6CCS3PRJ Final Year
A compiler for a purely functional programming language on the JVM

Final Project Report

Author: Elian Ouaknine

Supervisor: Dr Laurence Tratt

Student ID: 1745855

July 19, 2021

Abstract

Since the creation of the Scala programming language in 2004, followed later by modern languages, the functional programming paradigm has flourished again. It finds is way in mainstream open-source projects, no longer reserved to skilled developers. This project aims to create a purely functional programming language called Fula that targets the Java Virtual Machine. More specifically, the language will allow the user to write programs around pure functions composition without side-effects. Fula will also enable to perform I/O operations (without side effects) and to treat functions as objects. The goal of Fula is to provide a language that could help interested developers to discover the functional paradigm through its core characteristics. Enabling them to write well-known and easy algorithms in a purely functional way.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Elian Ouaknine

July 19, 2021

Word count: 10120

Acknowledgements

I would like to thank my supervisor, Dr Laurence Tratt, for offering his support and guidance throughout the creation of the project. The help is much appreciated.
I would also like to thank Dr Christian Urban who introduced me to functional programming and compiler design. Without doubt it led me to the idea of this programming language.

Contents

Chapter 1

Introduction

Nowadays especially in Object-Oriented Programming, programmers tends to write functions that modify some state. Consider the Java code below, it illustrates a function that process the next task from a queue. It takes as hidden input the queue, modify its state and output whatever the execute function performs. Without looking at the implementation, one can't get an idea of what it is doing exactly. Those side effects introduce bugs as each function can depend on anything and modify anything outside its declaration. As the codebase size increases it becomes harder to reproduce the state of a function call, therefore to write comprehensive tests. At some point if the programmer does not keeps track of *when* and *where* each function will be called, the codebase become unmanageable.

```
public void processNext() {
     Task task = TaskQueue.popTask();
     if (task != null) {
             execute(task);
     }
 }
```

Functional programming resides around the idea of pure functions, ones that do not allow side-effects. It finds its roots back to the early days of computing when Lambda calculus was created in the 1930s. This paradigm was then codified and introduced by John McCarthy with the language Lisp in the 1950s, followed later by Haskell. Functional programming gains popularity among academics but failed to enter business use and modern compilers for years.

However recently with the rise of multi-core processors interest in functional programming constructs rises. Modern multi-paradigm compilers such as Scala and Kotlin resides heavily on the functional paradigm. They allow the user to perform most of what functional programming languages do. However those languages are permissive as they do not define a clear frontier between the different paradigms. They are mostly built around the idea that functional programming should be used where it is the most appropriate. While useful in order to target a wider range of developers and acquire the business market, it makes it harder when the goal is to learn this specific paradigm. Especially as imperative programs are the most widely used, if the user can write impure functions he will eventually do so.

1.1 Project Summary

The primary aim of this project is to create a pure functional programming language called **Fula** that targets the Java Virtual Machine (JVM). The language is modelled after Scala. Fula supports most of the JVM primitive types (Int, Float, Boolean, String) as well as function types. It supports high-order and first-class functions. The language is statically typed through the compiler type-checker. Deliberately the language is not permissive and forces to use functional paradigm characteristics such as immutability and pure functions. Fula has been think as a way to learn functional programming by writing small well-known algorithms. It also serves as an experiment to discover how such a paradigm can be implemented on the Java Virtual Machine. Indeed the stack machine was not designed to support some of the functional constructs such as function object by default.

1.2 Report Structure

First the report will begin with a background section that will overview the topics relevant to the project. Subsequently, it will cover the requirements, specification and design of the language.

Next the major section that cover the implementation of the fula compiler. This chapter will first cover each phase of the compiler, especially the type-checker and the ByteCode generation. Secondly it will cover the implementation of the most interesting features of the language such as the IO type and the function objects.

Following this the language will be critically evaluated, considering its strength and weaknesses as a whole. Finally a conclusion section will summarize what has been done and overview possible future work.

Chapter 2

Background

In this chapter we discuss about the importance and idea behind the design of programming language. We will define functional programming in the context of the project. Subsequently we will describe the computer program that enables the design of programming language: the compiler.

2.1 Programming Languages

A programming language in its most singular form is a formal language comprised of a set of grammatical rules, instructions and syntax used in order to write programs. Those programs are used in order to control the behaviour of a computer system. As the computer is only able to understand a set of 0s and 1s, we need such languages that works as a middle-man between the human and the computer.

Like any language, a programming language is described following rules so that humans can understand each other when using it. More importantly the computer is able to make sense of it. Some known steps are usually used to define a formal language [2]:

1. Words are identified by tokens. Each word has a particular meaning, for instance a keyword **def**, in the context of the language. The tokens are described using regular expression, following a traversal tokens are attached to each words. We call this part lexical analysis.

2. The syntax of the language is defined by a grammar, rules that state how the tokens can be combine together. The meaning of a keyword can change regarding its place in a statement. Usually grammar are described by so-called context free-grammar because we

know how to parse them in linear time. A context free-grammar, as its name suggests, do not consider the context of the symbols. This part is called lexical analysis and produce an abstract syntax tree. This abstract syntax tree can then be analyzed in order to derive the context of each symbol.

3. Finally, we need to make sense of the statements defined by the grammar in order to understand what the writer wanted to tell. There are usually encoded in plain English on the user end, and their meaning is implemented in programs so that the computer can understand and execute those.

The number of programming languages is quite huge, more than 700 used in academics and industry, that it can be hard when it comes to make a choice for the design of a custom programming language. There exists sets of more or less strict rules that helps to describe a family of programming language that we called paradigm.

2.1.1 Functional Programming

Writing good software is hard, especially when it comes to huge codebase and infrastructure. Bugs, errors and incapacity of understanding the code increase with the size of the infrastructure. Functional programming follows the declarative paradigm, in which you **declare** what you want through the help of functions rather than describing the control flow of a state (imperative paradigm). In functional programming you only use *pure functions*, functions with no *side effects* [3]. For instance you cannot modify a variable, throw an exception, read from and to file, printing on the console. At first glance functional programming reduce heavily how we can write programming language, but it is for the sake of *modularity* [3]. Indeed, these programs will be easier to reuse, test and parallelize. It will be easier to recover from mistakes as the code is defined in smaller independents chunks.

Apart from pure functions, functional programming resides heavily on function composition. It is enable by *high-order* functions– function that takes functions as argument and returns functions. Moreover functions have their own type and are considered *first-class*– function can be assigned to values as any other type. Functional programming does not allow mutability, values are initialized but can never be modified or re-assigned. Therefore loops can not exist as in imperative languages, user should rely on recursive functions to mimic looping.

If a software only makes some computations and never returns the information back to the user it is meaningless. Whether it is writing to a file or printing to the screen so called *input/output* (IO) operations considered as side effects are crucial. The functional language

Haskell use a Monad type class called IO to encode the side effects as value of that type [5]. The IO Monad empowers the user with composition rules that makes side-effects easier to manipulate. Therefore instead of being performed, the side effects are described. The only IO action that will be run is the program in the one return by the main function. It enables the user to perform IO operations as in any imperative programming languages, treating side effects as any other values.

2.1.2 An abstract machine language: Java ByteCode

A computer *machine language* is a low-level language that is easily understand by the computer [2]. It is defined by its instruction set, and consists of a sequence of instructions and operations that acts directly on some of the hardware of the CPU like the registers. The Java Virtual Machine (JVM) is a *virtual machine language*. Virtual in the sense that it is not implemented directly on the hardware but as a software program that enables execution of programs compiled to Java ByteCode.

2.2 Compiler Design

A compiler is a program that takes programs written in an high-level language and transforms it into a semantically equivalent low-level language that can be executed directly in the computer or using a virtual machine already installed on the computer (the JVM). Compiler are usually break up into multiple sub-parts. The decomposition makes it easier to implement the compiler into smaller sub-programs that communicate through known interfaces.

2.2.1 Front-end

The front-end of a compiler is the part that analyses the input program (high-level language) and produce a tree representation of the program called the *abstract syntax tree.* It is language dependent as it reads an input program following a custom lexical syntax using given keywords and constructs. This part can be further decomposed into smaller parts that performs special actions.

1. First the lexer that attach tokens to each words of the program during the *lexical analysis.* It breaks a string (stream of characters) into a stream of tokens.

2. Secondly the *syntactical analysis* that takes the stream of tokens and parse it against a context-free grammar in order to produce an abstract syntax tree (AST). Most of the parser are now able to parse directly strings rather than stream of tokens.

3. The last phase of the compiler is called the *semantics analysis* performs modification and verification on the AST. It is mostly used to declare and check the validity of types. Operation rules are also dictate by the type-checker. During this phase name are resolved and any inconsistencies raise errors.

4. There can be other sub-phases that simplifies the AST or the type-checking phase. For instance *desugaring* translate ”sugar” constructs to identical constructs in the tree.

The frontend is responsible for translating the program to a tree and for *language dependent* transformations of the tree such as compile time type-checking and *desugaring*.

2.2.2 Back-end

The back-end of the compiler is responsible for generating low-level language from the AST that has been created, checked and optimized during the front-end phase. The generated program has the same semantically meaning of the high-level language. The generated code can be further optimized during a *peephole phase*[2]. The compiler will scan the generated set of instructions and remove or unnecessary load/store pairs. It is a common pattern especially in the JVM that is a stack based, when you load a value into memory and in the next instruction immediately load it again into the top of the stack. Moreover the back-end is responsible for any modification of the AST that is IR-dependent. For instance a modification of the tree that is necessary for being translated to JVM bytecode will be carried during this phase. If the language later targets another intermediate representation (IR) a new special-purpose back-end will then be created, avoiding the need to modify the current compiler front-end.

2.2.3 Type System

For statically-typed programming languages the type system is the core part of the compiler. It is used in programming languages to check the absence of undesirable or unsupported behavior– a **string** cannot be add to an **int**. The type systems consists of a set of rules called *typing judgments* that together restraint the behavior of a particular expression given its type. The type system use inference rules when the type is not explicitly given by the user. Otherwise the

$$\frac{\Gamma \vdash e_1 : \tau}{\Gamma \vdash \textbf{val}\ x = e_1; x : \tau}(\textit{e-val})$$

Figure 2.1: Typing judgement of the Fula compiler for the val expression.

system only has to check that the type of an expression match the type of the value to which it is assigned.

Typing judgements are written $\Gamma \vdash e_1 : \tau$. It can be read as *In the typing environment* Γ *the expression* e_1 *has type* τ . The typing environment is used to assign a type to a named value. The type systems is only concerned about typing rules, results of computation are never used.

Using inference rules, typing judgements can be composed in order to determine the type of an expression. Consider below the inference rule of the Fula programming language used to determine the type of a value after an assignment.

This rule can be read (from top to bottom of the type judgement) as if *in the typing environment* Γ *the expression* e_1 *has type* τ *it follows that the value x has also type* τ. Note that for practical reason *val* $x = e_1$ has type Unit in the Fula language, that's why the type of the expression evaluated here is *val* $x = e_1$*; x.*

2.2.4 Compiling to the JVM

A compiler that targets the JVM will generate a file with a **.class** extension [12]. This file is a ByteCode program that will be executed by the JVM. There exists libraries such as **ASM** or **Javassist** on the JVM that facilitate writing Java ByteCode. Those libraries are used by famous compilers such as Scala and Kotlin.

Java ByteCode can be considered as an Intermediate Representation (IR) of the program that relies on the JVM back-end to run. The JVM will in turn targets native code. The .class file, as it name suggests, is the definition of a class as it would be in Java. Each method define in the the .class file is represented as a set of instructions that are performed on a stack frame. The JVM has only a limited set of 32-bit primitive types such as *int* of *float*. All other types are classes and their name needs to be preceded by an **L**. At runtime the JVM will execute the main function of the class provided as argument. It will also load any class called by looking at the classpath. The *-cp* command can be used together with the *java* command to provide additional libraries to the classpath.

Chapter 3

Specification

3.1 Brief

The primary goal of this project is to create a compiler for a pure functional programming language. The output of the compiler is a **class** file that runs on the Java Virtual Machine. The compiler will be accompanied by a runtime library that extends the default language features.

3.2 Requirements

This project has two sets of requirements. First the requirements impose to the language that will be used by the user. Secondly the requirements to the compiler, used to translate a Fula program file to Java ByteCode.

3.2.1 Language Requirement Analysis

A MoSCoW analysis was performed (table 3.1) to guide the design and the implementation of the language. The language features are ranked by importance and difficulty of implementation.

MoSCoW Category	Language Features
Must-have	Pure functions, Function calls, Recursion, Expression (if-else, Arithmetic, Boolean) and Immutability
Should-have	Printing primitive types, Join printing actions, High-order functions and First-class functions
Could-have	Lambda expressions, Function lifting
Won't-have	Generic Type, Pattern Matching, List

Table 3.1: MoSCoW analysis of language features

Derived from the MoSCoW analysis the features of the Fula programming language are:

- function declaration, function call, recursive function (functions type must be annotated by the user).
- Syntactic sugar for declaring function that takes no argument.
- all functions are pure, there are no side effects.
- High-order functions.
- First-class functions, only for first-order functions.
- Variables are immutable, therefore called values.
- Assignment of an expression to a value.
- Type inference of values.
- Statically-typed and strongly-typed language.
- Boolean, Arithmetic and If-Else expressions.
- Lambda expressions, only for first-order functions.
- Function lifting, a function declaration is converted to a value, only for first-order functions.
- Printing primitive types. Describe as a value representation instead of a side effect using a special IO type.
- Combine IO type values together using the >> operator, borrowed to the Haskell language.

A Fula program is a sequence of function declarations followed by a main function as in the Haskell language. The syntax of the function declaration is borrowed to the Scala language. Thus, a function has no return statement, the expression returned is the last one of the function body. The main function takes no parameter and return an IO[Unit] value. At runtime the Java Virtual Machine will execute the side-effect described by the IO[Unit] value return by the main function. A description of the language syntax can be found in the Appendix.

