Person { name :: String, age :: Int }
```

The "record syntax" automatically generates all the needed accessor functions, and everything is fine. Why not take the same approach?

Three reasons.

First of all, anytime we define a record this way, we need to repeat the name of the record twice: once as the type name, the other time as the name of the constructor.

The second reason is that nothing prevents us to add more alternatives/constructors. The accessor functions will simply crash if applied to the other alternatives. This possibility makes no sense and has no use. A record has, logically, only one alternative.

And the last reason is that records are very useful. They might be a special case of unions, but an extremely common one. Singling them out does no injustice to anyone.

1.5.1 Switch/case

Now, we know how to construct union values, so how do we get any information out of them? The obvious thing that we'd like to do is to check the alternative, get the arguments, and do something with them.

In the early conceptions of the Funky language, this task used to be accomplished by, yet another, compiler generated function. At the time it was acceptable, because the language featured anonymous records, so the first argument to the function was the union and the second was an anonymous record, where each field represented one of the union's alternatives. It wasn't very aestetic, though. The support for anonymous records was eventually dropped and so had to be the abovementioned function.

The task of getting information out of a union was, of course, still needed, and so a new, special `switch`/`case` structure was introduced.

```
<switch-expression> ::= switch <expression> <cases>
<cases> ::= <nothing> | <cases> <case>
<case> ::= case <simple-variable> <expression>
```

The structure starts with the `switch` keyword followed by an expression we want to examine. The expression must evaluate to a value of a union type. It's followed by a list of cases. Each case starts with the `case` keyword and is immediately followed by the name of one of the union's alternatives. If the name is an infix identifier, it must be enclosed in parentheses. The name is a simple variable, but must not have a type annotation. The name of the alternative is followed by the case body, which must be a function that takes all the alternative's arguments and returns some result. All case bodies in a single switch must return a result of the same type.

All union's alternatives must be listed, each must be listed exactly once, and they must be listed in the same order as they are mentioned in the union's definition. The last condition might get dropped if it's shown that it brings some drawbacks, though none have been observed so far.

Let's take a look at some examples.

```
let false \axiom-of-choice
switch axiom-of-choice
case true  false
case false true
```

The above example manifests a disbelief in the axiom of choice. That's not important, though. The whole expression evaluates to `true`, because `axiom-of-choice` is `false` and the `false` case in the switch evaluates to `true`.

How about a union with arguments in alternatives? Here's a recursive function that counts the length of a list:

```
func length : List a -> Int =
    \list
    switch list
    case empty
        0
    case (::) \x \xs
        1 + length xs
```

In the case the list is empty, zero is simply returned. In the other case, when the list was formed using the `(::)` constructor, the case body is a function of two arguments. Say, the list was constructed with this expression: `1 :: 2 :: empty`, which can be parenthesized like this: `1 :: (2 :: empty)`. The first argument to the `(::)` function is `1` and the second argument is `2 :: empty`. Thus, when evaluating the switch, the case body of the `(::)` case gets passed the values used to construct the list, namely, the `x` argument gets the value of `1` and the `xs` argument gets the value of `2 :: empty`. Note, that in the `case`, we can't put the `::` identifier in-between `x` and `xs`, that would be nonsensical.

The case body of the `(::)` could've been written like this:

```
    case (::)
        const ((1 +) . length)
```

But that's a bit harder to read. What's important to realize, though, is that the case body doesn't need to explicitly name the arguments, it just has to be a function that fits the context, expressed in any viable way.

Here's how we could make functions for Peano arithmetics:

```
union Peano = zero | succ Peano

func pred : Peano -> Peano =
    \p
    switch p
    case zero
        zero
    case succ \p-1
        p-1

func (+) : Peano -> Peano -> Peano =
    \p \q
    switch p
    case zero
        q
    case succ \p-1
        p-1 + succ q

func (-) : Peano -> Peano -> Peano =
    \p \q
    switch q
    case zero
        p
    case succ \q-1
        pred p - q-1
```

Similarly, we could define `*`, `/`, and many other functions.

1.6 Aliases

Admittedly, aliases are the simplest type definition mechanism. They merely establish a rule that one type name stands for another type, albeit possibly more complex one. However, surprising typing power comes with recursive aliases, that is, aliases that refer to themselves, which we, unfortunately, won't discuss.

The grammar for aliases is this:

```
<alias> ::= alias <type-name> <variables> = <type>
<variables> ::= <nothing> | <variables> <type-variable>
```

All type variables listed between the name and the `=` sign can be used in the type on the right side.

The way to think about aliases is that whenever an alias is used in a type, it simply gets replaced by the right side of the definition with all the type variables rightly substituted.

The `String` type from the standard library is defined as an alias:

```
alias String = List Char
```

This choice may be retracted in the future and `String` may end up implemented internally in a more efficient manner.

Chapter 2
Code that reads top to bottom

Imperative languages, which feature structured programming paradigm (blocks of code, conditions, loops, etc.), have this incredibly useful property of vertical composability. Structured, imperative code can be read top to bottom, blocks of code understood independently of each other. Programmer scans the first part of the code, remembers invariants it imposes, proceeds to the next part of the code, and so on. This way of thinking comes very natural to Homo sapiens.

Functional languages tend to compose rather differently. A functional program, being an expression, has to be understood arguments first, function application second, which forms kind of an inside-out way of reading code. Unfortunately, this way is less natural and more difficult for humans. Functional languages compensate this deficit by introducing various syntactic features that make it possible to structure the code partially vertically. Features like pattern matching, let bindings, where bindings, guards, and Haskell's do notation all belong to this category. However, they don't solve the problem fully, because none of them nests very well.

Funky has two features that make it possible to omit pattern matching, let bindings built into the language, guards, or do notation, and still be able to write code that composes vertically, nests very well, and in many respects, resembles imperative code, yet is just a big, purely functional expression.

These two features are: trailing lambdas and the semicolon. In this chapter, we'll discuss the full extent of impact of these two features. In the next chapter, we'll see how these features scale to express intentions of side-effects, and allow constructing new, powerful functions that transform programs in ways impossible in other languages beknown to me.

Because seeing specific examples makes things easier to understand, we'll use list generators as our modus operandi. The techniques presented here on lists are just as well applicable to any substance suitable for vertical code. For example, the next chapter uses the same techniques in the context of expressing side-effects.

2.1 If and let

Let's start with two most common code structuring primitives – if and let. In most functional languages, these are built-in language constructs. Not in Funky. In Funky, if and let are just regular functions from the standard library and we'll soon see advantages of this approach. They're defined like this:

```
func if : Bool -> a -> a -> a =
    \cond \then \else
    switch cond
    case true  then
    case false else

func let : a -> (a -> b) -> b =
    \x \f
    f x
```

Let's examine `if` first. We can use it as a short conditional expression, like the ternary operator (`b ? t : e`) in C:

```
func string : Bool -> String = \bool if bool "true" "false"
```

That's simple.

Now, here comes the verticality. With the help of the semicolon, we can put the whole else-branch under the if expression. This makes it possible, and aesthetically pleasing, to put an arbitrarily large and vertical expression in place of the else-branch.