3.2.2 Compiler requirements

The compiler is the core part of the language. It can be split in two phases: the front-end that consists of creating an abstract syntax tree (AST). The AST is a tree of nodes, each node representing an element of the program: expression, declaration... The tree will then be simplified and the type of each node will be resolved. The second phase is the back-end of the compiler. It is responsible for translating the typed AST into Java ByteCode. Prior modifications of the tree are needed to allow bytecode generation.

Front-End Requirements

Here are described the most important requirements of the front-end phase of the Fula compiler. Details will be given in the following chapters. The front-end is responsible for the language-dependent transformations of the Tree.

- The compiler should take a file with a **.fula** extension as argument.
- The compiler should parse a file and produce an Abstract Syntax Tree that represents it.
- The compiler should modify any syntactic sugar (desugar) to its equivalent representation.
- The compiler should discard any expression that will never be used: expression that are not assigned to a value and that are not the last one of a block.
- The compiler should do a semantic analysis of the program: name, scope and type resolution.
- The compiler should convert the AST into a typed AST.

Back-End requirements

Here are described the most important requirements of the back-end phase of the Fula compiler. Details will be given in the following chapters. The back-end is responsible for translating the tree to an Intermediate Representation. Modifications are first applied to the AST in order to ease the translation.

- The back-end phase takes as input a typed AST.
- The compiler should modify the name of the values (mangling) so that they will be easily converted to their bytecode representation (register number).

- Function declaration that represents function objects should be adapted to accept object wrapper classes.
- The compiler should transform the typed AST into a class file representation of it.
- The output of the compiler should be a class file, that can be executed on the JVM.

3.2.3 System Requirements

The Fula language will be shipped as two jar files: the compiler and the runtime library. It strictly requires the Java(TM) SE Runtime Environment version 16 to be run. Details on how the compiler can be run are described in the user guide.

Most programming language are open source. They are not only means to write programs but also form communities around the paradigms and design choices. Hence, the compiler should be maintainable and open to modification, as one of the goal is to open source it through a flexible licence on GitHub. This could be a chance to give a second life to the created language after the project and give anyone a chance to extend it so that it fits one's needs.

Chapter 4

Design

This chapter will cover the theoretical part of the language: Section 4.1 details the data structures used by the compiler and 4.2 gives details about the design of the IO type class and the JVM class file.

4.1 Data Structures

The Fula compiler contains two major data structures: the Abstract Syntax Tree (AST) and a Symbol Table. Each phase of the Fula compiler is built around the traversal of the AST. The Symbol Table enables the semantic analysis: names and types are resolved thanks to table lookups.

4.1.1 The Abstract Syntax Tree (AST)

The AST that represents the fula program is a *recursive Algebraic Data Type* (ADT). An ADT is a composite type [4] that is formed by combining other types. The types are usually a combination of two type classes.

- The *sum* type, an alternation between two types. For instance: type Expr = If | Aop, means that the type Expr is either an *if* expression or an *arithmetic expression.*
- The other one is the *product* type, a combination of two types. For instance: type If = Bexp $\star$ Block $\star$ Block, means that the type If is a combination of a boolean expression type and two blocks type.

The Fula AST is composed of two major types: Expression and Declaration.

- Each expression supported by the language is represented as a subtype of the expression type. For instance the If expression is a product between a boolean expression and two blocks. A block is a type that represents a list of expression. The AST is considered to be a recursive algebraic data type as an expression can be the product of other expression types.

- A function definition is a type of declaration. It contains the name, the function type annotated by the user and the body (a list of expressions). Thus the top part of the AST that represents the fula program is a list of function declarations.

The parsed AST does not resolve the type of each expression and declaration. It simply reads the input given by the user in the program and build a node representing it in the AST. The type-checker phase of the compiler resolves the names and the types of each expression. Moreover it attaches the type of each expression to its node representation in the AST. The AST returned by the type-checker is called a *typed AST*.

The types in Fula are also represented as a *recursive Algebraic Data Type* also. There are five primitives single types: Integers, Float, Boolean, String, IO[Unit] and a primitive function type. It is a product type of primitives types (either single or function type): the type of each argument and the type of the return value. There is a limitation on the type of first-class functions. They consist of single primitive types only. They are written as **Int,String** $\Rightarrow$ **Float** for instance. The type of values are inferred from the expression assigned to it by the type-checker. However the type of function declarations and lambda expressions has to be annotated by the user in the program. In addition to those types there are internal types used only by the compiler. As in Scala assigning the result of an expression to a value is an expression of type Unit. It is not considered as a side effect as does not perform any action but still has a type because it is an expression. The Unit type alone can't be used in the Fula language, it has to be wrapped around an IO.

Usually compilers resides heavily on the Visitor Pattern. In those compilers the abstract syntax tree is an object structure. Instead of creating subclasses a visitor implements an operation to be performed on that object. The client that traverses the tree will allow the visitor to perform that operation on the given object. Thanks to algebraic data types we can use pattern matching to mimic the Visitor Pattern in a functional way. Pattern matching allows to traverse the AST in a type-safe way. For each traversal custom operations on each type are performed. Especially when the aim is not to modify but to create an output based on the node pattern matching is the most suited solution. The visit of each tree node will perform an operation that creates a new node. This node can be of the same type or different type of the node visited. Moreover its position can change on the tree, all of that rules are dependent on the purpose of the phase. It allows to avoid state for each phase. Each compiler phase will pass a reference to the top node of the newly constructed AST.

4.1.2 The Symbol Table

In order to perform the semantic analysis of the program the compiler needs an environment. This environment is a Symbol Table that keeps track of the values and the functions declared: the Symbols. It is designed as a parent pointer tree (a cactus stack) [9]. Each node keeps only a reference of its parent. Each node is a table that maps the name of a symbol to a symbol node. This node is nothing else than the name, the package where it has been declared and the type of the symbol. For each new entered scope a new table node is created, pointing to the parent scope table. Functions and values are represented in the same way because function overloading is not authorized in the Fula language at the program level. Thus functions can be lookup only by their name.

A Fula program is a sequence of function declarations. Each function can call any other function declared in the file except the main function. The root node of the Symbol Table maps the function declaration names to their symbols. Each time the type checker enters the body of a function a new table node is created representing the current scope. When the type-checker exit that scope, the current table node is discarded and a reference to the parent scope is returned.

Looking up a symbol in the Symbol Table always starts from the current scope, recursively scanning outer scopes. In a given scope each value has a unique name. However Fula allows value shadowing. While a value is declared with a unique name in a given scope, this name can be already used either for a value or a function in an outer scope. Thus when adding a symbol in the table only the current scope is checked, improving performance. Moreover during semantic analysis, the name resolved is the closest one from the given scope. As an illustration consider the fula code below (Listing 4.1). The type of the *tmp* value is Float and its value is *11.02*. It uses the value declared in the same scope: the closest one.

```
...
val position = 10;
```

```
    if (position == 10) {
      val position = 11.02;
      val tmp = position;
    }
    ...
```

Listing 4.1: Value shadowing illustrated

As said earlier at the program level function are resolved by their name only as function overloading is not allowed in the language. However the *print and println* built-in functions are overloaded. They can take Integer, Float, Boolean or String as arguments. In order to enable that behavior the Symbol Table contains an additional table at the root level called the package node. Whenever a lookup in the program scopes fails to find a match, the Symbol Table searches the package node. At this scope the table maps symbol signatures to symbol nodes. A symbol signature is a type product of a name (String) and a list of types (the types of the function arguments). Instead of using only the name of the function, the type of the arguments are used in addition.

4.2 Design details

4.2.1 The IO type

The Fula language is purely functional. All functions are *pure*: they have no side effects. In the Fula language it means that a function always returns a value and does nothing else. Therefore the behavior of a function can always be derived by looking at its signature.

In order to allow the user to print values to the console, considered as a *side effect*, the Fula language uses a special type called IO[Unit] [5]. The idea is borrowed to the IO Monad of the Haskell language. The Monad is a type class that wraps the side effect in a value. In other words a value that has a type **IO a** is considered as an *effectful* computation. If that value is eventually executed at runtime, it would produce a value of type **a** and perform an I/O computation. But the value in itself is completely inert, it only describes the effectful operation not the execution.

How a value of type IO is ever executed then? In the Haskell language the compiler looks for a particular value:

```
main :: IO ()
```

It is the IO value returned by the main function. This value will then be the only one executed at runtime. The **()** type is the same as the **Unit** type in Fula. It has only one value and represents a function that returns nothing, the well-known **Void**. In Fula the procedure is exactly the same. The compiler looks for the **IO[Unit]** value returned by the main function. It is also the only value that will be executed at runtime:

```
def main(): IO[Unit] = { ... };
```

In Fula the IO type is not a Monad. It is not provided with the *return and bind* functions, and does not follow the three monad laws. It only supports effectful computation of type Unit. As the IO type class in Fula is not provided with a *return* function, the only way to obtain an **IO[Unit]** value is to call either the *print or println* functions. Those functions are the Fula implementation of the Haskell *putStrLn* function. It takes a primitive type and returns an effectful computation that produces nothing: a value of type **IO[Unit]**. However an operator for combining IO[Unit] values is provided in Fula. It is the equivalent of the sequence operator in Haskell written >>. The sequence operator in Haskell,

```
(>>) :: IO () -> IO () -> IO ()
```

simply creates an IO computation that consists of running the two IO values in sequence. The result of the computation is discarded. In the Fula language it works perfectly as the only type of IO computations represented returns nothing. For example in Fula the sequence operator can be used as follows:

```
val greeting: IO[Unit] = println("Hello") >> println("World!");
```

4.2.2 The class file

The fula compiler translates a fula program to its java bytecode equivalent. It is the Intermediate Representation that runs on the Java Virtual Machine. It is represented as a class file and is the equivalent in java bytecode of a class definition in Java.

A Fula program is translated to a single class file. For example a program file named **mand.fula** will be translated into a class called **fula.mand**. The class has no constructor as there is no state maintained in the program. Each function declaration is translated to a **public static** class method. The signature of the main method is translated to the java one. At compile time an instruction is added at the end of the main method in order to run the effectful computation.

Each lambda expression is translated to two **public static** class methods with custom names. The first one is the actual implementation of the method with the primitive types of the JVM. The second method is called an adapted method. It enables to use the primitive types as objects. Indeed *int, float and boolean* types have corresponding classes on the java language. The *String and the IO[Unit]* in Fula are already objects they won't need to be box and unbox to wrapper classes. The adapted method interfaces the first implementation method with object wrapper classes. The primitive types are unboxed and the regular implementation method is called. If the return value is a primitive type, it is boxed to its object wrapper class representation and returned. We will see later in Chapter 5 that it is needed to represent function as objects.

Chapter 5

Implementation

5.1 The Choice of Language

The Fula compiler has been written using the Scala language. It is mostly written in a pure functional way. Functional programming languages are usually well-suited for writing compilers. They offer a lightweight way to represent the Abstract Syntax Tree. Moreover they provide a handy method through pattern matching to traverse the AST in a type safe way. There is no state and no side-effects, all the functions are pure. The last operation of the main method execute the combined effectful computations of the compiler. The exception handling is done in a functional way using the *Either* type. The source code can be considered as a collection of functions, algebraic data types and data structures. The idea is that a next version of the compiler could be written itself in the Fula language, as Scala did with its version 2. The ByteCode generation package of the Compiler is not written in a functional way. It has been built on top of the ASM Java Library and resides heavily on modifying a *MethodWriter* object that maintains a State while the AST is traversed.

The Fula runtime library depends on the Scala library version 2.13.2 for the function objects representation and on a single Java file for the IO type and operations.

5.2 The Compiler Pipeline

The Fula compiler follows the typical design of statically typed languages. It takes a program, written in a file with a **.fula** extension type-checks it and compiles it to an Intermediate Representation (IR). The IR chosen for the Fula language is Java ByteCode that runs on the

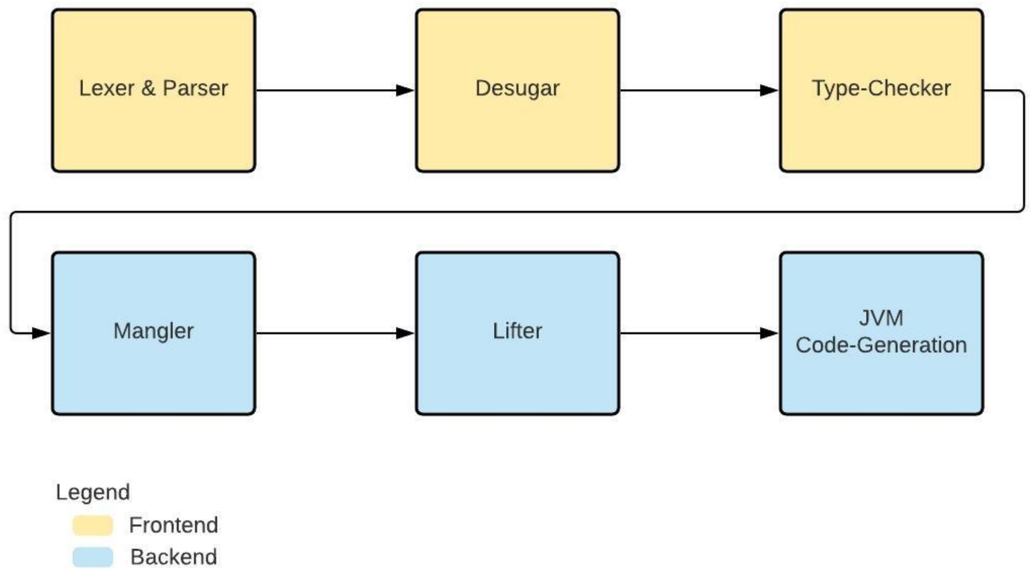

Figure 5.1: The compiler pipeline

Java Virtual Machine.

There are six phases in the Fula compiler (figure 4.1) that are divided into two phases: the front-end and the back-end. The compiler can be considered as a pipeline. Each step except the first and the last one modifies the tree in accordance to its own rules and pass it to the next step.

5.2.1 The front-end phase

During the front-end phase the Fula program is first translated to an AST. It is then simplified (desugared), type-checked and translated to a typed AST.

Lexer and Paser

The lexer and parser were written using the Scala **fastparse** library [7] [8]. Fastparse is a parser combinator that uses recursive descent parsing. A parser combinator is an high-order function that accepts several parsers as input and returns a new parser as output. It is designed as a top-down parser that starts building the abstract syntax tree from the root node. Every time it finds a matching input it advances to the next input. If it fails to match the input it backtracks to reach the next production rule of the grammar. The grammar of the Fula language is therefore not left-recursive has top-down parsers do not support it.

Desugar

Desugaring is the process of simplifying the AST for the next phases, so that they have to deal will less cases. In Fula there are four desugaring rules.

Fula supports declaring functions that take no parameter and have a single expression as body with a special syntactic sugar (see code below). Those functions can be written in the root scope of the program as values assignment. Those functions will be translated to regular definition declaration nodes.

```
val x: Int = 10;
val y: String = if (x() == 10) { "x"; } else {"Not␣x";};
```

The functions provided by the IO runtime library (print, println and >> operator) are parsed in special nodes by the compiler. Those functions and operators are desugared into regular function calls: *Assign* nodes. Indeed, parsing them in special nodes allows the compiler to track their position and replace them with the right name and package.

The lambda expression are desugared into regular function declarations. Their definition is moved to the root scope of the program. Where the lambda expression was declared, the compiler replaces it by a regular *Value* node that references the lambda expression using its name. The function declaration translation of the lambda expression is given a custom name. It consists of "$anonfun$\$" followed by the name of the function where the lambda expression was declared, suffixed with its position in the function. For example if two functions are declared in the *test* function, the first will be given $anonfun\$test\$0$ name and the second one $anonfun\$test\$1$ name.

In Fula, function's body consists of a sequence of expressions. Expressions can be assigned to values and the last expression is used as the return value. If an expression is neither assigned to a value or the last expression of a block (whether the root block of the function or an if block) the compiler discards it. This expression will never be used by the compiler. It makes the Fula compile quite permissive in which programs can be written. As the desugar phase is before the type-checking one, type errors can be written and the compiler won't notify the user. Therefore the compiler can translate programs that contains type errors in one's free of errors at compile time. Notifying the user of such changes should be considered as a important feature to be added.

```
def func(): Int = {
        val x = 10;
        x+10.02; // This expression is discarded and has a type error.
        (x:Int):Int => x+2; // This expression is discarded.
        x;
};
val x: Int = 10;
val y: String = if (x() == 10) { "x"; }Âăelse {"Not␣x";};
```

Type Checker

The type checker is the core part of the compiler. Thanks to typing judgements it is able to statically type the programs. Moreover it is responsible to resolve the names, modify some AST nodes based on their type and attach the type to each expression.

The type checker uses the Symbol Table to perform the semantic analysis (name and type resolving). It adds at the root node scope level each function symbol. The default functions of the Fula language are added to the package node of the Symbol Table. Therefore calls to default function will also be type checked. The type checker then traverses each function declaration. For each function a new scope is created, that will be discarded when the function scope is exited. A Symbol is added in the function body scope for each argument. Each time a value is traversed, its associated expression is first typed and a Symbol is added to the Symbol Table. The Symbol Table checks that no Symbol has been already declared with the same name only in the current scope. When the type checker traverses a reference to a value it search the Symbol Table from the current scope to the root scope. If it finds a match, it checks the type equality if the user annotated it, and attach the inferred type to the node (See Figure 5.2).

The type checker can infer the type of expressions created in function declarations (see type judgement Figure 2.1 and Figure 5.2). However, the user still has to annotate the type for function declaration (arguments' and return types) and lambda expressions. The expressions have either one of the five single primitive types or a function type. Function type can only be used for first-order functions. Only function declarations can be high-order functions. The type of functions are resolved by checking the equality between the return type and the last expression of the function body.

The compiler uses an internal special purpose type to represent function type for values. It simply consists of a wrapper type around the function type of a function declaration. The type

```
val x: Int = 10.02; // Type error        val x = 10.02; // No type error
```

Figure 5.2: Type equality is checked if the user annotates the value assignment. Type is inferred if no type is annotated.

```
def func(): Int = {
        ...
};
def main(): IO[Unit] = {
        val x = func(); // Keep as an
            ↪ Assign node in the AST
        ...
}
```

```
def main(): IO[Unit] = {
        val func = ():Int => x+2; //
            ↪ Type is wrapper
        val x = func(10); // Modified
            ↪ to an Apply node in the
            ↪ AST
        ...
}
```

Figure 5.3: The type checker modifies the Assign node to an Apply node in the AST if the function called is an expression.

checker treats as equal a function wrapper type and a function type. As the type checking phase is the only one to keep track of the symbols type it is the most suited one to modify nodes based on their type. In the Fula language calling a declared function and a value function is done the same way. However on the JVM it is not the case. Function values will be represented as object whereas function declaration can be called directly. Whenever the type checker traverses an *Assign* node, if the type of the function is a wrapper around a function type, meaning it is a function value and not a declaration, the node will be transformed to an *Apply* node in the AST. Those two distinct nodes will facilitate the bytecode generation (Figure 5.3).

The same method is applied when a function declaration is lifted to an expression and a lambda expression is created. During the desugar phase the lambda creation has been replace by a simple value reference to the function declaration that implements it. A referenced value and a lifted expression are parsed the same way by the compiler, in a *Value* node. In order to determine where a function is lifted to an expression and a lambda expression created) the type-checker resolve the type of the value referenced. If the value's type is a regular function type, that means it is a function declaration or a lambda expression translated to a function declaration, the *Value* node is translated to a *Function* node. See an example in Figure 5.4.

The type of each expression is resolved while traversing the tree using the regular type judgements. For an arithmetic expression and a boolean expression the type of each expression has to be the same. For an if expression the type of each block has to be the same also. In figure 5.5 are illustrated the type judgements in Fula for the subtract arithmetic expression and the if expression.

```
def func(x:Int): Int = {
        ...
};
def main(): IO[Unit] = {
        val x: Int => Int = func; //
            ↪ Translated to a Function
            ↪  node.
        ...
};

def main(): IO[Unit] = {
        val func = ():Int => x+2; //
            ↪ Type is a function as
            ↪ replaced with the
            ↪ function declaration
            ↪ name that implements the
            ↪  Lambda during desugar.
        val x = func; // Translated to
            ↪ a Function node.
        ...
}
```

Figure 5.4: The type checker modifies the Value node to a Function node in the AST when a function declaration is lifted to an expression.

$$\frac{\Gamma \vdash e_1 : \mathit{Float} \qquad \Gamma \vdash e_1 : \mathit{Float}}{\Gamma \vdash e_1 - e_2 : \mathit{Float}}(\mathit{e\text{-}sub}) \qquad \frac{\Gamma \vdash e_1 : \mathit{Boolean} \qquad \Gamma \vdash b_1 : \tau \qquad \Gamma \vdash b_2 : \tau}{\Gamma \vdash \textbf{if } e_1 \ \{b_1\} \textbf{ else } \{b_2\} : \tau}(\mathit{e\text{-}if})$$

Figure 5.5: Typing judgements of the Fula compiler for the subtract and if expressions.

5.2.2 The back-end phase

During the back-end phase of the Fula compiler the tree is prepared and subsequently modified to java bytecode. Preparation of the tree involves mangling the name of the values and adapting the implementation of the function expressions.

Mangler

The typed AST is traversed and the compiler performs *name mangling* of the values declared in the functions body. This ensures that the values names are not shadowed anymore in the function body, each value has a unique name. It is necessary as the JVM is a stack machine, function scope can not be represented in the class file, therefore two *local* values in the same function can not have the same name. The name given to each value in a function consists of the number of the value starting at 0 prefixed with a "_" character. During bytecode generation the underscore will be dropped and the name converted to an integer value. In figure is an example of how the name of values in a function are mangled.

Lifter

The typed AST is traversed one more time before bytecode generation. During the traversal each *Function* node that represents either a lifted function or a lambda expression is collected into a set. The compiler discard duplicates, especially when the same function declaration is

```
def func(x:Int): Int = {
        val z = if (x==10) {
                val x = 30;
                x+2; // value here is
                    ↪ 32
        } else {
                val z = 30;
                z;
        };
        x+z;
};
```

```
def func(_0:Int): Int = {
        val _1 = if (_0==10) {
                val _2 = 30;
                _2+2; // value here is
                    ↪ 32
        } else {
                val _3 = 30;
                _3;
        };
        _0+_1;
};
```

Figure 5.6: Name mangling in the context of value shadowing.

lifted many times in the program. For each *Function* node collected a new function declaration is created. Remember that a *Function* node represents a function declaration as an expression value. Its name consists of the name of the function being lifted suffixed with "$*adapted*". The purpose of this new function declaration is to interface the lifted function with a signature that allows the use of Java Object Wrapper class (Integer class for instance) where a primitive type is needed. The function will unbox the wrapped objects to their primitive type representation on the JVM and then call the initial function implementation with primitive types. If the returned value is a primitive type in the JVM (Int, Float or Boolean) the value returned will be boxed to its object wrapper class representation. We will see that it is needed when calling first-class function.

ByteCode Generation

The Java ByteCode generation resides heavily on the ASM library [1]. It uses the event-based API based on the Visitor pattern that traverses the class file while creating it.

Each function body is translated to its stack-based representation. For most of the Fula language constructs it is quite straightforward, it is sufficient to proceed a pre-order traversal of the typed AST. The AST nodes are translated to their java bytecode equivalent form and are added sequentially to the MethodWriter visitor.

Translating if expressions to java bytecode requires the use of Labels (Figure 5.7). Because the JVM is stack-based there is no scope java bytecode. First the result of the binary expression is added onto the stack: 0 if false, 1 if true. The result in then compared with the value 0, using the **IFEQ** instruction. If the value is equal to 0, meaning the boolean expression is false, the execution jump to the given label. (line 15 of Figure 5.7). Otherwise the instructions continues to be executed until it finds a **GOTO** instruction. The GOTO instruction is used to jump to

```
if (x == 10) {
        ...
} else {
...
};
```

```
...
10: ISUB
11: IFNE 14
12: LDC 1
13: GOTO 15
14: LDC 0
15: IFEQ 21
...
20: GOTO 30
21: ...
...
30: // execution is handed back to the
    ↪  outer scope.
```

Figure 5.7: Translation of if expression in java bytecode.

the end of the if expression, after the execution of the *false* block.

Language Runtime

In order to run a class file compiled from Fula, a runtime library has to be used. This runtime library contains the *scala library* and the special purpose *IO library* that enables the representation of computation as values (IO type). Those two libraries have been zipped together in a jar file called *fular.jar* (for fula run). The jar library needs to be added to the classpath of the java runtime environment while running the class file using *java -cp fular.jar* .

5.3 First-class function in a JVM language

The Fula language supports first-class functions, therefore high-order functions. In the JVM there is no function type, first-class functions must be converted to function objects.

5.3.1 The invokedynamic instruction

In Java 7 the **invokedynamic** instruction was added to the JVM instruction set [10]. It started to be used by the Java compiler since Java 8 and the introduction of lambda expressions [6].

Because lambda expressions and lifted functions are not primitive types on the JVM, they must be represented as object references. When calling a function declaration in the java bytecode representation of a Fula program the **invokestatic** instruction is used. Indeed the implementation of the *call target* is known as compile-time, it can be referenced directly.

In Fula the invokedynamic instruction is used to dynamically link lambda expressions and lifted function objects to a local defined static method. Therefore the Fula can avoid creating a custom anonymous class for each function object. In order to represent the functions as objects on the JVM the Fula compiler uses the Scala library. Each function type has been created as a handcrafted class that takes a single method to be implemented called *apply*. A function that takes no argument is called *Function0*, one that takes a single argument is called *Function1* and so on until *Function22*. The implementation of the apply method does not exist, the invokedynamic indeed used to link the implementation method to the object at runtime when the function object is created. In java bytecode when executing an invokedynamic instruction the name of the method implemented and the type of the function object are automatically by the VM for the boostrap method:

```
0: invokedynamic #91, 0 // InvokeDynamic #0:apply:()Lscala/Function1;
```

First we need to introduce the MethodHandles API of the Java Language to understand the invokedynamic instruction. Method handles can be considered as the function pointers we find in the C language, they point to a target, and can be used to invoke a function. MethodHandles come with a strategy to lookup for the function pointed. In the Fula compiler the lookup used is always an invokestatic reference because every method implemented is a static class method. At compile time the signature of the method referenced needs to be known. But an invokedynamic instruction returns a CallSite and not directly a MethodHandle. A CallSite can be considered as placeholder for a MethodHandles reference. A CallSite instead of directly a method handle is used because the method pointed could change at runtime but the Fula compiler does not use this behavior. In Fula the method used to create CallSite objects is **altMetafactory** [11] of the Java language. This static method has been specially created to create function objects from lambda expressions and method reference in Java 8. It is used as the bootstrap method for creating the needed CallSite. The object returned by the bootstrap method is a CalleSite even if the type function provided is a Function1 object. At runtime the type function provided will be used, but because we are only interested in calling the linked apply method of the object a CallSite object can be used.

At runtime, the first time the invokedynamic instruction is executed a bootstrap method is called to bound the CallSite to the target method that has the same signature and contains the implementation of the lambda expression or lifted expression represented as a value. The method linked to the call site is the static adapted method created during the Lift phase of the compiler. That is because the Function object of Scala (Function0, Function1 ...) can not handle primitive types but only object types. This method allows to avoid creating an anonymous function every time a new lambda expression is created. The bootstrap method example below is an example from a class file compiled with Fula. This can be viewed using the javap command. The bootstrap method used is altMetaFactory. The implemented method is called *hundred.$anonfun$main$0$adapted* and uses a static method handle reference to be invoked.

```
BootstrapMethods:
  0: #88 REF_invokeStatic java/lang/invoke/LambdaMetafactory.altMetafactory:(
      ↪ Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/
      ↪ invoke/MethodType;[Ljava/lang/Object;)Ljava/lang/invoke/CallSite;
    Method arguments:
      #77 (Ljava/lang/Object;)Ljava/lang/Object;
      #80 REF_invokeStatic fula/hundred.$anonfun$main$0$adapted:(Ljava/lang/
          ↪ Object;)Lfula/helper$IO;
      #81 (Ljava/lang/Object;)Lfula/helper$IO;
      #37 1
```

5.3.2 Calling a first-class function

Calling a first-class function is represented by an *Apply* node in the AST. It is represented in java bytecode by calling the interface apply method that we have bound to the call site using the invokedynamic instruction. For instance calling the apply method created above is translated to java bytecode by:

```
28: invokestatic #41 // Method scala/runtime/BoxesRunTime.boxToInteger:(I)
    ↪ Ljava/lang/Integer;
31: invokeinterface #47, 2 // InterfaceMethod scala/Function1.apply:(Ljava/
    ↪ lang/Object;)Ljava/lang/Object;
36: checkcast #49 // class fula/helper$IO
```

Because the function object only works with objects types, the primitive types have to be boxed and unboxed. Recall that the Fula compiler uses the java primitive types, we never want to leave an object wrapper class on the stack (except for the String and the IO type). The arguments are boxed to their objects wrapper class and the return value is unboxed to its primitive type representation. For the String and IO type we can avoid boxing and unboxing but we have to cast the Object type to the actual given type used.