```
func fizzbuzz : Int -> String =
    \number
    if ((number % 15) == 0)
        "fizzbuzz";
    if ((number % 3) == 0)
        "fizz";
    if ((number % 5) == 0)
        "buzz";
    string number
```

Here's the infamous FizzBuzz problem. The then-branch of the first `if` is the string `"fizzbuzz"`. Notice, that we put a semicolon after `"fizzbuzz"`. This causes the whole

rest of the function take the role of the else-branch. The last line is the else-branch of the last `if` and thus acts as a "catch-all" – gets returned when all the conditions above fail.

In Haskell, if expression is composed of three keywords: `if b then t else e`. This, coupled with Haskell's whitespace semantics make it awkward to use the if expression for anything other than short conditional expressions. Use-cases like above are instead handled by guards, or pattern-matching in other cases. The shortcoming of these features, however, is that they can only be used at the top-level of a function, and don't nest.

Funky's `if` function nests very well. In the previous example, each then-branch was just a very simple short expression. That doesn't need to be the case. A then-branch can just as well be an arbitrarily large, vertical expression. Here's how:

```
if condition (
    then-branch
    ...
);
else-branch
...
```

We'll see concrete examples of this form later.

We can already see, though, that the `if` function in Funky can be used equally well for binary branching, series of cases, or simple ternary expressions. There's no need for guards.

Now we'll examine the `let` function. This function is different from `if`, because it doesn't exploit the semicolon in order to achieve code verticality, instead, it utilizes the notion of a trailing lambda instead.

The `let` function is used to assign a value to a variable to be used later in the code. For example:

```
let "me stay by your" \side
reverse side ++ " " ++ side
```

This expression evaluates to: `"ruoy yb yats em me stay by your"`. What's really happening here is an application of the `let` function with two arguments: `"me stay by your"` and the function `\side reverse side ++ " " ++ side`. But, we can instead think of it as a vertical composition of the two lines, where the first line assigns the `side` variable, while the second line returns the result of the whole expression. This kind of thinking is the key benefit of code that reads top to bottom.

In general, this is how we use `let`:

```
let value \variable
next-code
...
```

The "reversed order" of the assignment (value first, variable second) may seem weird at first, but takes no time getting used to.

If we want to assign multiple variables, we just stack them up:

```
let (filter prime? (count 2))   \primes
let (take-while (< 100) primes) \small-primes
let (length small-primes)       \num-small-primes
"there are " ++ string num-small-primes ++ " small primes (< 100)"
```

This expression evaluates to: `"there are 25 small primes (< 100)"`, which is true. The `count` function returns an infinite list of integers starting from the given number and increasing by one.

Each variable introduced by `let` can refer to all variables introduced above it, but it can't refer to itself – it's not possible to straightforwardly define a variable recursively. This can be fixed with the `fix` function, and we'll talk about it more later in this chapter.

2.2 Aiding verticality

There are two kinds of functions that help us write code that reads top to bottom. The first kind exploits the semicolon, while the second kind makes use of trailing lambdas. We'll call these functions *vertical functions*. If and let are prime examples of the two kinds of vertical functions.

The first kind is: semicolon kind vertical functions. They are characterized by their signature, which has this general form (`V` is an arbitrary type):

```
... -> V -> V
```

They take some arguments, these arguments are put before the semicolon, then they take a "continuation" after the semicolon, and they return some transformation of the continuation based on the arguments. "If" is an example of a semicolon kind vertical function.

The second kind is: trailing lambda kind vertical functions. They are also characterized by their signature, this is their general form (`V` is, again, an arbitrary type):

```
... -> (... -> V) -> V
```

The signature is similar to the first kind of vertical functions, except the "continuation" takes some arguments, which get supplied by the vertical function. Semicolon is no longer needed as the trailing lambda does the job. "Let" is an example of a trailing lambda kind vertical function.

Most functions are not vertical, nor they ought to be. Instead, authors of libraries should carefully craft a set of vertical functions for use-cases when it makes sense. Some libraries need not provide any vertical functions at all, while others are built around them (such as libraries for expressing side-effects).

2.3 Lists as generators

Let's recall the definition of the standard `List` type:

```
union List a = empty | a :: (List a)
```

The type of the `(::)` constructor is `a -> List a -> List a`. However unintuitive it may seem at first, this type perfectly fits the semicolon kind vertical function schema. Being infix makes it hard to use vertically, though, so let's give it an alternative name:

```
func yield : a -> List a -> List a = (::)
```

The name "yield" is intentionally reminiscent of Python's generator syntax. Generators in Python are basically lazy lists in an otherwise strict and imperative language. Since lists in Funky are lazy anyway, we can treat them just like generators in Python.

Now that we have `yield`, we can start constructing lists vertically:

```
func counting-on-fingers : List Int =
    yield 1;
    yield 2;
    yield 3;
    yield 4;
    yield 5;
    empty
```

The `count` function counts natural numbers starting from a given number. It simply takes an integer `n` and returns an infinite list `[n, n + 1, n + 2, ...]`.

Here's how it looks vertical:

```
func count : Int -> List Int =
    \from
    yield from;
    count (from + 1)
```

It yields the first number and recursively continues counting from the next integer, which produces the desired result.

A slightly less trivial function is `range`, which is similar to `count`, except it also takes the upper-bound as an argument. It takes two numbers – `from` and `to` – and returns a finite list `[from, from + 1, ..., to - 1, to]`.

```
func range : Int -> Int -> List Int =
    \from \to
    if (from > to)
        empty;
    yield from;
    range (from + 1) to
```

A vertical use of `if` is involved as well, which shows how vertical functions can be (usually, but not always) seamlessly combined.

Here's another relatively trivial function, which duplicates each element in a list, producing a new, twice as long, list.

```
func dup-elements : List a -> List a =
    \list
    if (empty? list)
        empty;
    let (first list) \x
    yield x;
    yield x;
    dup-elements (rest list)
```

For example, `dup-elements [1, 2, 3]` returns `[1, 1, 2, 2, 3, 3]`. In this function, we see a seamless simultaneous use of `if`, `let`, and `yield`.

The Collatz conjecture says that if we take any natural number and repeatedly apply a simple procedure to it, which will be described immediately, we'll eventually reach the number one. The procedure is as follows: if the number is even, divide it by

2, otherwise multiply it by 3 and add 1. For example, let's start with 6. It's even, so we need to divide it by 2 and we get 3. That's odd, so the next number is $3 \cdot 3 + 1 = 10$. Next is 5, then 16, then 8, 4, 2, and finally 1.

Here's a function that takes a number and returns a list containing successive applications of the procedure described in the Collatz conjecture, including the first number and the final 1:

```
func collatz : Int -> List Int =
    \number
    yield number;
    if (number == 1)
        empty;                    # reached one, we're done
    if ((number % 2) == 0)
        (collatz (number / 2));  # divide by 2
    collatz (1 + number * 3)     # multiply by 3 and add 1
```

For example, `collatz 6` returns `[6, 3, 10, 5, 16, 8, 4, 2, 1]`. Starting from another number, `collatz 7` returns `[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]`.

Notice how we use `if` to eliminate cases one by one. This makes it very readable and understandable.

As a last example in this section, we'll implement the merge sort algorithm. The algorithm works in the following manner: if we've got a list of length one or less, return the list; otherwise, divide it into two halves, recursively sort those, and finally merge them using the merge function. Merge function takes two sorted lists and merges them into one sorted list by sequentially picking the next smallest element from their fronts. For simplicity, we'll sort a list of integers. General sorting would simply add one more argument: the comparison function.

Here's a nice vertical implementation of the merge function:

```
func merge : List Int -> List Int -> List Int =
    \left \right
    if (empty? left)
        right;
    if (empty? right)
        left;
    if (first left <= first right) (
        yield (first left);
```

```
        merge (rest left) right
    );
    yield (first right);
    merge left (rest right)
```

For example, `merge [1, 3, 5] [2, 5, 6]` returns `[1, 2, 3, 4, 5, 6]`. Now we put the merge function to use in the complete merge sort algorithm:

```
func merge-sort : List Int -> List Int =
    \numbers
    if (length numbers <= 1)
        numbers;
    let (length numbers / 2) \half
    let (take half numbers)  \left
    let (drop half numbers)  \right
    merge (merge-sort left) (merge-sort right)
```

Here we use both if and a series of let assignments, once again, demostrating the vertical composability.

2.4 Function `when`

Sometimes it's desirable conditionally "execute" one piece of vertical code and then have it followed by another piece of code that is to be "executed" unconditionally, no matter what. The word "execution" is rightfuly put inside quotes here: vertical code intentionally looks imperative, but in reality it's just a declarative way of constructing a data structure, or composing functions.

As a primitive example, say we want to make a function that given an integer returns a singleton list containing just that integer if the integer is even, or it returns a list containing that integer prepended by the previous integer if the integer is odd. We might write such function like this:

```
func useless : Int -> List Int =
    \number
    if ((number % 2) == 1)
        [number - 1, number];
    [number]
```

Using the `yield` function, we can rewrite the function as follows, which is longer, but serves our purpose better:

```
func useless : Int -> List Int =
    \number
    if ((number % 2) == 1) (
        yield (number - 1);
        yield number;
        empty
    );
    yield number;
    empty
```

We see, that in both cases, we want to yield the given number, but in the case that the number is odd, we also want to yield the previous number. There's a function called `when` in the standard library for these situations. Here's the definition:

```
func when : Bool -> (a -> a) -> a -> a =
    \cond \then \next
    if cond (then next) next
```

If the condition is true, `when` evaluates to `then next` (or `then; next` for better understanding), while in the other case it evaluates to just `next`.

In our case of the `useless` function, we want to evaluate to:

```
yield (number - 1);
yield number;
empty
```

in the case the condition is true, and otherwise we just want to evaluate to:

```
yield number;
empty
```

Clearly then, we can make good use of `when`:

```
func useless : Int -> List Int =
    \number
    when ((number % 2) == 1)
        (yield (number - 1));
    yield number;
    empty
```

The code is now clearer, shorter, and no longer repetitive.

Another good example is the `filter` function, which takes a predicate and a list and returns the same list with all the elements that don't pass the predicate left out. Here's a common way to define it:

```
func filter : (a -> Bool) -> List a -> List a =
    \p \list
    if (empty? list)
        empty;
    if (p (first list)) (
        yield (first list);
        filter p (rest list)
    );
    filter p (rest list)
```

As we can see, both branches in the second if expression end with the same line. This can be improved with the `when` function:

```
func filter : (a -> Bool) -> List a -> List a =
    \p \list
    if (empty? list)
        empty;
    when (p (first list))
        (yield (first list));
    filter p (rest list)
```

Now we've ended up with a very clear and concise `filter` implementation.

How about nesting more "commands" inside the body of `when`? It's not so straightforward: this doesn't type check, because `yield` expects a list as its second argument, not a `List a -> List a` function:

```
when true (
    yield 1;  # => List a -> List a, i.e. expected arg is List a
    yield 2   # => List a -> List a, cannot be argument to above
);
yield 3;
yield 4;
empty
```

The problem is solvable, and the solution finds wide use in many other situations. The trick is to include an explicit argument marking the code after the `when` function, like this:

```
when true (
    \next
    yield 1;
    yield 2;
    next
);
yield 3;
yield 4;
done
```

Another solution to the problem would be to utilize function composition instead of the semicolon:

```
when true
    (yield 1 . yield 2);
yield 3;
yield 4;
done
```

This solution is fine for yielding, however, it doesn't scale well for certain trailing lambda kind vertical functions, as we'll see in the next chapter.

2.5 Inline recursion with `fix`

Another thing that's sometimes desirable to do inside a function is to spin up a recursive computation that needs to remember more values than what's provided in the function's arguments, which prevents making the function straightforwardly recursive.

A prime example of this phenomenon is the list reverse function. The idea is, that we start with an empty list and we successively prepend elements of the input list to it until we drain the input list, at which point we get the input list reversed. The recursive computation here needs to remember two lists: the accumulated reversed prefix of the input list, and the remaining suffix of the input list. However, the reverse function itself only takes one argument and thus simple recursion is out of the question.

The first solution is to define a helper function with the accumulator:

```
func reverse : List a -> List a = reverse-helper []

func reverse-helper : List a -> List a -> List a =
    \left \right  # reversed prefix, remaining suffix
    if (empty? right)
        left;
    reverse-helper (first right :: left) (rest right)
```

This works, but isn't nice. The `reverse-helper` function has only a single use: to be called by the `reverse` function. That's not enough reason to make it a top-level function.

What we'd really like to do is to somehow wire the `reverse-helper` recursion inside `reverse`. That's where the `fix` function comes in. The name "fix" stands for "fix-point combinator", which is notoriously difficult to understand how it really works. We'll explain what `fix` does in a very understandable and useful way – understanding why it does that is another challenge that we won't attempt here. Although, in the end, it's not actually that hard. Here's the definition of `fix`:

```
func fix : (a -> a) -> a = \f f (fix f)
```

As we've said, we'll make no attempts at understanding this definition. Instead, here's how we'll think about the `fix` function. Whenever we write:

```
fix \f expression
```

we think of it as a function defined by this formula:

```
f = expression
```

The trick is that the identifier *`f`* may occur in *`expression`*, which makes it possible to define *`f`* recursively.

For example, here's the factorial function expressed using the `fix` combinator:

```
fix \fac \n
    if (n <= 0)
        1;
    n * fac (n - 1)
```

In this case, the `fix` functions gets specialized to this type:

```
((Int -> Int) -> Int -> Int) -> Int -> Int
```

And since we're passing the argument of type `(Int -> Int) -> Int -> Int`, the type of the whole expression is `Int -> Int`. This whole expression can be put into parentheses and called as a function:

```
(fix \fac \n
    if (n <= 0)
        1;
    n * fac (n - 1)) 5
```

The result of the above expression is `120`, which is the 5!. This works, but is not particularly aestetically pleasing. The whole recursive expression needs to be in parentheses and the argument comes at the end which makes it easily missed when skimming over the code.

Instead, we can utilize the `|>` function from the standard library to "insert" an argument from the left side. The `|>` function is defined like this (there a few more overloaded versions in the standard library):

```
func (|>) : a -> (a -> b) -> b = \x \f f x
```

Notice that its type and definition are identical the those of `let`. The main difference is that `|>` is infix and used in different situations.

Utilizing `|>`, we can rewrite the above expression as follows:

```
5 |> fix \fac \n
    if (n <= 0)
        1;
    n * fac (n - 1)
```

The result is, once again, `120`, but the expression is more readable.

With the acquired knowledge, let's free `reverse` from its extra helper function:

```
func reverse : List a -> List a =
    [] |> fix \loop \left \right
        if (empty? right)
            left;
        loop (first right :: left) (rest right)
```

Using the `fix` combinator to express loops and inline recursion may seem a little awkward at first, but getting used to it is not a big deal.

Let's take a few more examples. Here's a function that lists all natural numbers, including 0:

```
func naturals : List Int =
    0 |> fix \loop \n
        yield n;
        loop (n + 1)
```

Given a `prime?` function, which takes an integer, checks whether it's a prime or not, and returns a boolean accordingly, here's a function that lists all primes:

```
func primes : List Int =
    2 |> fix \loop \n
        when (prime? n)
            (yield n);
        loop (n + 1)
```

And here's a linear-time function that lists the Fibonacci sequence:

```
func fibonacci : List Int =
    1 |> 0 |> fix \loop \x \y
        yield x;
        loop y (x + y)
```

Here we pass two arguments to the loop. Because `|>` passes arguments from the left side, we need to pass them in a seemingly reverse order. This is another thing that might seem initially awkward when using `fix`.

2.6 "Non-determinism"

The idea of non-determinism in programming comes with a little misleading name – computers are deterministic, after all. It is, however, possible to express a deterministic program in the "language of non-determinism". This approach gives an incredible expressive power for certain kinds of problems.

Here's the core idea of non-determinism we're going to use: there's a list of options – pick one of them (randomly/non-deterministically), and produce some results based on the choice. What are all the possible results we can produce?

For example, here's the list of options: `[1, 2, 3, 4, 5]`. We pick one of them, and we produce a number one greater. So, in case we picked `3`, we produce `4`. What are all the possible results? They're `[2, 3, 4, 5, 6]`. Here's another example: we'll use the same list of options. We pick one of them, and if we picked an odd one, we don't produce anything. If we, however, picked an even one, we produce it as it is. This time, all the possible results are these: `[2, 4]`.

One of the most beautiful problems demostrating the power of non-determinism is the task of finding all permutations of a list. Using the language of non-determinism, we can express the solution very clearly. In case we've got an empty list, we produce one permutation: that empty list. Otherwise, a pick one of the permutations of the rest of the list (the list without its first element). Then we pick a place in the permutation and insert the first element of the original list there. Then we produce what we've got. Clearly, all the possible results produced by this function are precisely all the permutations of the original list. As we'll see, the actual expression in code is no more complicated than what we've just described.

To express non-determinism, we need to make a `pick` function, which "picks" from the a list of options and accumulates all the produced results. Here's how it looks like:

```
func pick : List a -> (a -> List b) -> List b =
    \list \f
    if (empty? list)
        [];
    f (first list) ++ pick (rest list) f
```

The `pick` function simply runs each element of the list through the given function. The function returns a list of produced results per element. Finally, `pick` concatenates all the obtained results into the final list.

Conveniently, `pick` is a trailing lambda kind vertical function. And that's exactly how we're going to use it.

Here's the first mentioned example:

```
pick [1, 2, 3, 4, 5] \x
yield (x + 1);
empty
```

The above expression evaluates, as expected, to `[2, 3, 4, 5, 6]`. Here's the next mentioned example:

```
pick [1, 2, 3, 4, 5] \x
when ((x % 2) == 0)
    (yield x);
empty
```

In this example, we've vertically combined the `pick` function and the `when` function in a seamless manner. The result is `[2, 4]`.

In fact, the `filter` function can be implemented using `pick`:

```
func filter : (a -> Bool) -> List a -> List a =
    \p \list
    pick list \x
    when (p x)
        (yield x);
    empty
```

And the `map` function, too:

```
func map : (a -> b) -> List a -> List b =
    \f \list
    pick list \x
    yield (f x);
    empty
```

This, however, is not the forte of non-determinism. To see its full power, let's implement the permutations.

The first function that we need will be called `insert`. This function takes an element and a list and returns all possible "insertions" of the element into that list. Here's an example: `insert 3 [1, 2]` evaluates to `[[3, 1, 2], [1, 3, 2], [1, 2, 3]]`. The original list remained in order, all we did is we inserted the extra element in all possible places. So, here's how we implement `insert`:

```
func insert : a -> List a -> List (List a) =
    \x \list
    yield (x :: list);
    if (empty? list) empty;
    pick (insert x (rest list)) \tail
    yield (first list :: tail);
    empty
```

First, we need to produce the case when we insert the element at the beginning of the list. Then, if the list is actually empty, we don't want to do anything more. If it's not empty, we non-deterministically pick one of the ways we can insert the element into the rest of the list and we produce all these possibilities, except we also prepend the missing first element of the original list.

Now it's time to make the `permutations` function. This function gets a list and returns a list of all permutations of the list. For example: `permutations [1, 2, 3]` should return `[[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]`.

Here's how we implement `permutations` with the help of `insert`:

```
func permutations : List a -> List (List a) =
    \list
    if (empty? list)
        (yield []; empty);
    pick (permutations (rest list)) \tail
    pick (insert (first list) tail) \perm
    yield perm;
    empty
```

First, if we're dealing with an empty list, we just produce one result: that empty list. Otherwise, we pick one of the permutations of the rest of the list. Then we pick one of the possible insertions of the first element to that permutation and we produce the result. That's all.

The idea of non-determinism is not suitable for all problems. The situations where it shines usually involve exploring some space of possibilities. In those situations, however, the expressiveness of non-determinism shows to be mesmerizing.

Chapter 3
Side-effects and interpreters

Pure functional languages define functions that transform data structures. It's data in, data out. At no point is there a place for an action, a side-effect. However, we need to do side-effects or else our programs are just useless declarations of computation.

The topic of side-effects has been one of the most discussed in the field of functional languages. Many languages, such as LISP solve the problem of side-effects by introducing impurity. In pure and lazy languages, such as Funky, introduction of side-effects usually requires some kind of trickery proven to not be too harmful. Two most prominent techniques for managing side-effects in pure functional programs happen to be monads and uniqueness types. Monads[10] are used by Haskell, Idris, and others, while uniqueness types[8] are used by languages such as Clean.

Both monads and uniqueness types are approching the problem of side-effects by attempting to solve a rather different problem: how to make sure that a certain sequence of expressions gets evaluated in a certain order? In a lazy languages, such thing is not straightforwardly guaranteed by the runtime and instead has to be made guaranteed by some kind of data structure or a limitation imposed by the type system.

Monads make the evaluation order guaranteed by composing atomic monadic actions in such a way that the next monadic action cannot be evaluated before the previous one, because it requires the result of the previous action for its evaluation.

Uniqueness types make the same guarantee by imposing a type system limitation, that at one point, there may only exist a single reference to a value of a uniqueness type. This guarantees that all function applied to a value of a uniqueness type must be evaluated in a predictable sequential order.

Once the guarantee of evaluation order is established, runtime may insert side-effecting operations to evaluations of certain runtime-supported functions. The language gains side-effects while keeping all the benefits of purity and laziness.

We believe that both monads and uniqueness types are tackling the wrong problem. We don't believe that guaranteeing the evaluation order is the key to solving side-effects in pure and lazy functional languages.

In this chapter, we'll explore Funky's approach to side-effects. The approach is very comprehensible: it doesn't introduce any concepts to the language, aside from those already discussed. Of course, enforcement of evaluation order sometimes emerges, but

it's more of an accident, rather than an intention. The approach is also extremely flexible: it allows performing arbitrary transformations of the whole program – for example, encoding the whole I/O with AES is just one function application away. The approach taken by Funky is so obvious, that it's quite curious that no other language, to our knowledge, has adopted it before.

3.1 Interpreters

A program in Funky is just a value. There's no built-in type for side-effects, like `IO` in Haskell, in fact, nothing regarding side-effects is built-in. Everything is a value of either built-in or user-created types.

The thing is, that a value may become a subject of interpretation. A value of a particular type may be viewed as recipe for side-effects. Then, a specialized program, an *interpreter*, takes this value, examines it, and actually executes the side-effects described by the value. This is the approach taken by Funky.

At this day, Funky's compiler and runtime is written in the Go programming language. Since interpreters interact with the runtime, they too have to be written in Go. In future, this may be expanded to other languages as well (if so, it will be possible to write interpreters in C++, for example).

This section doesn't assume much familiarity with Go, but it assumes familiarity with imperative programming and ability to intuitively understand programs written in an unfamiliar, but not alien programming language.

Now we'll walk through the tools Funky provides to create interpreters. An interpreter is usually a simple program. The first thing that an interpreter does is it calls the `funky.Run` function from the `"github.com/faiface/funky"` package. Here's how it looks in Go:

```
package main

import "github.com/faiface/funky"

func main() {
    program := funky.Run("main")
}
```

This program wouldn't compile, yet, because the `program` variable is unused, and Go disallows unused variables.

The `funky.Run` function does a lot of things. It's the powerhouse of any Funky interpreter. It reads command-line flags, loads Funky files supplied in command-line arguments, parses them, type-checks them, and compiles them into the internal runtime

representation. If an error occurs during any of the mentioned steps, `funky.Run` reports the error to the standard error output and terminates the program.

As we can see, `funky.Run` takes one argument. This argument is the name of the function we'd like to interpret in our interpreter. It's usually called `main`, but doesn't have to be. There mustn't be multiple overloaded versions of this function, only one is allowed, `funky.Run` will report an error and terminate otherwise.

The value returned from `funky.Run` is of type `*runtime.Value` (the `*` symbol means pointer), which is a type from the `"github.com/faiface/funky/runtime"` package. This value is the compiled representation of the `main` function from the source files supplied on the command-line, that we can interact with.

The `*runtime.Value` type has these methods (bodies omited):

```
func (*runtime.Value) Char() rune
func (*runtime.Value) Int() *big.Int
func (*runtime.Value) Float() float64
func (*runtime.Value) Field(i int) *runtime.Value
func (*runtime.Value) Alternative() int
func (*runtime.Value) Apply(arg *runtime.Value) *runtime.Value

// these three are implemented using the above six
func (*runtime.Value) Bool() bool
func (*runtime.Value) List() []*runtime.Value
func (*runtime.Value) String() string
```

The syntax of method in Go starts with the `func` keyword, which is followed by the receiver argument (the value before the "dot" in a method call), then we see the method name followed by the list of arguments in parentheses, and finally the return type.

All of the above method are used to interact with a runtime value. Each method only works with certain types of values. If a method is used with a wrong type of value, it simply panics (crashes the program). However, Funky's type system always makes sure that types are used consistently in a program, and so the programmer of an interpreter can, with absolute certainty, expect the types of runtime values without any runtime examination. Runtime examination of types is not possible anyway, not yet, at least.

The first three methods – `Char`, `Int`, and `Float` – return the actual value of a runtime value of one of the primitive, built-in, types. Note, that the return type of `Int` is `*big.Int`, an arbitrary precision integer, and not just `int`, which would be either 32-bit or 64-bit integer.

The next method – `Field` – is used to retrieve record and union fields. Of course, if the runtime value is not a record or a union, `Field` panics. If the runtime value is a record, `Field` returns the `i`-th field of the record, in the order mentioned in the record's definition, starting from 0. If the runtime value is a union, `Field` returns the `i`-th argument in the current union's alternative, again, indexing from 0.

The `Alternative` method works only on unions, and returns the number of the alternative the union value has. For example, in `union List a = empty | a :: (List a)`, `empty` is the alternative number 0, and `::` is the alternative number 1.

And finally, the `Apply` method works on functions, it takes another runtime value and passes it as an argument to the function, returning the result.

There are three more methods, but they are simply implemented using the above six, and it's clear what they do.

We also need to be able to create new runtime values, mostly for passing them as arguments to functions. The `"github.com/faiface/funky/runtime"` package offers these functions for that purpose:

```
func MkChar(c rune) *runtime.Value
func MkInt(i *big.Int) *runtime.Value
func MkFloat(f float64) *runtime.Value
func MkRecord(fields ...*runtime.Value) *runtime.Value
func MkUnion(alt int, fields ...*runtime.Value) *runtime.Value

// these three are, again, implemented using the above five
func MkBool(b bool) *runtime.Value
func MkList(elems ...*runtime.Value) *runtime.Value
func MkString(s string) *runtime.Value
```

The three dots (`...`) are Go's syntax for variadic arguments.

As we might have noticed, there's no `MkFunc` function, an interpreter can't create functions on it's own, just values. There a few reasons, first, it would be rather complicated. It would also require some kind of runtime compilation, which would be expensive. Overall, the costs and trouble associated with adding this feature largely outweights the marginal benefits it brings.

With the acquired knowledge, we are ready to built a simple interpreter. Our first interpreter will interpret a `main` function of type `Int -> Int`, very simple. It wil simply read a number from the standard input, pass it to the function, and print the result back out.

Here's the code of the interpreter, let's call it `intfunc`:

```
// intfunc.go

package main

import (
    "fmt"
    "math/big"

    "github.com/faiface/funky"
    "github.com/faiface/funky/runtime"
)

func main() {
    program := funky.Run("main")
    number := new(big.Int)
    fmt.Scan(number)
    result := program.Apply(runtime.MkInt(number))
    fmt.Println(result.Int())
}
```

Now we compile the interpreter to an executable called `intfunc` and we're ready to use it. Here's the Funky program we're going to test it on, the classic factorial function:

```
# factorial.fn

func n! : Int -> Int =
    \n
    if (n <= 0)
        1;
    n * n! (n - 1)

func main : Int -> Int = n!
```

Since Funky has no support for packages and imports yet, we need to include all the files of the standard library in the command-line arguments to the interpreter. Say all the files of the standard library are in the `stdlib` directory.

Now, we can run the program (user input is emphasized, `$` signifies the shell prompt):

```
$ intfunc stdlib/*.fn factorial.fn
5
120
```

It worked!

This gave us an idea of interpreters, how they work and interact with Funky's runtime.

3.2 Interactive command-line programs

Now we'll move on to implement a more serious, albeit still very simple, interpreter for interactive command-line programs. Then we'll see how we can put the techniques for vertical code, described in the previous chapter, to use when writing programs for this interpreter. All the ideas described in this section are generally applicable in virtually any other interpreter.

The capabilities of the interpreter we'll be studying in this section are rather simple: printing characters to the standard output, reading characters from the standard input, and ending the program. The data structure to be interpreted by this interpreter is strikingly simple. We'll call it `IO`:

```
union IO = done | putc Char IO | getc (Char -> IO)
```

It's a union with three alternatives. The first one signifies that the program is finished and nothing more is to be done. The second one has two arguments. The first argument is a character to be printed to the standard output. The second argument is what's to be done next, it's the rest of the program. The third alternatives has a single argument: a function. This function takes a character from the standard input and returns the rest of the program.

This simple union is capable of representing all possible command-line programs of this kind. All of them somehow alternative between printing and reading characters, until they eventually finish.

At the first sight, it might seem that expressing actual command-line programs using this data structure will be hard. The opposite is true. Function `putc` is a semicolon kind vertical function. Likewise, `getc` is a trailing lambda kind vertical function. These can be combined into more complex, more expressive functions, like `print`, `scan`, and a lot more, as we'll see later in this chapter.

Before we dive into programming command-line programs, we need the interpreter. Here it is:

```
package main

import (
    "bufio"
    "io"
    "os"

    "github.com/faiface/funky"
    "github.com/faiface/funky/runtime"
)

func main() {
    program := funky.Run("main")
    in, out := bufio.NewReader(os.Stdin), bufio.NewWriter(os.Stdout)
    defer out.Flush()
loop:
    for {
        switch program.Alternative() {
        case 0: // done
            break loop
        case 1: // putc
            out.WriteRune(program.Field(0).Char())
            program = program.Field(1)
        case 2: // getc
            out.Flush()
            r, _, err := in.ReadRune()
            if err == io.EOF {
                break loop
            }
            program = program.Field(0).Apply(runtime.MkChar(r))
        }
    }
}
```

The real interpreter for the `IO` data structure in the Funky's repository, called `funkycmd` (located at `interpreters/funkycmd/` in the Funky's repository tree), is nearly iden-

tical to this one, except it also adds proper error handling of edge conditions. We've omited those here, for simplicity.

Here's how the interpreter functions. First it sets up buffered input and output, for better performance. Then it enters a loop, where in each iteration it either prints a character, reads a character, or ends the program. Inside the loop, we check the `program`'s alternative, so we know what to do.

If the alternative is 0, we end the program.

If the alternative is 1, we're supposed to print a character in the first argument of the `putc` alternative. We retrieve the field using the `Field` method and get its value using the `Char` method. Then we print it. After printing the character, we move on to the rest of the program, which is in the second argument of the `putc` alternative.

And lastly, if the alternative is 2, we're supposed to read a character. First, we flush all that we've buffered for printing, user needs to see it now. Then, we read the character. If we've encountered EOF, we abruptly end the program. Otherwise, we pass the character as an argument to the function in the first and only argument of the `getc` alternative, and we continue executing the result of this application.

Let's see if our interpreter works.

```
# test.fn

union IO = done | putc Char IO | getc (Char -> IO)

func main : IO =
    putc 'a';
    putc 'b';
    putc 'c';
    putc '\n';
    done
```

And we run it:

```
$ funkycmd stdlib/*.fn test.fn
abc
$
```

The program printed `abc`, including a newline, just as expected.

From now on, we'll assume that the definition of the `IO` data structure is in a file inside `stdlib/funkycmd` directory and so we'll be running our programs using this command:

```
$ funkycmd stdlib/*.fn stdlib/funkycmd/*.fn test.fn
```

Also, from now on, we'll not mention this command, and we'll simply show the interaction that follows.

Here's a "cat" program, a program that copies the standard input to the standard output:

```
func main : IO =
    getc \c
    putc c;
    main
```

It's an infinite recursive definition of the `IO` data structure, but we can just think of it as a recursive function that reads a character, prints it back, and repeats over and over. Here's an example of its running:

```
hello, cat!
hello, cat!
do you cat?
do you cat?
you do cat!
you do cat!
^D
```

At the end, user pressed the Ctrl+D combination which causes the end of file and so the program stopped.

Nothing much interesting can be achieved in a reasonable manner using just `putc`, `getc`, and `done`. In the following sections, we'll explore all kinds of ways we can abstract high-level functions on top of these basic three. And it's going to be much more powerful than monads, uniqueness types, or imperative languages.

3.2.1 `print`, `println`, `scanln`

The `putc` function prints a character. It doesn't technically print it, the interpreter does, but we can think about it that way. A string is a list of characters. It should be easy to write a recursive function that transforms a string into a series of `putc` applications. Additionally, we'd like this function to be vertical. Here's that function:

```
func print : String -> IO -> IO =
    \s \next
```

```
    if (empty? s)
        next;
    putc (first s);
    print (rest s);
    next
```

Alternatively, we can use the right fold:

```
func print : String -> IO -> IO = \s \next fold< putc next s
```

What `print` does is it transforms a string into a series of `putc` applications, where the last `putc` application is followed by the `next` argument, i.e. what the `print` was followed by.

For example: both `print "ab"; done`, and `print "a"; print "b"; done` evaluate to `putc 'a'; putc 'b'; done`.

Now that we have `print`, we can easily make a convenience function `println`, which is same as `print`, except it additionally prints a newline:

```
func println : String -> IO -> IO = print . (++ "\n")
```

A simple function composition does the job.

Finally, we can write the "Hello, world!" program in Funky!