Those function objects can be treated like any other object in Fula. They can be assigned to values, passed as argument of functions and returned by function, enabling high-order functions in Fula.

5.4 IO Type on the JVM

The IO type on the JVM is used to represent an effectful computation as a value. The implementation of functions returning IO values has similarities with the implementation of first-class function. Instead of using invokedynamic they are written as anonymous classes and can therefore be called with invokestatic (there compile-time name is known).

5.4.1 The IO type

```
interface IO {
    void unsafeRunSync();
}
```

Listing 5.1: Implementation of the IO type in Java

The IO type is represented by an interface named IO written in Java. It takes a single interface method in which the effectful computation is implemented. When executed at runtime the interface method will run the side-effects. The Fula user has no access to this method, it is reserved for the compiler to execute the IO value at the end of the main function.

```
108: invokeinterface #124, 1 // InterfaceMethod fula/helper$IO.unsafeRunSync
    ↪ :()V
```

5.4.2 The IO functions

The Fula language does not provide the ability to write custom IO values. The only way to get a value of type IO[Unit] is to call the *print, println or join* functions. Therefore the implementation of those functions is known at compile-time and can be implemented directly using anonymous function (Listing 5.2).

```
static IO printFula(final int num) {
    return new IO() {
        public void unsafeRunSync() {
            System.out.print(num);
        }
    };
};

static IO printlnFula(final int num) {
    return new IO() {
        public void unsafeRunSync() {
            System.out.println(num);
        }
    };
};

static IO join(IO a, IO b) {
    return new IO() {
        public void unsafeRunSync() {
            a.unsafeRunSync();
            b.unsafeRunSync();
            return;
        }
    };
}
```

Listing 5.2: Implementation in Java of functions returning an IO value.

The printFula static method is the implementation of the *print* function in the Fula language for the Int type. When called it returns an IO object with the interface method implemented. The implemented method call the print method of the java language with the argument passed to the static method. Therefore the IO value is just a on object that embeds a function performing a side effect. The join method works the same way. It takes two IO objects as arguments and returns an anonymous IO object with its interface method implemented. The implementation of the interface method calls sequentially the method of the two IO objects passed as arguments.

The IO type could be easily to a type class. The IO interface would have a generic object type. The implementation of the interface method embedded in the IO object would be linked in the object at runtime using the invokedynamic instruction has explained in to precedent section.

Chapter 6

Conclusion

6.1 Evaluation

6.1.1 Compiler Requirements Review

The Fula programming language meets all the criteria that makes it a usable purely functional programming language. Evaluating against the success criteria described in chapter 3 leads to:

Designing a purely functional programming language on the JVM Fula is a purely functional programming language. It gives the ability to the user to write some complex algorithms on the JVM in a purely functional way.

Representing effectful computation as values The IO[Unit] type of Fula allows the user to use side effects as value and to combine them. The only side effects implemented is printing to the screen, but it is enough for experiments.

Implementing first-class functions on the JVM The Fula language allows the user to write lambda expressions and assigned lifted functions to values. first-class functions are implemented using the invokedynamic instruction of the JVM. This same instruction can be used to make the IO type a real type class.

Implementing a static type-checker The Fula compiler has a static and strong type-checker that is capable of inferring value's types. It enables the user to write programs that will not trigger runtime errors, proving the correctness of their algorithm.

6.1.2 Rooms for improvments

Programs can be hard to write in the language for newcomers because the errors given by the compiler are poorly described. Any syntax error would trigger the same error log and gives no hint to the user how to recover from it. Discarding never used expressions should be moved after the type checking part. Even if the compiled program would perfectly works because the expressions with type errors are discarded, it lets the user write programs in Fula that contains errors. The compiler will never let the user know his program works and the user could have a false sense of correctness of his program.

6.2 Summary

The main goal of the language was to empower users with a language that will allow them to discover functional programming by writing small algorithms and printing their output to the screen in a functional way. It has been fulfilled and the source code is shipped with some examples that shows how such algorithms could be written in Fula(mandelbrot set, sierpinski triangle..).

6.2.1 Future works

The major extensions that Fula would follow are the implementation of type classes and list processing. It would allow the user to discover the real strength of functional programming languages. The user will be able to use high-order functions and first-class function in a more meaningful way. The type classes would support ad-hoc polymorphism that would allow the user to discover an alternative to subtype polymorphism of object oriented programming language.

References

[1] ASM. Asm library home page. https://asm.ow2.io/. Accessed 2021-02-12.

[2] Bill Campbell, Swami Iyer, and Bahar Akbal-Delibas. *Introduction to Compiler Construction in a Java World*. CRC Press Taylor Francis Group, Boca Raton, FL, 2013.

[3] Paul Chiusano and Runar Bjarnason. *Functional Programming in Scala*. Manning Publications Co., Shelter Island, NY 11964, 2015.

[4] Haskell Community. Algebraic data type. https://wiki.haskell.org/Algebraic_data_type. Accessed 2020-11-14.

[5] Haskell community. Introduction to io. https://wiki.haskell.org/Introduction_to_IO, 2020. Accessed 2020-12-02.

[6] Ben Evans. Understanding java method invocation with invokedynamic. https://www.oracle.com/a/ocom/docs/corporate/java-magazine-nov-dec-2017.pdf#page=67. Accessed 2021-03-15.

[7] Li Haoyi. Fastparse 2.2.2. https://com-lihaoyi.github.io/fastparse. Accessed 2021-01-15.

[8] Li Haoyi. *Hands-on Scala Programming*. Li Haoyi, 2020.

[9] Alexander Krolik. Symbol tables comp 520. https://www.cs.mcgill.ca/~cs520/2019/slides/7-symbol.pdf, 2019. Accessed 2021-02-05.

[10] Oracle. Chapter 6. the java virtual machine instruction set. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html. Accessed 2021-03-15.

[11] Oracle. Class lambdametafactory. https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/LambdaMetafactory.html. Accessed 2021-03-15.

[12] Bill Venners. *Inside the Java 2 Virtual Machine*. McGraw-Hill Education, 2000.

Appendix A

User Guide

The user guide is a specification of the Fula language. It serves as a guide to write Fula programs.

A.1 Types

A.1.1 Single types

There are five single primitive types: **Int, Float, Boolean, String, IO**[**Unit**]:

- **Int** values are 32-bit signed integers: -2147483648 to 2147483647
- **Float** values are 32 bit IEEE 754 single-precision float.
- **Boolean** values are either **True** or **False**.
- **String** values starts and ends with a **”**. They are composed of digits, letters and ”_”, ”∗”, ” ” (whitespace), ”.” characters.
- **IO**[**Unit**] values represent effectful computations. The only way to get an IO[Unit] value in Fula is to call *print or println* functions.

A.1.2 Function type

Functions type are composition types of single primitive types. They are written as: **Int,String** => **Boolean**, a function that takes an Int and String as arguments and returns a Boolean. Function types that takes no parameter are written as: => **Boolean** . Function type can only take single primitive types.

A.2 Function declaration

A Fula program is a sequence of function declarations. The last function declared is the main function. Functions are separated using a semi-column. Don't forget the semi column after the main function declaration.

```
def function1(x: Int): Int = {
...
};

def function2(x: String): Float = {
...
};

def main(): IO[Unit] = {
...
};
```

A.2.1 function declaration

In Fula the types have to be annotated when declaring a function. The syntax is really similar to the method definition of the Scala language.

```
def function(x1: Int, x2: Float): String = {
...
};
```

There is no return statement in a function declaration, the last expression of the body is considered as the *returned* expression.

```
def max(x:Int, y:Int): Int = {
        if (x>y) {
                x; // return value if x>y True
        } else {
                y; // return value if x>y False
        };
};
```

Function declared are high-order functions: they can take functions arguments and return function.

```
def specialMap(x:Int, f: Int => Int): Int,Int => Int = {
        (x:Int,y:Int): Int => f(x)+f(y); // this is a lambda expression
};
```

A.2.2 main function

The **main** function takes no parameter and returns an IO[Unit] value.

```
def main(): IO[Unit] = {
        print("Hello") >> print("World!");
};
```

At runtime the effectful computation returned by the main function will be executed.

A.2.3 function body

The body of a function is a sequence of expressions. Expressions are separated using a semi-column. Don't forget the semi column after the last expression.

```
def rsa(p: Int, q: Int): Int = {
        val p = n * q;
        val z = (p - 1) * (q - 1);
        ...
        rsaValue;
};
```

A.2.4 syntatic sugar: val functions

At the root level functions that takes no parameters and has a single expression as body can be written using the val syntax. The type has to be annotated. Those functions are then called as if they were regular functions with no parameters.

```
val p: Int = 10;
val z: Int = 30;


def main(): IO[Unit] = {
        val res = p() * z();
        println(res);
};
```

A.2.5 default language functions

There are two default functions: *print and println.* Those functions do not print to the screen directly but return a value that represents an effectful computation. Values returned by those functions can be combined together using the sequence operator:

```
def main(): IO[Unit] = {
        println("Fula") >> println("is") >> println("amazing.");
};
```

A.3 Expressions

An expression is an entity that can be evaluated and assigned to a value.

A.3.1 Assigning an expression to a value

Assigning an expression to a value is itself. It has type Unit, avoid putting the value onto the stack and directly popping most of the time (Fula targets the stack machine JVM). Types are inferred by the compiler but can be annotated.

```
def func(): Int = {
        val x = 10;
        val y = 10.02;
        val z: Float = y;
        ...
};
```

A.3.2 Arithmetic Expression

Arithmetic expressions can only be used with Int and Float types. A Float can not be added to an Int. They return a value of the same of the computed values. The operators supported are:

- + add
- - subtract
- / divide
- $*$ multiplicate
- % modulo
- & bitwise AND: **only for Int**

A.3.3 Boolean Expression

Boolean expressions are evaluated as **Boolean** values. The supported boolean operators are:

- == for **Int, Float, Boolean and String**
- ! = for **Int, Float and Boolean**
- >= for **Int and Float**
- <= for **Int and Float**
- > for **Int and Float**
- < for **Int and Float**

A.3.4 If-else Expression

If-else expression takes a Boolean expression and are evaluated to the last expression of the block executed. The type of the expression is the type of the two blocks. Therefore the type of the two blocks must be equal.

```
        ...
        val res = if (in == 10) {
                in;
        } else {
                in*2;
        };
        ...
};
```

A.3.5 Lambda Expression

A lambda expression is an anonymous function assigned a value. The type of the lambda expression has to be annotated it can not be inferred.

```
        ...
        val lambda1 = (x:Int):Int => x+x/2;
        val lambda2: Float,Float => Float = (x:Float,y:Float):Float => (x+y)*(x
            ↪ -y);
        ...
};
```

A.3.6 Lifted function declarations

First-order (not high-order) declared functions can be lifted to function expressions and assigned to a value.

```
def add3(x:Int,y:Int,z:Int):Int = {
        x+y+z;
};

def main(): IO[Unit] = {
        val addNotAnnotated = add3;
        val addAnnotated: Int,Int,Int => Int = add3;
        ...
};
```

Appendix B

Source Code

Source code listing of the Fula compiler and the runtime library.

B.1 AST

B.1.1 The Fula Type Algebraic Data Type

Listing B.1: The FLType (Fula Type) algebraic data type together with its corresponding companion object. FLType.scala file.

```
package ast;


trait FLType
case object FLObject extends FLType
case object FLUnknown extends FLType
case object FLInt extends FLType
case object FLIntObject extends FLType
case object FLFloat extends FLType
case object FLFloatObject extends FLType
case object FLBoolean extends FLType
case object FLBooleanObject extends FLType
case object FLUnit extends FLType
case object FLString extends FLType
case class FLIO(typ: FLType) extends FLType
case class FLWrap(typ: FLType) extends FLType
case class FLFunc(args: Seq[FLType], typ: FLType) extends FLType

object FLType {

        def createFunctionType(args: Seq[FLType],retType: FLType) : FLFunc = {
                val wrapArgs: Seq[FLType] = args.map(typ => wrapTypeToFunction(
                    ↪ typ))
                val wrapRet: FLType = wrapTypeToFunction(retType)
                FLFunc(wrapArgs,wrapRet)
        }

        def createIOType(): FLType = FLIO(FLUnit)

        def createSingleType(typ: String): FLType = typ match {
                case "Int" => FLInt
                case "Float" => FLFloat
                case "Boolean" => FLBoolean
                case "String" => FLString
                case _ => createIOType()
        }

        def wrapTypeToFunction(typ: FLType) = typ match {
                case FLFunc(_,_) => FLWrap(typ)
                case _ => typ
        }

        def getFunctionArgTypes(typ: FLType) : Seq[FLType] = typ match {
                case FLWrap(FLFunc(args,typ)) => args
                case FLFunc(args,typ) => args
                case _ => Seq()
        }

        def getFunctionReturnType(typ: FLType) : FLType = typ match {
```

```
		case FLWrap(FLFunc(args,retType)) => retType
		case FLFunc(args,retType) => retType
		case _ => typ
	}

}
```

B.1.2 The Abstract Syntax Tree Algebraic Data Type

Listing B.2: The AST algebraic data type together with its corresponding companion object. The Ast.scala file.

```
package ast;

object Ast {

	sealed trait AstNode

	sealed trait Tok
	object Tok {
		case class Identifier(iden: String) extends Tok
		case class IntegerTok(num: Int) extends Tok
		case class FloatTok(num: Float) extends Tok
		case class BooleanTok(num: Boolean) extends Tok
		case class StringTok(str: String) extends Tok
		case class Type(typ: String) extends Tok {
			override def toString() = typ
		}
	}

	type Block = Seq[Expr]

	sealed trait Expr extends AstNode
	object Expr {
		case class If(a: Bexp, e1: Block, e2: Block) extends Expr
```

```
        case class Assign(name: String, args: Seq[Expr]) extends Expr
        case class Value(s: String) extends Expr
        case class IntExpr(i: Int) extends Expr
        case class FloatExpr(d: Float) extends Expr
        case class BooleanExpr(b: Boolean) extends Expr
        case class StringExpr(str: String) extends Expr
        case class Aop(o: String, a1: Expr, a2: Expr) extends Expr
        case class Val(name: String, typ: FLType, e: Expr) extends Expr
        case class Lambda(args: Seq[(String,FLType)],typ: FLType, e:
            ↪ Expr) extends Expr

        case class WriteLn(e: Expr) extends Expr
        case class Write(e: Expr) extends Expr
        case class Join(io1: Expr, io2: Expr) extends Expr

        sealed trait Bexp extends Expr
        object Bexp {
                case class Bop(o: String, a1: Expr, a2: Expr) extends
                    ↪ Bexp
        }
}

sealed trait Decl extends AstNode
object Decl {
        case class Def(name: String, pack: String, args: Seq[(String,
            ↪ FLType)], typ: FLType, body: Block) extends Decl
        case class Main(name: String="main", pack: String, typ: FLType=
            ↪ FLFunc(Seq(),FLIO(FLUnit)), body: Block) extends Decl
}


type ParseProg = Seq[AstNode]
```

```
        type Prog = Seq[Decl]
}
```

B.1.3 The Typed Abstract Syntax Tree Algebraic Data Type

Listing B.3: The typed AST algebraic data type together with its corresponding companion object. TypeAst.scala file.

```
package ast;


object TypeAst {


        sealed trait TypeAstNode


        type TypeBlock = Seq[TypeExpr]


        sealed trait TypeExpr extends TypeAstNode
        object TypeExpr {
                case class TyIf(a: TypeBexp, e1: TypeBlock, e2: TypeBlock, typed
                    ↪ : FLType) extends TypeExpr
                case class TyAssign(name: String, pack: String, args: Seq[
                    ↪ TypeExpr], typed: FLType) extends TypeExpr
                case class TyApply(name: String, args: Seq[TypeExpr], typed:
                    ↪ FLType) extends TypeExpr
                case class TyValue(s: String, typed: FLType) extends TypeExpr
                case class TyIntExpr(i: Int, typed: FLType=FLInt) extends
                    ↪ TypeExpr
                case class TyFloatExpr(d: Float, typed: FLType=FLFloat) extends
                    ↪ TypeExpr
                case class TyBooleanExpr(b: Boolean, typed: FLType=FLBoolean)
                    ↪ extends TypeExpr
                case class TyStringExpr(str: String, typed: FLType=FLString)
                    ↪ extends TypeExpr
                case class TyAop(o: String, a1: TypeExpr, a2: TypeExpr, typed:
                    ↪ FLType) extends TypeExpr
        case class TyVal(name: String, e: TypeExpr, exprTyp: FLType,
            ↪ typed: FLType=FLUnit) extends TypeExpr
        case class TyFunction(name: String, pack: String, typed: FLType)
            ↪  extends TypeExpr

        sealed trait TypeBexp extends TypeExpr
        object TypeBexp {
                case class TyBop(o: String, a1: TypeExpr, a2: TypeExpr,
                    ↪ exprTyp: FLType, typed: FLType=FLBoolean) extends
                    ↪ TypeBexp
        }
}

sealed trait TypeDecl extends TypeAstNode
object TypeDecl {
        case class TyDef(name: String, pack: String, args: Seq[(String,
            ↪ FLType)], body: TypeBlock, typed: FLFunc) extends
            ↪ TypeDecl
        case class TyMain(name: String="main", pack: String, body:
            ↪ TypeBlock, typed: FLFunc=FLFunc(Seq(),FLIO(FLUnit)))
            ↪ extends TypeDecl
}

type TypeProg = Seq[TypeDecl]



def getNodeType(node: TypeAstNode) : FLType = node match {
        case TypeExpr.TypeBexp.TyBop(_,_,_,_,typ) => typ
        case TypeExpr.TyIf(_,_,_,typ) => typ
        case TypeExpr.TyAssign(_,_,_,typ) => typ
        case TypeExpr.TyApply(_,_,typ) => typ
        case TypeExpr.TyValue(_,typ) => typ
        case TypeExpr.TyFunction(_,_,typ) => typ
```

```
            case TypeExpr.TyIntExpr(_,typ) => typ
            case TypeExpr.TyFloatExpr(_,typ) => typ
            case TypeExpr.TyBooleanExpr(_,typ) => typ
            case TypeExpr.TyStringExpr(_,typ) => typ
            case TypeExpr.TyAop(_,_,_,typ) => typ
            case TypeExpr.TyVal(_,_,_,typ) => typ
            case TypeDecl.TyDef(_,_,_,_,typ) => typ
            case TypeDecl.TyMain(_,_,_,typ) => typ
        }

        def getBlockType(block: TypeBlock) : FLType = {
            try {getNodeType(block.last) }
            catch {case e: Exception => FLUnit }
        }
}
```

B.2 Data Structures

B.2.1 Symbol

Listing B.4: The Symbol data structure together with its corresponding companion object. Symbol.scala file.

```
package struct;
import ast.Ast._;
import ast.FLType._;
import ast._;



case class Symbol(name: String, symTyp: FLType, pack: Option[String])
case class SymbolSignature(name: String, argsTyp: Option[Seq[FLType]])

object Sym {

        def createSymbolSignature(sym: Symbol) : SymbolSignature = {
```

```
        val symName: String = sym.name;
        val argsTyp: Option[Seq[FLType]] = sym.symTyp match {
                case FLFunc(args,typ) => Some(args)
                case _ => None
        }
        SymbolSignature(symName,argsTyp)
    }

    def createSymbolDecl(node: Ast.Decl) : Symbol = node match {
        case Decl.Def(name,pack,args,typ,_) => Symbol(name,
            ↪ createFunctionType(args.map(_._2),typ), Some(pack))
        case Decl.Main(name,pack,typ,_) => Symbol(name,FLFunc(Seq(),typ)
            ↪ , Some(pack))
    }

    def createSymbolArg(node: (String,FLType)) : Symbol = {
        Symbol(node._1,wrapTypeToFunction(node._2), None)
    }

    val declToSymbols = (nodes: Seq[Ast.Decl]) => nodes.map(node =>
        ↪ createSymbolDecl(node))
    val argsToSymbols = (nodes: Seq[(String,FLType)]) => nodes.map(arg =>
        ↪ createSymbolArg(arg))

}
```

B.2.2 Symbol Table

Listing B.5: The Symbol Table data structure together with its corresponding companion object. SymbolTable.scala file.

```
package struct;

import struct.Sym._;
import ast._;
```

```
trait SymbolTable

case class Package(table: Map[SymbolSignature,Symbol]) extends SymbolTable
case class Root(table: Map[String, Symbol], pack: SymbolTable) extends
    ↪ SymbolTable
case class Node(table: Map[String, Symbol], parent: SymbolTable) extends
    ↪ SymbolTable

object SymbolTable {

        def getSymbol(symName: String, typCall: Option[Seq[FLType]], symT:
            ↪ SymbolTable) : Either[String, Symbol] = {
                lookup(SymbolSignature(symName,typCall),symT) match {
                        case Some(Symbol(name,typ,pack)) => if (symName==name)
                            ↪ Right(Symbol(name,typ,pack)) else Left("The␣value␣
                            ↪ " + symName + "␣is␣not␣defined.")
                        case None => {
                                Left("The␣value␣" + symName + "␣is␣not␣defined.")
                        }
                }
        }

        def putSymbol(symName: String, symType: FLType, packName: Option[String
            ↪ ], symT: SymbolTable) : Either[String, SymbolTable] = {
                putSymbol(Symbol(symName, symType, packName),symT)
        }

        def putSymbol(symbol: Symbol, symTable: SymbolTable): Either[String,
            ↪ SymbolTable] = {
                if (alreadyDeclared(symbol,symTable,lookupScope)) {
                        Left(errorLogging(symbol))
                } else {
```

```
                symTable match {
                        case Package(map) => Right(Package(map + (
                            ↪ createSymbolSignature(symbol) -> symbol)))
                        case Root(map,pack) => Right(Root(map + (symbol.
                            ↪ name -> symbol),pack))
                        case Node(map, parent) => Right(Node(map + (
                            ↪ symbol.name -> symbol),parent))
                }
        }
}

def putMultipleSymbol(symbols: Seq[Symbol], symT: SymbolTable): Either[
    ↪ String,SymbolTable] = symbols match {
        case symbol::xs => for {
                newSymT <- putSymbol(symbol,symT)
                symTab <- putMultipleSymbol(xs,newSymT)
        } yield (symTab)
        case Nil => Right(symT)
}

def errorLogging(sym: Symbol) : String = sym.symTyp match {
        case FLFunc(_,_) => "Function␣" + sym.name + "␣is␣already␣
            ↪ defined."
        case _ => "Value␣" + sym.name + "␣is␣already␣defined."
}

def openScope(symTable: SymbolTable) : SymbolTable = {
        Node(Map(),symTable)
}

def alreadyDeclared(sym: Symbol, symTable: SymbolTable, f: (
    ↪ SymbolSignature, SymbolTable) => Option[Symbol]) : Boolean = {
        f(createSymbolSignature(sym),symTable).map(symbol => symbol.name
                ↪ ==sym.name).getOrElse(false)
        }

        def lookup(symbol: SymbolSignature, symTable: SymbolTable) : Option[
            ↪ Symbol] = symTable match {
                case Package(table) => getFromMap(symbol,table)
                case Root(table,pack) => if (getFromMap(symbol.name,table) !=
                    ↪ None) getFromMap(symbol.name,table) else lookup(symbol,
                    ↪ pack)
                case Node(table,parent) => if ( getFromMap(symbol.name,table) !=
                    ↪  None) getFromMap(symbol.name,table) else lookup(symbol,
                    ↪ parent)
        }

        def lookupScope(symbol: SymbolSignature, symTable: SymbolTable) :
            ↪ Option[Symbol] = symTable match {
                case Package(table) => getFromMap(symbol,table)
                case Root(table,pack) => if (getFromMap(symbol.name,table) !=
                    ↪ None) getFromMap(symbol.name,table) else lookup(symbol,
                    ↪ pack)
                case Node(table,parent) => getFromMap(symbol.name,table)
        }

        def getFromMap[A](symbol: A, table: Map[A,Symbol]) : Option[Symbol] = {
                try {
                        Some(table(symbol))
                } catch {
                        case e:Exception => None
                }
        }
}
```

B.3 Frontend

B.3.1 Parser

Lexical Parser

Listing B.6: The Lexical token and Type parser combinators. Lexicals.scala file.

```scala
package frontend.parser;

import ast.Ast;
import fastparse._
import fastparse.MultiLineWhitespace._

import ast._;
import ast.FLType._;

object Lexicals {

        val keywordList: Set[String] = Set("if", "else", "def", "main", "val",
            ↪ "printlnFula", "True", "False", "printFula", "IO", "println", "
            ↪ print", "join", "unsafeRunSync", "helper", "Int", "Float", "
            ↪ Boolean", "String", "IO[Unit]")

        def lowercase [_ : P] : P[String] = P( CharIn("a-z") ).!
        def uppercase[_ : P] : P[String] = P( CharIn("A-Z") ).!
        def letter[_ : P] : P[String] = P( lowercase | uppercase ).!
        def digit [_ : P] : P[String] = P( CharIn("0-9") ).!
        def unsignedIntegerStr[_ : P] : P[String] = P( digit.rep(1) ).!
        def IntegerStr[_ : P] : P[String] = P ( "-".?.! ~ unsignedIntegerStr).!
        def BooleanStr[_ : P] : P[String] = P ("True" | "False").!

        def Boolean[_ : P] : P[Ast.Tok.BooleanTok] = P(BooleanStr).map(_.
            ↪ toBoolean).map{ case bool => Ast.Tok.BooleanTok(bool) }
        def Integer[_ : P] : P[Ast.Tok.IntegerTok] = P(IntegerStr).map(_.toInt)
            ↪ .map{ case num => Ast.Tok.IntegerTok(num)}
        def Float[_ : P] : P[Ast.Tok.FloatTok] = P ( IntegerStr ~ "." ~
            ↪ unsignedIntegerStr ).!.map(_.toFloat).map{ case num => Ast.Tok.
            ↪ FloatTok(num)}

        def StringLex[_ : P] : P[Ast.Tok.StringTok] = P (( letter | digit | "_"
            ↪  | "*" | "␣" | ".").repX ).!.map{ case str => Ast.Tok.StringTok(
            ↪ str) }

        def IOType[_ : P] : P[FLType] = P( ("IO[Unit]")).!.map{ case (typ) =>
            ↪ FLType.createIOType() }
        def SingleType[_ : P] : P[FLType] = P( ("Int" | "Float" | "Boolean" | "
            ↪ String" )).!.map{ case (typ) => FLType.createSingleType(typ) }
        def SingleTypes[_ : P] : P[FLType] = P ( SingleType | IOType )
        def FuncType[_ : P] : P[FLType] = P ( SingleTypes.rep(0,",") ~ "=>" ~
            ↪ SingleTypes ).map{ case (seqTyp,typ) => FLFunc(seqTyp,typ) }
        def Type[_ : P] : P[FLType] = P( FuncType | IOType | SingleType )

        def Identifier[_ : P]: P[Ast.Tok.Identifier] = P( letter ~ (letter |
            ↪ digit | "_").repX).!.filter(!keywordList.contains(_)).map{ case
            ↪ iden => Ast.Tok.Identifier(iden)}

}
```