```
func main : IO =
    println "Hello, world!";
    done
```

It runs:

```
Hello, world!
```

Now that we can print lines, reading lines is the logical next step. Reading the input character by character is rather inconvenient, and making a function for reading whole lines is easy. Here it is:

```
func scanln : (String -> IO) -> IO =
    \f
    "" |> fix \loop \s
        getc \c
        if (c == '\n')
            (f (reverse s));
        loop (c :: s)
```

The `scanln` function makes use of the inline recursion for looping, as we've learned it in the previous chapter. The loop has a single variable to remember: a line accumulator. Each iteration reads a character, and checks if it's a newline. If it is, then the line has been successfully scanned and needs to be passed to the lambda. We need to reverse it, though, because we're accumulating the line by sequentially prepending characters to the accumulator. If the read character is not a new line, we simply prepend it to the accumulator and continue the loop.

Functions `print`, `println`, and `scanln` are already a good toolbox for creating all kinds of interactive programs. Here's a simple greeting program:

```
func main : IO =
    print "What's your name? ";
    scanln \name
    println ("Hello, " ++ name ++ "!");
    done
```

Here's what it does:

```
What's your name? Michal
Hello, Michal!
```

Here's a more complex program. This program plays a game with the user. The user chooses a number between 1 and 100, and the program proceeds to guess the number. With each guess, the user has to respond, whether the guess is correct, or whether the number he chose is less or more than the guessed one. The program uses binary search to efficiently guess the number:

```
func main : IO =
    println "Think of a number between 1 and 100.";
    100 |> 1 |> fix \loop \min \max
        let ((min + max) / 2) \mid
        print ("Is your number " ++ string mid ++ "? ");
        scanln \response
        if (response == "less")
            (loop min (mid - 1));
        if (response == "more")
            (loop (mid + 1) max);
        if (response == "yes")
            (println "Yay!"; done);
        println "You have to respond one of less/more/yes.";
```

```
        loop min max
```

The program starts a loop in which it remembers the lower and the upper bound of the number it's guessing. Then it prints out a guess and asks the user for a response. Depending on the response, it either proceeds with updated bounds, or it ends the program proclaiming happiness. If the response wasn't a valid one, it says so, and continues the loop with unchanged bounds.

A sample running of the program:

```
Think of a number between 1 and 100.
Is your number 50? no
You have to respond one of less/more/yes.
Is your number 50? less
Is your number 25? more
Is your number 37? more
Is your number 43? less
Is your number 40? more
Is your number 41? more
Is your number 42? yes
Yay!
```

This program combines various verical functions. It has a loop, series of cases, both semicolon and trailing lambda kind vertical functions, all nesting and combining seamlessly. A similar program in Haskell would have to be split to multiple functions, or it'd look quite messy.

The difference here isn't in the approach to side-effects. Instead, the whole difference is in syntax. Funky makes it easy to use vertical functions, while Haskell makes it messy and verbose. In Haskell, the equivalent of Funky's concise semicolon is the eye-catching `$` function. Anonymous lambda abstractions take more space, because their arguments need to be followed by an arrow: `->`. Small syntactic differences can have a huge impact on viability of constructs.

3.2.2 `ungetc, skip-whitespace, scan`

One very useful input function reads whitespace-delimited words of text, that is, whenever used, it first skips all the whitespace until it reaches a non-whitespace character. Then proceeds to accumulate all non-whitespace characters until it reaches a whitespace character again, at which point it hands over the accumulated result and stops reading. We'd like to make such a function, and call it `scan`. White `scanln` reads whole lines, `scan` reads words.

For example:

```
func main : IO =
    print "Type two words: ";
    scan \word-1
    scan \word-2
    println (word-1 ++ ", " ++ word-2);
    done
```

This program runs like this:

```
Type two words:    uptown


     funk
uptown, funk
```

As you can see, the user inserted a fair amount of whitespace around both words, including newlines, but `scan` read the words correctly anyway.

As we've already described, `scan` needs to first skip all the whitespace in the input. Could we make a general `IO` function for skipping whitespace? Something with this signature?

```
func skip-whitespace : IO -> IO
```

How would such a function work? It would first read a character, then check if it's a whitespace character. If it is, it would just recursively continue the same routine. In the other case, though, the character isn't a whitespace and thus isn't supposed to be skipped. But we've already read it. We'd need to somehow put it back on the input, so the following reading procedures can pick it back up.

This wouldn't be possible with an imperative language, or even with monadic `IO` in Haskell, without some kind of explicit access to the inner-workings of the input system, or without special support from the I/O library.

In C, such a function exists, it's called `ungetc`. It's a 100 line function, deals directly with the input buffers, sometimes creates its own "ungetc buffer", and touches a lot of flags. In short, it's nothing we'd be able to just make on a request. In Haskell, we found no equivalent, and to our knowledge, no equivalent can be made by the user of Haskell, as it would require a direct support from the `IO` monad.

In Funky, the situation is different. What makes the difference? `IO` is not a low-level imperative mechanims. `IO` is not an opaque, compiler-specific, data structure. `IO` is a data structure like any other. Most importantly, it's fully transparent. This makes it possible to make functions that not only prepend `putc`/`getc` operations, but transform it arbitrarily. The first demonstration of this power is the `ungetc` function.

Here's how we'll implement it:

```
func ungetc : Char -> IO -> IO =
    \c \io
    switch io
    case done
        done
    case putc \d \jo
        putc d;
        ungetc c;
        jo
    case getc \f
        f c
```

The idea is, that `ungetc` takes a character and an `IO`, and it searches through that `IO` until it finds the first `getc` command. When it does, it passes the character to the function following `getc` and replaces the original `getc` with the result of this application. In the implementation, we can see exactly this happening.

Now that we have `ungetc`, we can implement `skip-whitespace`. First, we'll make a utility function to check if a character is a whitespace:

```
func whitespace? : Char -> Bool =
    \c
    any (c ==) [' ', '\t', '\n', '\r']
```

The `any` function takes a predicate and a list and checks if any element in the list satisfies the predicate. Now, back to `skip-whitespace`:

```
func skip-whitespace : IO -> IO =
    \next
    getc \c
    if (whitespace? c) (
        skip-whitespace;
        next
    );
    ungetc c;
    next
```

It gets a character, check if it's a whitespace, if it is it recursively continues, otherwise it puts it back on the input (using `ungetc`) and follows with `next`.

Finally, we can implement the `scan` function, which uses both `skip-whitespace`, and `ungetc` (to put the final whitespace character, marking the end of the word, back):

```
func scan : (String -> IO) -> IO =
    \f
    skip-whitespace;
    "" |> fix \loop \s
        getc \c
        if (whitespace? c) (
            ungetc c;
            f (reverse s)
        );
        loop (c :: s)
```

The `scan` function is a trailing lambda kind vertical function by its type. It skips the whitespace, and then it enters a loop to accumulate the word. Each iteraction reads a character, and checks if it's a whitespace. In the case it's not a whitespace, we just continue accumulating. If it is, we've reached the end of the word and so we need to first `ungetc` the "unintentionally" read whitespace character, then pass the accumulated word to the follow-up function.

Here's a simple calculator program that uses `scan`:

```
func main : IO =
    print "> ";
    scan \x-str
    scan \op
    scan \y-str
    let (extract (float x-str)) \x
    let (extract (float y-str)) \y
    println (
        if (op == "+") (string (x + y));
        if (op == "-") (string (x - y));
        if (op == "*") (string (x * y));
        if (op == "/") (string (x / y));
        "invalid operation: " ++ op
    );
    main
```

The built-in `float` function takes a string and returns `Maybe Float`, to properly account for cases when the string doesn't contain a correct floating-point number. For simplicity, we don't handle this case, and simply extract the value out. If the number was, indeed, invalid, our program just crashes.

Here's a sample run:

```
> 2 + 3
5
> 44 * 44
1936
> 10 / 3
3.3333333333333335
> ^D
```

3.2.3 Chaining

Sometimes it's desirable to chain two or more `IO` objects to be "executed" one after another. This is especially useful when we want to apply some transformations to the first `IO` object that we don't want to apply to the other ones.

The operation of chaining `IO`s is in character similar to the list concatenation, by no accident. Hence, we'll give it the same name, we'll overload the `++` function for chaining `IO`s.

The actual implementation of `++` is surprisingly simple:

```
func (++) : IO -> IO -> IO =
    \io \jo
    switch io
    case done
        jo
    case putc \c \io
        putc c;
        io ++ jo
    case getc \f
        getc \c
        f c ++ jo
```

If the first `IO` finished, the second one continues. Otherwise we just propagate the chaining down the structure until we eventually finish the first `IO`.

The type of the chaining function conforms to the schema of semicolon kind vertical functions. However, in its infix form, `++` isn't usable as a vertical function. In the cases we actually do want to use it as a vertical function, we can easily overcome this problem

by using it in its prefix form, and thereby creating a "`(++)` block":

```
func main : IO =
    println "before the (++) block";
    (++) (
        println "inside the (++) block";
        done
    );
    println "after the (++) block";
    done
```

The above program prints:

```
before the (++) block
inside the (++) block
after the (++) block
```

As we've already mentioned, the most important property of chaining is that all transformation functions applied inside the block have no effect outside of it. The only real transformation function that we've seen so far is `ungetc` (and the ones that use it, all the other ones just prepend `putc`/`getc`), and here's how it's effect gets lost when reaching the end of a `(++)` block:

```
func main : IO =
    ungetc 'a';
    ungetc 'b';
    (++) (
        getc \c
        putc c;  # prints 'b'
        ungetc 'x';
        done
    );
    getc \c
    putc c;  # prints 'a', not 'x'
    putc '\n';
    done
```

This program prints:

```
ba
```

This property of canceling effect of transformations comes especially handy with other, more advanced transformations. We'll see some examples of those later in this chapter.

3.2.4 `when`, `for`, `when++`, `for++`

Now we'll shortly cover some control structures.

The `when` function works exactly as with lists:

```
func main : IO =
    print "Do you like oranges? ";
    scanln \answer
    when (answer == "yes")
        (println "That's awesome!");
    println "We're selling kilogram / 1 euro. How much? ";
    scanln \amount-str
    let (extract (int amount-str)) \amount
    when (amount < 5)
        (println "(You cheap.)");
    println "Thank you for your order.";
    done
```

Sample running:

```
Do you like oranges? no
We're selling kilogram / 1 euro. How much? 0
(You cheap.)
Thank you for your order.
```

Aside from `when`, we can make a general for-each looping construct, which made no sense with lists, but makes a lot of sense with `IO`. The function will take a list and basically "execute" a block of code per element. Here it is:

```
func for : List a -> (a -> b -> b) -> b -> b =
    \list \body \next
    if (empty? list)
        next;
    body (first list);
    for (rest list) body;
    next
```

The implementation should be self-descriptive.

Here's how we can use it:

```
func main : IO =
    for [1, 2, 3, 4, 5, 6, 7, 8, 9] (
        \x \next
        println (string x);
        next
    );
    println "finished";
    done
```

This program can be alternatively written with function composition:

```
func main : IO =
    for [1, 2, 3, 4, 5, 6, 7, 8, 9]
        (println . string);
    println "finished";
    done
```

Sometimes, we want to apply some transformations inside the bodies of `when` and `for` that we don't want to have an effect outside. We could put a `(++)` block inside the body, but we can also make ourselves alternative versions of `when` and `for` that do this for us automatically. We'll call them `when++` and `for++`. These functions are not needed as often, as `when` and `for`, but occasionally they come very handy. Their definitions are almost identical to the definitions of `when` and `for`, except that the simple application of the body to the `next` argument is replaced by chaining. Here they are:

```
func when++ : Bool -> IO -> IO -> IO =
    \cond \then \next
    if cond (then ++ next) next

func for++ : List a -> (a -> IO) -> IO -> IO =
    \list \body \next
    if (empty? list)
        next;
    body (first list) ++ for++ (rest list) body next
```

We're not going to show any example of their usage as they're easily conceivable

Instead, we'll move on to a way more interesting topic: grand transformations.

3.2.5 Grand transformations

Being just an ordinary, transparent data structure, `IO` is fitted for arbitrary, computable transformations. So far, we've only witnessed a glimpse of what this really means. Mere two examples, `ungetc` and `++`, are not enough to convey the full extent of the consequences. Arbitrary, computable transformations means exactly what it says. Practically any conceivable transformation is possible.

In this section, we'll take a look at a few examples, that incrementally show more and more powerful transformations. Of course, we'll still stick to contrived examples, but they should carry the ideas nonetheless.

As the first example, say the program responding to the user is using a typewriter with a disfunctional 'h' key. The objective of our transformation will be to replace all letters 'h' in the output with spaces. Then we'll apply this tranformation to the "cat" program and see what happens.

The transformation is very simple:

```
func no-letter-h : IO -> IO =
    \io
    switch io
    case done
        done
    case putc \c \io
        if (c == 'h')
            (putc ' ')
            (putc c);  # this is the else branch
        no-letter-h;
        io
    case getc \f
        getc \c
        no-letter-h;
        f c
```

The code is easy, what's important is to propagate the `no-letter-h` applications down the `IO` data structure.

Now we'll apply the transformation to the "cat" program. This could be our first

attempt:

```
func main : IO =
    no-letter-h;
    getc \c
    putc c;
    main
```

However, there's a subtle problem. The `no-letter-h` transformation gets applied in every recursive application, eventually creating an ever-growing stack of transformations that progressively slow down the program with each read/written character. This is better:

```
func main : IO =
    no-letter-h;
    fix \loop
        getc \c
        putc c;
        loop
```

Running this program, we'll get something like this?

```
hi!
 i!
how are you?
 ow are you?
what the hell?
w at t e  ell?
```

Seems to work fine.

Our next example will reverse the lines in the output. What does that mean? There are two situations when a line is finishes and needs to be shown to the user: 1. on the newline character, 2. when requesting user input, i.e. on `getc`. So, our transformation will accumulate characters in the output until one of the two abovementioned situations arises. When that happens, it prints out the accumulated string, but reversed. Thus, all lines in the output will appear reversed to the user.

Here's how we'll do it:

```
func reverse-lines : IO -> IO =
    \io
    io |> "" |> fix \loop \s \(io : IO)
        switch io
        case done
            done
        case putc \c \jo
            if (c == '\n') (
                println s;
                loop "";
                jo
            );
            loop (c :: s);
            jo
        case getc \f
            print s;
            getc \c
            loop "";
            f c
```

At the beginning, we enter a loop, that remembers two things: the line accumulator `s`, and the current `IO`. Then it examines the latter. When printing, two cases may occur. Either we've reached the newline, in that case we print the accumulated string reversed (it's already reversed, because we're accumulating it by prepending), or it's not a newline, in which case we simply accumulate a new character. When reading a character, we always need to print the accumulated string, then proceed to actually read the character, and then continue the loop with an empty accumulator.

Let's see how this works. We could use "cat" again, but that doesn't demonstrate the case, when we need to print before getting user input. Here's out example:

```
func main : IO =
    reverse-lines;
    print " What's your name? ";
    scan \name
    println ("Hello, " ++ name ++ "!");
    done
```

And here's its sample run:

```
 ?eman ruoy s'tahW Michal
!lahciM ,olleH
```

Notice how we've put an extra space at the beginning of the name question. This is so that we get a space at the end of the prompt when reversed.

And now for the last example. This one is a funny one. It scans the output, looking for numbers. When it finds a space-delimited integer number, it attaches a little note to it telling us what the factorial of that number is. Specifically, say the output of a program is this single line:

```
I have 2 hands with 5 fingers.
```

Our transformation makes the output to be this instead:

```
I have 2 (2! = 2) hands with 5 (5! = 120) fingers.
```

We'll approach this problem as follows. We'll use a loop with a string accumulator. This accumulator will be used for storing space-delimited words. Every time a program prints a character, we either accumulate it, or we reset the accumulator in the case the character is a whitespace. When encountering a whitespace, we'll also do one other thing. At that point, the accumulator contains the whole space-delimited word. We'll check if this word is a valid number. This is easy, simply convert the string to `Maybe Int` using the built-in `int` function and check if the result is not `nothing`. If it is a valid number, we'll print the note.

It's all fairly easy. Assuming we have an implementation of the factorial function called `n!`, here's how it's done:

```
func tell-factorial : IO -> IO =
    \io
    io |> "" |> fix \loop \word \(io : IO)
        switch io
        case done
            done
        case putc \c \jo
            when (whitespace? c && word != "") (
                \next
                let (int (reverse word)) \(maybe-number : Maybe Int)
                switch maybe-number
                case nothing
                    next
                case just \x
                    print (
                        " ("          ++
                        string x      ++
                        "! = "        ++
                        string (n! x) ++
                        ")"
                    );
                    next
            );
            putc c;
            loop (if (whitespace? c) "" (c :: word)) jo
        case getc \f
            getc \c
            loop word (f c)
```

You can see some type annotations in the code. This is because the type checker is sometimes not smart enough to know that a variable is a union before entering the switch structure. This will eventually get fixed.

Let's see if it works. We'll use it to transform the "cat" program, once again:

```
func main : IO =
    tell-factorial;
    fix \loop
        getc \c
        putc c;
        loop
```

And we run it:

```
ready 2 go?
ready 2 (2! = 2) go?
I am 22 years old.
I am 22 (22! = 1124000727777607680000) years old.
I'll sell you a T-shirt for 13 dollars.
I'll sell you a T-shirt for 13 (13! = 6227020800) dollars.
1 2 3 4 5
1 (1! = 1) 2 (2! = 2) 3 (3! = 6) 4 (4! = 24) 5 (5! = 120)
```

This example is, in itself, useless. However, it beautifully demonstrates the kind of power grand transformations give to us. This power brings many benefits, the most important one of which is: separation of concerns. In most languages, many program features have to be, more or less, tied together. In Funky, separating them into multiple independent functions is possible to an unprecedented degree.

What's even more important, this separation is possible without the need of any "abstraction framework". In most, especially imperative, languages, creating an abstraction infrastructure (abstract classes, interfaces, ...) is a necessary precondition for separating concerns. In Funky, separation comes naturally, without the need for an infrastructure. This makes it possible for code to grow naturally and without much bureaucracy.

3.3 Complete minimal transparent representation

In this last section, we want to describe an idea that we believe is important in making all of the things above possible.

Let us introduce it by a metaphor. Ink and paper. These two substances are all that's needed to make a book (exluding the cover). Ink may be put on the paper by the means of a pen, a typewriter, or a printer. However, neither the pen, the typewriter, nor the printer are a part of the book.

What we're talking about here is the principle of separating data and behavior. The book is the data. The pen, the typewriter, and the printer are behaviour. And here's my point: *once the form of the data is set and good, arbitrarily abstract behavior over the data can be added any time*. There's no need to attach behaviour to data.

However, this statement really only holds in a pure functional, and, at least optionally, lazy language. Why is that? Often, we want to model our program partially concurrently: we want some part of the computation to pause termporarily, until we request it to continue again. A good example of this are iterators. Iterators are basically loops, that pause and run on demand, giving us values. But, in order to implement iterators in a strict and imperative language, one needs to remember the state of the iterator in some variable, then mutate this variable any time the iterator progresses. With laziness, this is never needed. Behavior is instead embedded in the thunks of lazy evaluation. Data structures – instead of being just data – remember paused computations that are run on demand. This ultimate abstraction makes it possible to completely avoid storing behavioral state in data and thus makes the above statement true.

But given some requirements of data, how do we find a form that's good? I propose this form should satisfy these three characteristics:

1. Completeness – all possible instances of the data should be representable.
2. Minimality – there should be one and only one way of representing each logically distinguished instance of the data.
3. Transparency – given that all parts of the form are necessary and none are redundant, there's no need to encapsulate anything, the form should be fully transparent.

A data structure that satisfies all these conditions is called *complete minimal transparent representation*.

Satisfying the first and the third condition makes it possible to leverage arbitrarily abstract, high-level operations and transformations on the data – functions. Satisfying the second condition makes it convenient.

All the data structures that we've dealt with in this work satisfy the three conditions – `List` and `Maybe` are easy examples. Given their extremely simple definitions, just think about what kinds of stuff we were able to do with them. Furthermore, literally any computable transformation is possible on them. They don't need to adapt for any additional behaviour. They represent their data fully, minimally, and transparently. All we ever need to add is more functions.

The `IO` data structure we used to represent side-effects is another example. All possible programs that interact on the standard input and ouput are representable by this data structure. Each one is representable in one unique way (in terms of `putc`/`getc`/`done`, not in terms of high-level functions). The structure is fully transparent, there's nothing to hide. All the behavioral information is perfectly hidden in the lazy thunks/closures/to-be-evaluated function applications.

That is why we've been able to build functions that elevate the way we work with `IO` to such a high-level of abstraction. And we've hardly came the whole way.

Now, consider that these complete, minimal, transparent representations exist for all kinds of things. Web servers, video games, GUI applications, the list goes on. Just imagine what kinds of things we'd be able to do to them. Take the games example. We have a game done, but now we'd like to add an intro sequence to each level. In an imperative languages, this would possible have an influence on the base structure of the game. With our representation, we could build the intro separately, then just combine it with the rest of the game using an appropriate combinator function. Similarly, adding or removing a physics feature could be a matter of applying or not applying a single function.

This, in our view, is a genuinely good vision of programming. We hope we'll get there someday.

Conclusion

In this work we described the Funky programming language in its entirety, as well as outlined many use cases of its major innovative features, most notably the idea of vertical code and side-effect interpreters.

The initial step in the design of the language is seamingly backwards: omition of many features traditionally included in purely functional languages, such as pattern matching, or guards. But, we argue that this step backwards made it possible to see the bigger picture, and to eventually design a language that's better.

Funky has the consistency and seamless extensibility of LISP: built-in functions look and feel just like the ones created by the programmer, and the ones created by the programmer feel just like the built-in ones. However, Funky also features the readability and familiarity of more traditional languages. Features like the semicolon and trailing lambdas prevent the myriad of parentheses found in LISP, while infix functions make mathematical expressions – and others where it makes sense – look familiar and immediately understood.

The type system allows expressing most kinds of interesting types. We believe that it supports expressing all kinds of types of actual interest in practical programming, but this is arguable. Most notably, Funky's type system lacks higher-kinded types. In our view, this comes with a benefit: the madness of category theory is categorically prevented. While beautiful and useful in theoretical sciences, no one missed the category theory in actual practical programming until Haskell started claiming otherwise. As we've seen, all the things achieved by monads in Haskell are easily achievable in Funky, albeit not so generally. This generality, however, more often than not clutters intentions and meaning instead of clarifying them.

There still are a few features that are planned, but haven't made it yet into the language. The main one is package management and imports. The other one is a not-yet-mentioned `with` keyword that will allow defining functions based on the existence of other functions. For example: if an `==` function exists for a type `a`, define the `!=` function based on it. This feature is the one missing feature in the expressiveness of Funky's type system – after its addition, the expressiveness will match, and in many cases exceed, the expressiveness of Haskell's type system.

What's the future of Funky? At the point of writing this thesis, the language is virtually unkown. But, the language was not invented for the purposes of this thesis, it was vice-versa. We will make all the efforts to get the word out there, because we believe we've got something worthy in our hands.

Bibliography

[1] Luis Damas and Robin Milner. Principal type-schemes for functional programs. *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '82*, (October):207–212, 1982.

[2] Martin Grabm. Algorithm W Step by Step. *Stat*, 2006:1–7, 2006.

[3] Idris - Modules and Namespaces - Explicit Namespaces, http://docs.idris-lang.org/en/latest/tutorial/modules.html#explicit-namespaces.

[4] Achim Jung. A short introduction to the Lambda Calculus. , :1–10, 2004.

[5] Haskell/Lenses and functional references, https://en.wikibooks.org/wiki/Haskell/Lenses_and_functional_references.

[6] Records in Haskell, https://ghc.haskell.org/trac/ghc/wiki/Records.

[7] Guy L. Steele. Growing a language. 1999.

[8] Edsko De Vries, Rinus Plasmeijer and David M Abrahamson. Equality-Based Uniqueness Typing. , 0:1–16.

[9] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89*, (October):60–76, 1989.

[10] Philip Wadler. Monads for functional programming. , (May 1995):233–264, 1993.

Appendix A

The attached CD contains the source code of the compiler and the runtime, the small standard library for the Funky programming language, and a short usage manual. The same can also be found on the GitHub page of the project: https://github.com/faiface/funky