Expression Parser

Listing B.7: The Expression parser combinators. Expressions.scala file.

```
package frontend.parser;

import ast.Ast;
import ast._;
import ast.FLType._;
import fastparse._
import fastparse.MultiLineWhitespace._
```

```
object Expressions {

        def eq[_ : P] : P[Ast.Expr.Bexp] = P(atom_bexp ~ "==" ~ atom_bexp).map{
            ↪  case (x, z) => Ast.Expr.Bexp.Bop("==", x, z)}
        def diff[_ : P] : P[Ast.Expr.Bexp] = P(atom_bexp ~ "!=" ~ atom_bexp).
            ↪ map{ case (x, z) => Ast.Expr.Bexp.Bop("!=", x, z)}
        def lt[_ : P] : P[Ast.Expr.Bexp] = P(atom_bexp ~ "<" ~ atom_bexp).map{
            ↪ case (x, z) => Ast.Expr.Bexp.Bop("<", x, z)}
        def lte[_ : P] : P[Ast.Expr.Bexp] = P(atom_bexp ~ "<=" ~ atom_bexp).map
            ↪ { case (x, z) => Ast.Expr.Bexp.Bop("<=", x, z)}
        def gt[_ : P] : P[Ast.Expr.Bexp] = P(atom_bexp ~ ">" ~ atom_bexp).map{
            ↪ case (x, z) => Ast.Expr.Bexp.Bop("<", z, x)}
        def gte[_ : P] : P[Ast.Expr.Bexp] = P(atom_bexp ~ ">=" ~ atom_bexp).map
            ↪ { case (x, z) => Ast.Expr.Bexp.Bop("<=", z, x)}
        def bexp_paren[_ : P] : P[Ast.Expr.Bexp] = P( "(" ~ bexp ~ ")" )

        def atom_bexp[_ : P] : P[Ast.Expr] = P ( boolean | aexp | string_expr )

        def bexp[_ : P]: P[Ast.Expr.Bexp] =  P( eq | diff | lt | lte | gt | gte
            ↪  | bexp_paren )

        def int[_: P] : P[Ast.Expr.IntExpr] = P (Lexicals.Integer).map{ case(
            ↪ Ast.Tok.IntegerTok(num)) => Ast.Expr.IntExpr(num)}
        def double[_: P] : P[Ast.Expr.FloatExpr] = P (Lexicals.Float).map{ case
            ↪ ( Ast.Tok.FloatTok(num)) => Ast.Expr.FloatExpr(num)}
        def boolean[_ : P] : P[Ast.Expr.BooleanExpr] = P(Lexicals.Boolean).map{
            ↪  case( Ast.Tok.BooleanTok(bool)) => Ast.Expr.BooleanExpr(bool)}
        def value[_ : P] : P[Ast.Expr.Value] = P (Lexicals.Identifier).map{
            ↪ case(Ast.Tok.Identifier(id)) => Ast.Expr.Value(id) }
        def string_expr[_ : P] : P[Ast.Expr.StringExpr] = P ( "\"" ~~ Lexicals.
            ↪ StringLex ~~ "\"" ).map{ case(Ast.Tok.StringTok(str)) => Ast.
            ↪ Expr.StringExpr(str) }
```

```
def chainA[_: P](p: => P[Ast.Expr], op: => P[String]) = P( p ~ (op ~ p)
    ↪ .rep ).map{case (lhs, rhs) =>rhs.foldLeft(lhs){case (lhs, (op,
    ↪ rhs)) =>Ast.Expr.Aop(op, lhs, rhs)}}
def aexp_paren[_ : P] : P[Ast.Expr] = P ( "(" ~ aexp ~ ")" )
def op[_ : P](op: String) = P (op).!

def aexp[_ : P] : P[Ast.Expr] = P (chainA(factor,op("+")|op("-")))
def factor[_ : P] : P[Ast.Expr] = P (chainA(atom,op("*")|op("%")|op("/"
    ↪ )|op("&")))
def atom[_ : P] : P[Ast.Expr] = P ( assign_expr | aexp_paren | value |
    ↪ double | int )

def semi_chain[_ : P](p: => P[Ast.Expr]): P[Seq[Ast.Expr]] = P( p.rep
    ↪ (1,";"))

def comma_chain[_ : P](p: => P[Ast.Expr]): P[Seq[Ast.Expr]] = P( p.rep
    ↪ (0,",") )

def if_expr[_ : P] : P[Ast.Expr.If] = P ( "if" ~ "(" ~ bexp ~ ")" ~ "{"
    ↪  ~ block ~ "}" ~ "else" ~ "{" ~ block ~ "}").map{
        case (bexp,if_seq,else_seq) => Ast.Expr.If(bexp,if_seq,else_seq)
}

def assign_expr[_ : P]: P[Ast.Expr.Assign] = P ( Lexicals.Identifier ~
    ↪ "(" ~ comma_chain(expr) ~ ")" ).map{
        case (Ast.Tok.Identifier(id),exprs) => Ast.Expr.Assign(id, exprs
            ↪ )
}

def writeln_expr[_ : P]: P[Ast.Expr] = P ("println" ~ "(" ~ expr ~ ")")
    ↪ .map{ case (expr) => Ast.Expr.WriteLn(expr) }
def write_expr[_ : P]: P[Ast.Expr] = P ("print" ~ "(" ~ expr ~ ")").map
```

```
    ↪ { case (expr) => Ast.Expr.Write(expr) }

def inside_join[_ : P] : P[Ast.Expr] = P (if_expr | writeln_expr |
    ↪ write_expr | assign_expr | value )
def join[_ : P]: P[Ast.Expr] = P (inside_join ~ (">>" ~ inside_join).
    ↪ rep(1)).map{case (lhs, rhs) =>rhs.foldLeft(lhs){case (lhs, rhs)
    ↪ => Ast.Expr.Join(lhs, rhs)}}

def val_expr[_ : P]: P[Ast.Expr.Val] = P ( "val" ~ Lexicals.Identifier
    ↪ ~ (":" ~ Lexicals.Type).? ~ "=" ~ expr ).map{ case(Ast.Tok.
    ↪ Identifier(id),Some(typ),expr) => Ast.Expr.Val(id,typ,expr) case
    ↪ (Ast.Tok.Identifier(id),None,expr) => Ast.Expr.Val(id,FLUnknown,
    ↪ expr) }

def val_func[_ : P]: P[Ast.Expr.Val] = P ( "val" ~ Lexicals.Identifier
    ↪ ~ ":" ~ Lexicals.Type ~ "=" ~ expr ).map{ case(Ast.Tok.
    ↪ Identifier(id),typ,expr) => Ast.Expr.Val(id,typ,expr) }

def lambda_expr[_: P]: P[Ast.Expr.Lambda] = P ( ArgsLambda ~ ":" ~
    ↪ Lexicals.SingleTypes ~ "=>" ~ expr ).map{ case(args,typ,expr) =>
    ↪  Ast.Expr.Lambda(args,typ,expr) }

def ArgsLambda[_ : P]: P[Seq[(String,FLType)]] = P ("(" ~ (Lexicals.
    ↪ Identifier.map{ case(Ast.Tok.Identifier(id)) => id } ~ ":" ~
    ↪ Lexicals.SingleTypes).rep(0,",") ~ ")")

def expr[_ : P]: P[Ast.Expr] = P ( if_expr | join | writeln_expr |
    ↪ write_expr | lambda_expr | bexp | aexp | assign_expr |
    ↪ string_expr | double | boolean | value | int )

def block_expr[_ : P]: P[Ast.Expr] = P ( val_expr | expr )

def block[_ : P]: P[Ast.Block] = P ( Expressions.semi_chain(Expressions
    ↪ .block_expr) ~ ";" | Expressions.block_expr.map{ case expr =>
    ↪ List(expr) } ~ ";")

}
```

Declaration Parser

Listing B.8: The Declaration parser combinators. Declaration.scala file.

```
package frontend.parser;


import ast.Ast;
import ast._;
import ast.FLType._;
import fastparse._
import fastparse.MultiLineWhitespace._


object Declaration {


    def Args[_ : P]: P[Seq[(String,FLType)]] = P ((Lexicals.Identifier.map{
        ↪ case(Ast.Tok.Identifier(id)) => id } ~ ":" ~ Lexicals.Type).rep
        ↪ (0,","))


    def Func[_: P]: P[Ast.Decl] = P( "def" ~ Lexicals.Identifier ~ "(" ~
        ↪ Args ~ ")" ~ ":" ~ Lexicals.Type ~ "=" ~ "{" ~ Expressions.block
        ↪ ~ "}" ).map{
            case(Ast.Tok.Identifier(id),args,typ,block) => Ast.Decl.Def(id,"
                ↪ XXXX",args,typ,block)
    }


    def Main[_: P]: P[Ast.Decl] = P( "def" ~ "main" ~ "(" ~ ")" ~ ":" ~ "IO
        ↪ [Unit]" ~ "=" ~ "{" ~ Expressions.block ~ "}" ).map{
            case(block) => Ast.Decl.Main(pack="XXXX",body=block)
    }
}
```

Fula Parser

Listing B.9: The Fula language parser combinators. Fula.scala file.

```
package frontend.parser;

import ast.Ast;
import fastparse._
import fastparse.MultiLineWhitespace._
import java.beans.Expression

object Fula {

  def FuncDef[_ : P]: P[Ast.AstNode] = ( Declaration.Func | Expressions.
      ↪ val_func )

  def FuncList[_ : P]: P[Ast.ParseProg] = FuncDef.rep(1,";")

        def Prog[_ : P]: P[Ast.ParseProg] = P ( Declaration.Main.map{ case main
            ↪ => List(main)} ~ ";" |
                                         (FuncList ~ ";" ~ Declaration.Main ~ ";").
                                             ↪ map{ case(funcs,main) => funcs :+
                                             ↪ main}
                                       )
}
```

Top program Parser

Listing B.10: Execute the top program parser. ProgParser.scala file.

```
package frontend.parser;

import ast.Ast;
import fastparse._;
import fastparse.Parsed;
```

```
object ProgParser {

        def parseProg(prog: String): Either[String, Ast.ParseProg] = fastparse
            ↪ .parse(prog, Fula.Prog(_)) match {
                        case Parsed.Success(tree,_) => Right(tree)
                        case Parsed.Failure(_,_,_) => Left("Syntax␣error.")
        }
}
```

B.3.2 Desugar

Desugar Expression

Listing B.11: Functions used to desugar expressions. DesugarExpr.scala file.

```
package frontend.desugar;

import ast.TypeAst._;
import ast.FLType._;
import ast._;

object DesugarExpr {

        def desugarArgsCall(argList: Seq[Ast.Expr], className: String, defList:
            ↪  DefList) : Either[String, (Seq[Ast.Expr],DefList)] = argList
            ↪ match {
                case arg::xs => for {
                        desugaredArg <- desugarExpr(arg,className,defList)
                        desugaredArgs <- desugarArgsCall(xs,className,
                            ↪ desugaredArg._2)
                } yield (desugaredArg._1+:desugaredArgs._1,desugaredArgs._2)
                case Nil => Right(List(),defList)
        }

        def desugarExpr(expr: Ast.Expr,className: String, defList: DefList) :
            ↪ Either[String,(Ast.Expr,DefList)] = expr match {
```

```
case Ast.Expr.Lambda(args,typ,e) => desugarExpr(e,className,
    ↪ defList).map(ret => {
        val liftedFunction: (String,DefList) = ret._2.addLambda(
            ↪ Ast.Expr.Lambda(args,typ,ret._1),className)
        (Ast.Expr.Value(liftedFunction._1),liftedFunction._2)
})
case Ast.Expr.Write(e1) => desugarExpr(e1,className,defList).map
    ↪ (de1 => (Ast.Expr.Assign("printFula", Seq(de1._1)),de1._2
    ↪ ))
case Ast.Expr.WriteLn(e1) => desugarExpr(e1,className,defList).
    ↪ map(de1 => (Ast.Expr.Assign("printlnFula", Seq(de1._1)),
    ↪ de1._2))
case Ast.Expr.Join(e1,e2) => for {
        de1 <- desugarExpr(e1,className,defList)
        de2 <- desugarExpr(e2,className,de1._2)
} yield (Ast.Expr.Assign("join",Seq(de1._1,de2._1)),de2._2)
case Ast.Expr.Aop(op,e1,e2) => for {
        de1 <- desugarExpr(e1,className,defList)
        de2 <- desugarExpr(e2,className,de1._2)
} yield (Ast.Expr.Aop(op,de1._1,de2._1),de2._2)
case Ast.Expr.Bexp.Bop(op,e1,e2) => for {
        de1 <- desugarExpr(e1,className,defList)
        de2 <- desugarExpr(e2,className,de1._2)
} yield (Ast.Expr.Bexp.Bop(op,de1._1,de2._1),de2._2)
case Ast.Expr.If(bexp,b1,b2) => for {
        desugaredB1 <- desugarBody(b1,className,defList)
        desugaredB2 <- desugarBody(b2,className,desugaredB1._2)
} yield (Ast.Expr.If(bexp,desugaredB1._1,desugaredB2._1),
    ↪ desugaredB2._2)
case Ast.Expr.Assign(name,args) => for {
        desugaredArgs <- desugarArgsCall(args,className,defList)
} yield (Ast.Expr.Assign(name,desugaredArgs._1),desugaredArgs._2
    ↪ )
```

```
            case _ => Right(expr,defList)
        }

        def desugarBody(body: Ast.Block, className: String, defList:DefList):
            ↪ Either[String,(Ast.Block,DefList)] = body match {
                case Ast.Expr.Val(name,typ,expr)::xs => for {
                        desugaredExpr <- desugarExpr(expr,className,defList)
                        desugaredBody <- desugarBody(xs,className,desugaredExpr.
                            ↪ _2)
                } yield (Ast.Expr.Val(name,typ,desugaredExpr._1)+:desugaredBody.
                    ↪ _1,desugaredBody._2)
                case expr::Nil => desugarExpr(expr,className,defList).map(exprD
                    ↪ => (List(exprD._1),exprD._2))
                case expr::xs => desugarBody(xs,className,defList)
                case Nil => Right(List(),defList)
        }
}
```

Desugar Declaration

Listing B.12: Functions used to desugar expressions. DesugarDecl.scala file.

```
package frontend.desugar;

import ast.Ast._;
import ast._;

object DesugarDecl {

        def desugarDecl(expr: Ast.AstNode, className: String) : Either[String,
            ↪ Seq[Ast.Decl]] = expr match {
                case Ast.Expr.Val(name, typ, e) => DesugarExpr.desugarBody(Seq(e
                    ↪ ),className,DefList(name,0,Seq())).map(body => Ast.Decl.
                    ↪ Def(name, className,Seq(), typ, body._1)+:body._2.getDefs
                    ↪ )
                case Ast.Decl.Def(name,_, args, typ, body) => DesugarExpr.
                    ↪ desugarBody(body,className,DefList(name,0,Seq())).map(
                    ↪ body => Ast.Decl.Def(name, className, args, typ, body._1)
                    ↪ +:body._2.getDefs)
                case Ast.Decl.Main(name,_, typ, body) => DesugarExpr.desugarBody
                    ↪ (body,className,DefList(name,0,Seq())).map(body => body.
                    ↪ _2.getDefs:++Seq(Ast.Decl.Main(name, className, typ, body
                    ↪ ._1)))
        }
}
```

Desugar Program

Listing B.13: Functions used to the fula program. DesugarProg.scala file.

```
package frontend.desugar;


import ast.TypeAst._;
import ast.FLType._;
import ast._;


case class DefList(wrapperName: String, current: Int, defs: Seq[Ast.Decl.Def])
    ↪  {
        def addLambda(lambda: Ast.Expr.Lambda, className: String): (String,
            ↪ DefList) = {
                val newName: String = "$anonfun$" + wrapperName + "$" + current
                val newDefs: Seq[Ast.Decl.Def] = defs :+ Ast.Decl.Def(newName,
                    ↪ className,lambda.args,lambda.typ,Seq(lambda.e))
                (newName,DefList(wrapperName,current+1,newDefs))
        }


        def getDefs(): Seq[Ast.Decl.Def] = defs
}


object DesugarProg {
        def desugarProg(prog: Ast.ParseProg, className: String) : Either[String
            ↪ ,Ast.Prog] = prog match {
                case decl::xs => for {
                        desugaredDecl <- DesugarDecl.desugarDecl(decl, className
                            ↪ )
                        desugaredProg <- desugarProg(xs, className)
                } yield (desugaredDecl:++desugaredProg)
                case Nil => Right(List())
        }
}
```

B.3.3 Typer

Type equality and Type operation permission

Listing B.14: Functions used to check the equality between to types and the fact that an operator is allowed for a given type. TypeChecker.scala file.

```
package frontend.typer;

import ast.FLType._;
import ast._;

object TypeChecker {

        def typeEqual(type1: FLType, type2: FLType): Either[String,FLType] = if
            ↪  (typeEqualBool(type1,type2) == true) if (type1 != FLUnknown)
            ↪ Right(type1) else Right(type2) else Left("Type␣error.")

        def typeEqualBool(type1: FLType, type2: FLType): Boolean = {
                if (type1 == type2) true else (type1,type2) match {
                        case (FLUnknown,_) => true
                        case (_,FLUnknown) => true
                        case (FLFunc(_,_),FLWrap(typ)) if typ==type1 => true
                        case (FLWrap(typ),FLFunc(_,_)) if typ==type2 => true
```

```
                    case _ => false
                }
        }

        def typeEqualList(argsExpr: Seq[FLType], argsType: Seq[FLType]) :
            ↪ Either[String, Seq[FLType]] = (argsExpr,argsType) match {
                case (type1::xs1,type2::xs2) => if (typeEqualBool(type1,type2))
                    ↪ typeEqualList(xs1,xs2).map{ seq => type1+:seq } else Left
                    ↪ ("Type␣error␣in␣function␣call.")
                case (Nil,Nil) => Right(argsExpr)
                case (_,_) => Left("Type␣error␣in␣function␣call.")
        }

        def operationAllowedOnType(op: String, typ: FLType) : Either[String,
            ↪ FLType] = (op,typ) match {
                case ("+"|"-"|"*"|"%"|"/"|"<"|"<="|"=="|"!=",FLInt|FLFloat) =>
                    ↪ Right(typ)
                case ("==",FLString) => Right(typ)
                case ("&",FLInt) => Right(typ)
                case ("=="|"!=",FLBoolean) => Right(typ)
                case (_,_) => Left("Operator␣" + op + "␣not␣allowed␣on␣type␣" +
                    ↪ typ + ".")
        }

}
```

The built-in functions of the Fula language

Listing B.15: Built-in functions. BuiltInPackage.scala file.

```
package frontend.typer;


import struct._;
import ast._;
```

```
object BuiltInPackage {



        val printInt: Symbol = Symbol("printFula",FLFunc(Seq(FLInt),FLIO(FLUnit
            ↪ )),Some("helper"))
        val printlnInt: Symbol = Symbol("printlnFula",FLFunc(Seq(FLInt),FLIO(
            ↪ FLUnit)),Some("helper"))
        val printFloat: Symbol = Symbol("printFula",FLFunc(Seq(FLFloat),FLIO(
            ↪ FLUnit)),Some("helper"))
        val printlnFloat: Symbol = Symbol("printlnFula",FLFunc(Seq(FLFloat),
            ↪ FLIO(FLUnit)),Some("helper"))
        val printBoolean: Symbol = Symbol("printFula",FLFunc(Seq(FLBoolean),
            ↪ FLIO(FLUnit)),Some("helper"))
        val printlnBoolean: Symbol = Symbol("printlnFula",FLFunc(Seq(FLBoolean)
            ↪ ,FLIO(FLUnit)),Some("helper"))
        val printString: Symbol = Symbol("printFula",FLFunc(Seq(FLString),FLIO(
            ↪ FLUnit)),Some("helper"))
        val printlnString: Symbol = Symbol("printlnFula",FLFunc(Seq(FLString),
            ↪ FLIO(FLUnit)),Some("helper"))
        val join: Symbol = Symbol("join",FLFunc(Seq(FLIO(FLUnit),FLIO(FLUnit)),
            ↪ FLIO(FLUnit)),Some("helper"))

        val fulaBuiltInPack = Seq(printInt,printlnInt,printFloat,printlnFloat,
            ↪ printBoolean,printlnBoolean,printString,printlnString,join)

}
```

Type Expression

Listing B.16: Type expression functions. ExprTyper.scala file.

```
package frontend.typer;


import struct._;
import struct.SymbolTable._;
import struct.Sym._;

import frontend.typer.TypeChecker._;

import ast.TypeAst._;
import ast.Ast._;
import ast.FLType._;
import ast._;

object ExprTyper {

       def typeVal(valExpr: Ast.Expr.Val, symT: SymbolTable) : Either[String,
           ↪ (TypeAst.TypeExpr.TyVal,SymbolTable)] = valExpr match {
               case Ast.Expr.Val(name, typ, expr) => for {
                       tyExpr <- typeExpr(expr,symT)
                       valType <- typeEqual(typ,getNodeType(tyExpr))
                       updatedType = FLType.wrapTypeToFunction(valType)
                       newSymT <- SymbolTable.putSymbol(name, updatedType, None
                           ↪ , symT)
               } yield (TypeAst.TypeExpr.TyVal(name,tyExpr,updatedType,FLUnit),
                   ↪ newSymT)
       }

       def typeIf(ifExpr: Ast.Expr.If, symT: SymbolTable): Either[String,
           ↪ TypeAst.TypeExpr.TyIf] = ifExpr match {
               case Ast.Expr.If(bexp,block1,block2) => for {
                       tyBexp <- typeBexp(bexp, symT)
                       tyBlock1 <- typeBlock(block1, openScope(symT))
                       tyBlock2 <- typeBlock(block2, openScope(symT))
                       block1Type = getBlockType(tyBlock1)
                       block2Type = getBlockType(tyBlock2)
                       ifType <- typeEqual(block1Type,block2Type)
               } yield (TypeAst.TypeExpr.TyIf(tyBexp,tyBlock1,tyBlock2,ifType))
```

```
}

def typeAssign(assignExpr: Ast.Expr.Assign, symT: SymbolTable): Either[
    ↪ String, TypeAst.TypeExpr] = assignExpr match {
        case Ast.Expr.Assign(name,args) => for {
                tyArgs <- typeBlock(args, symT)
                exprTypes = tyArgs.map(tyExpr => getNodeType(tyExpr))
                funcSym <- SymbolTable.getSymbol(name,Some(exprTypes),
                    ↪ symT)
                argsTypes = FLType.getFunctionArgTypes(funcSym.symTyp)
                _ <- typeEqualList(exprTypes,argsTypes)
                returnType = FLType.getFunctionReturnType(funcSym.symTyp
                    ↪ )
                pack = funcSym.pack match { case Some(str) => str case
                    ↪ None => "" }
                newTypeExpr = functionCallType(name,pack,tyArgs,funcSym.
                    ↪ symTyp,returnType)
        } yield (newTypeExpr)
}

def functionCallType(name: String, pack: String, args: Seq[TypeExpr],
    ↪ funcTyp: FLType, retType: FLType): TypeAst.TypeExpr = funcTyp
    ↪ match {
        case FLWrap(_) => TypeAst.TypeExpr.TyApply(name,args,retType)
        case _ => TypeAst.TypeExpr.TyAssign(name,pack,args,retType)
}

def typeValue(valueExpr: Ast.Expr.Value, symT: SymbolTable): Either[
    ↪ String, TypeAst.TypeExpr] = valueExpr match {
        case Ast.Expr.Value(name) => for {
                valType <- SymbolTable.getSymbol(name,None,symT)
                newValueExpr = specialValue(valType)
        } yield(newValueExpr)
```

```
}

def specialValue(valSym: Symbol): TypeAst.TypeExpr = valSym.symTyp
    ↪ match {
        case FLFunc(_,_) => {
                val pack = valSym.pack match { case Some(str) => str
                    ↪ case None => "" }
                TypeAst.TypeExpr.TyFunction(valSym.name, pack, valSym.
                    ↪ symTyp)
        }
        case _ => TypeAst.TypeExpr.TyValue(valSym.name,FLType.
                ↪ wrapTypeToFunction(valSym.symTyp))
}

def typeAop(aopExpr: Ast.Expr.Aop, symT: SymbolTable): Either[String,
    ↪ TypeAst.TypeExpr.TyAop] = aopExpr match {
        case Ast.Expr.Aop(op,aexp1,aexp2) => for {
                tyAexp1 <- typeExpr(aexp1, symT)
                tyAexp2 <- typeExpr(aexp2, symT)
                aopType <- typeEqual(getNodeType(tyAexp1),getNodeType(
                    ↪ tyAexp2))
                opType <- TypeChecker.operationAllowedOnType(op, aopType
                    ↪ )
        } yield (TypeAst.TypeExpr.TyAop(op,tyAexp1,tyAexp2,opType))
}

def typeExpr(expr: Ast.Expr, symT: SymbolTable) : Either[String,
    ↪ TypeAst.TypeExpr] = expr match {
        case expr: Ast.Expr.If => typeIf(expr,symT)
        case expr: Ast.Expr.Assign => typeAssign(expr,symT)
        case expr: Ast.Expr.Aop => typeAop(expr,symT)
        case expr: Ast.Expr.Value => typeValue(expr,symT)
        case expr: Ast.Expr.Bexp.Bop => typeBexp(expr,symT)
        case Ast.Expr.IntExpr(num) => Right(TypeAst.TypeExpr.TyIntExpr(
            ↪ num,FLInt))
        case Ast.Expr.FloatExpr(num) => Right(TypeAst.TypeExpr.
            ↪ TyFloatExpr(num,FLFloat))
        case Ast.Expr.BooleanExpr(bool) => Right(TypeAst.TypeExpr.
            ↪ TyBooleanExpr(bool,FLBoolean))
        case Ast.Expr.StringExpr(str) => Right(TypeAst.TypeExpr.
            ↪ TyStringExpr(str,FLString))
        case _ => {println(expr);Left("Syntax␣error.")}
}

def typeBexp(bexp: Ast.Expr.Bexp, symT: SymbolTable) : Either[String,
    ↪ TypeAst.TypeExpr.TypeBexp.TyBop] = bexp match {
        case Ast.Expr.Bexp.Bop(op,expr1,expr2) => for {
                tyExpr1 <- typeExpr(expr1,symT)
                tyExpr2 <- typeExpr(expr2,symT)
                bexpType <- typeEqual(getNodeType(tyExpr1),getNodeType(
                    ↪ tyExpr2))
                opType <- TypeChecker.operationAllowedOnType(op,
                    ↪ bexpType)
        } yield (TypeAst.TypeExpr.TypeBexp.TyBop(op,tyExpr1,tyExpr2,
            ↪ bexpType,FLBoolean))
}

def typeBlock(block: Ast.Block, symT: SymbolTable) : Either[String,
    ↪ TypeAst.TypeBlock] = block match {
        case (valExpr:Ast.Expr.Val)::xs => typeVal(valExpr, symT).
            ↪ flatMap{
                case (tyVal, newSymT) => for {
                        tyBlock <- typeBlock(xs,newSymT)
                } yield (tyVal+:tyBlock)
        }
        case expr::xs => for {
```

```
                    tyExpr <- typeExpr(expr, symT)
                    tyBlock <- typeBlock(xs, symT)
                } yield (tyExpr+:tyBlock)
                case Nil => Right(List())
        }
}
```

Type Declaration

Listing B.17: Type declaration functions. DeclTyper.scala file.

```
package frontend.typer;

import struct._;
import struct.SymbolTable._;
import struct.Sym._;

import frontend.typer.TypeChecker._;

import ast.TypeAst._;
import ast.Ast._;
import ast.FLType._;
import ast._;

object DeclTyper {

        val addArgDecls = (args: Seq[(String, FLType)], symT: SymbolTable) =>
            ↪ SymbolTable.putMultipleSymbol(Sym.argsToSymbols(args),symT)


        def typeFunctionBody(decl: Ast.Decl, symT: SymbolTable) : Either[String
            ↪ , TypeAst.TypeDecl] = decl match {
                case Ast.Decl.Def(name, pack, args, typ, body) => for {
                        argSymT <- addArgDecls(args,symT)
                        tyBlock <- ExprTyper.typeBlock(body,argSymT)
                typeOfBlock = getBlockType(tyBlock)
                declType <- typeEqual(typeOfBlock,typ)
            } yield (TypeAst.TypeDecl.TyDef(name, pack, args.map{case (name,
              ↪ typ) => (name, FLType.wrapTypeToFunction(typ))} , tyBlock
              ↪  , createFunctionType(args.map(_._2),declType) ))

            case Ast.Decl.Main(_,packMain,typ,body) => for {
                tyBlock <- ExprTyper.typeBlock(body,symT)
                typeOfBlock = getBlockType(tyBlock)
                mainType <- typeEqual(typeOfBlock,getFunctionReturnType(
                  ↪ typ))
            } yield (TypeAst.TypeDecl.TyMain(pack=packMain,body=tyBlock))
        }

        def typeFunctions(prog: Ast.Prog, symT: SymbolTable) : Either[String,
          ↪ TypeAst.TypeProg] = prog match {
            case decl::xs => for {
                func <- typeFunctionBody(decl,openScope(symT))
                funcs <- typeFunctions(xs,symT)
            } yield (func+:funcs)
            case Nil => Right(List())
        }
}
```

Type Program

Listing B.18: Type program functions. ProgTyper.scala file.

```
package frontend.typer;

import ast.Ast._;
import ast._;
import struct._;

object ProgTyper {
        val addFuncDecls = (nodes: Seq[Ast.Decl], symT: SymbolTable) =>
            ↪ SymbolTable.putMultipleSymbol(Sym.declToSymbols(nodes),symT)

        val addPackage = (symbols: Seq[Symbol], symT: SymbolTable) =>
            ↪ SymbolTable.putMultipleSymbol(symbols,symT)

        def typeProg(prog: Ast.Prog): Either[String, TypeAst.TypeProg] = for {
                        packSym <- addPackage(BuiltInPackage.fulaBuiltInPack,
                            ↪ Package(Map()))
                        val root: SymbolTable = Root(Map(),packSym)
                        rootSym <- addFuncDecls(prog, root)
                        typProg <- DeclTyper.typeFunctions(prog, rootSym)
                } yield (typProg)
}
```

B.4 Backend

B.4.1 Mangler

Name mapper data structure

Listing B.19: Name mapper data structure used for mangling names. NameMapper.scala file.

```
package backend.mangler;

case class NameMapper(table: Map[String, String], parent: Option[NameMapper],
    ↪ current: Int) {
        def getParent(): Option[NameMapper] = parent

        def getTable(): Map[String, String] = table

        def addName(name: String) : (NameMapper,String) = {
                val newName: String = "_" + current.toString
                val newRoot: Map[String, String] = table + (name -> newName)
                (NameMapper(newRoot, parent, current+1),newName)
        }

        def openScope() : NameMapper = {
                NameMapper(Map(), Some(this), current)
        }

        def closeScope() : Option[NameMapper] = {
                parent.map(p => NameMapper(p.getTable, p.getParent, current))
        }

        def getName(name: String) : Option[String] = getFromMap(name) match {
                case Some(str) => Some(str)
                case None => parent match {
                        case Some(mapper) => mapper.getName(name)
                        case None => None
                }
        }

        def getFromMap(name: String) : Option[String] = {
                try { Some(table(name)) }
                catch { case e:Exception => None }
        }
}
```

Mangle Expression

Listing B.20: Mangle expression functions. MangleExpr.scala file.

```
package backend.mangler;

import ast.TypeAst._;
import ast.FLType._;
import ast._;

object MangleExpr {
```

```
def mangleIf(expr: TypeAst.TypeExpr.TyIf, nameMap: NameMapper) : (
    ↪ TypeAst.TypeExpr.TyIf,NameMapper) = expr match {
        case TypeAst.TypeExpr.TyIf(bexp, b1, b2, typ) => {
                val (dBexp,m0) = mangleBexp(bexp,nameMap)
                val (dB1,m1) = mangleBlock(b1,m0)
                val (dB2,m2) = mangleBlock(b2,m1)
                (TypeAst.TypeExpr.TyIf(dBexp, dB1, dB2,typ),m2)
        }
}

def mangleAssign(expr: TypeAst.TypeExpr.TyAssign, nameMap: NameMapper)
    ↪ : (TypeAst.TypeExpr.TyAssign,NameMapper) = expr match {
        case TypeAst.TypeExpr.TyAssign(name, pack, args, typ) => {
                val (dArgs,m0) = mangleExprList(args,nameMap)
                (TypeAst.TypeExpr.TyAssign(name,pack,dArgs,typ),m0)
        }
}

def mangleApply(expr: TypeAst.TypeExpr.TyApply, nameMap: NameMapper) :
    ↪ (TypeAst.TypeExpr.TyApply,NameMapper) = expr match {
        case TypeAst.TypeExpr.TyApply(name, args, typ) => {
                val dName: String = nameMap.getName(name) match {
                        case Some(str_new) => str_new
                        case None => name
                }
                val (dArgs,m0) = mangleExprList(args,nameMap)
                (TypeAst.TypeExpr.TyApply(dName,dArgs,typ),m0)
        }
}

def mangleAop(expr: TypeAst.TypeExpr.TyAop, nameMap: NameMapper) : (
    ↪ TypeAst.TypeExpr.TyAop,NameMapper) = expr match {
        case TypeAst.TypeExpr.TyAop(op, aexp1, aexp2, typ) => {
                val (dAexp1,m0) = mangleExpr(aexp1,nameMap)
                val (dAexp2,m1) = mangleExpr(aexp2,m0)
                (TypeAst.TypeExpr.TyAop(op,dAexp1,dAexp2,typ),m1)
        }
}

def mangleValue(expr: TypeAst.TypeExpr.TyValue, nameMap: NameMapper) :
    ↪ (TypeAst.TypeExpr.TyValue,NameMapper) = expr match {
        case TypeAst.TypeExpr.TyValue(str, typ) => {
                val dStr: String = nameMap.getName(str) match {
                        case Some(str_new) => str_new
                        case None => str
                }
                (TypeAst.TypeExpr.TyValue(dStr,typ),nameMap)
        }
}

def mangleVal(expr: TypeAst.TypeExpr.TyVal, nameMap: NameMapper) : (
    ↪ TypeAst.TypeExpr.TyVal,NameMapper) = expr match {
        case TypeAst.TypeExpr.TyVal(name, expr, typExpr, typ) => {
                val (m0,dName) = nameMap.addName(name)
                val (dExpr,m1) = mangleExpr(expr,m0)
                (TypeAst.TypeExpr.TyVal(dName, dExpr,typExpr, typ),m1)
        }
}

def mangleBexp(bexp: TypeAst.TypeExpr.TypeBexp, nameMap: NameMapper) :
    ↪ (TypeAst.TypeExpr.TypeBexp.TyBop,NameMapper) = bexp match {
        case TypeAst.TypeExpr.TypeBexp.TyBop(op, aexp1, aexp2, exprTyp,
            ↪ typ) =>
                val (dAexp1,m0) = mangleExpr(aexp1,nameMap)
                val (dAexp2,m1) = mangleExpr(aexp2,m0)
                (TypeAst.TypeExpr.TypeBexp.TyBop(op, dAexp1, dAexp2,
                    ↪ exprTyp,typ),m1)
}



def mangleExpr(expr: TypeAst.TypeExpr, nameMap: NameMapper) : (TypeAst.
    ↪ TypeExpr,NameMapper) = expr match {
        case expr: TypeAst.TypeExpr.TyIf => mangleIf(expr,nameMap)
        case expr: TypeAst.TypeExpr.TyAssign => mangleAssign(expr,
            ↪ nameMap)
        case expr: TypeAst.TypeExpr.TyApply => mangleApply(expr,nameMap)
        case expr: TypeAst.TypeExpr.TyAop => mangleAop(expr,nameMap)
        case expr: TypeAst.TypeExpr.TyValue => mangleValue(expr,nameMap)
        case expr: TypeAst.TypeExpr.TyVal => mangleVal(expr,nameMap)
        case expr: TypeAst.TypeExpr.TypeBexp.TyBop => mangleBexp(expr,
            ↪ nameMap)
        case _ => (expr,nameMap)
}



def mangleExprList(block: TypeAst.TypeBlock, nameMap: NameMapper) : (
    ↪ TypeAst.TypeBlock,NameMapper) = block match {
        case expr::xs => {
                val (mangleedExpr,exprMap) = mangleExpr(expr,nameMap)
                val recuCall = mangleExprList(xs,exprMap)
                (mangleedExpr+:recuCall._1,recuCall._2)
        }
        case Nil => (List(),nameMap)
}

def mangleBlock(block: TypeAst.TypeBlock, nameMap: NameMapper) : (
    ↪ TypeAst.TypeBlock,NameMapper) = {
        val blockMap = nameMap.openScope()
```

```
            val (mangleedBlock, blockMap2) = mangleExprList(block, blockMap)
            (mangleedBlock,blockMap2.closeScope().get)
        }

        def mangleDeclBody(block: TypeAst.TypeBlock, nameMap: NameMapper) :
            ↪ TypeAst.TypeBlock = {
            mangleBlock(block,nameMap)._1
        }
}
```

Mangle Declaration

Listing B.21: Mangle declaration functions. MangleDecl.scala file.

```
package backend.mangler;

import ast.TypeAst._;
import ast.FLType._;
import ast._;

import NameMapper._;

object MangleDecl {

        def mangleDeclArgs(args: Seq[(String,FLType)], nameMap: NameMapper): (
            ↪ List[(String,FLType)],NameMapper) = args match {
            case arg::xs => {
                    val (nameMap2, argName) = nameMap.addName(arg._1)
                    val argEntry: (String,FLType) = (argName,arg._2)
                    val recurCall = mangleDeclArgs(xs,nameMap2)
                    (argEntry+:recurCall._1,recurCall._2)
            }
            case Nil => (List(), nameMap)
        }
        def mangleDecl(decl: TypeAst.TypeDecl) : TypeAst.TypeDecl = decl match
            ↪ {
            case TypeAst.TypeDecl.TyDef(name, pack, args, body, typed) => {
                    val nameMap = NameMapper(Map(), None, 0)
                    val (mangleedArgs, nameMap2) = mangleDeclArgs(args,
                        ↪ nameMap)
                    val mangleedBody: TypeAst.TypeBlock = MangleExpr.
                        ↪ mangleDeclBody(body, nameMap2)
                    TypeAst.TypeDecl.TyDef(name, pack, mangleedArgs,
                        ↪ mangleedBody, typed)
            }
            case TypeAst.TypeDecl.TyMain(name, pack, body, typed) => {
                    val nameMap = NameMapper(Map(), None, 0)
                    val mangleedBody: TypeAst.TypeBlock = MangleExpr.
                        ↪ mangleDeclBody(body, nameMap)
                    TypeAst.TypeDecl.TyMain(name, pack, mangleedBody, typed)
            }
        }
}
```

Mangle Program

Listing B.22: Mangle program functions. MangleProg.scala file.

```
package backend.mangler;


import ast.TypeAst._;
import ast.FLType._;
import ast._;


object MangleProg {


        def mangleProg(prog: TypeAst.TypeProg) : TypeAst.TypeProg = prog.map(
            ↪ decl => MangleDecl.mangleDecl(decl))
}
```

B.4.2 Lifter

Lift Expression

Listing B.23: Lift expression functions. LiftExpr.scala file.

```
package backend.lifter;


import ast.TypeAst._;

import ast.FLType._;

import ast._;


object LiftExpr {


       def liftExpr(expr: TypeAst.TypeExpr): Set[FunctionBundle] = expr match
           ↪ {
              case TypeAst.TypeExpr.TyIf(b,e1,e2,_) => liftExpr(b)++liftBody(
                  ↪ e1)++liftBody(e2)
              case TypeAst.TypeExpr.TyAssign(_,_,args,_) => liftBody(args)
              case TypeAst.TypeExpr.TyApply(_,args,_) => liftBody(args)
              case TypeAst.TypeExpr.TyAop(_,a1,a2,_) => liftExpr(a1)++liftExpr
                  ↪ (a2)
              case TypeAst.TypeExpr.TypeBexp.TyBop(_,a1,a2,_,_) => liftExpr(a1
                  ↪ )++liftExpr(a2)
              case TypeAst.TypeExpr.TyVal(_,e,_,_) => liftExpr(e)
              case TypeAst.TypeExpr.TyFunction(name,pack,typ) => Set(
                  ↪ FunctionBundle(name,pack,typ))
              case _ => Set()
       }


       def liftBody(body: TypeAst.TypeBlock, acc: Set[FunctionBundle]=Set()):
           ↪ Set[FunctionBundle] = body match {
              case expr::xs => liftBody(xs, LiftExpr.liftExpr(expr)++acc)
              case Nil => acc
```

```
        }

}
```

Lift Declaration

Listing B.24: Lift declaration functions. LiftDecl.scala file.

```
package backend.lifter;


import ast.TypeAst._;

import ast.FLType._;

import ast._;


object LiftDecl {


        def liftDecl(decl: TypeAst.TypeDecl): Set[FunctionBundle] = decl match
            ↪ {
                case TypeAst.TypeDecl.TyDef(_,_,_,body,_) => LiftExpr.liftBody(
                    ↪ body)
                case TypeAst.TypeDecl.TyMain(_,_,body,_) => LiftExpr.liftBody(
                    ↪ body)
        }

}
```

Lift Program

Listing B.25: Lift program functions. LiftProg.scala file.

```
package backend.lifter;


import ast.TypeAst._;

import ast.FLType._;

import ast._;


import org.objectweb.asm.Handle;
import org.objectweb.asm.Opcodes._;

case class FunctionBundle(name: String, pack: String, typ: FLType)

object LiftProg {

    def liftProg(prog: TypeAst.TypeProg) : TypeAst.TypeProg = {
        val liftDecls: Set[FunctionBundle] = getLiftedFunctions(prog)
        val adaptedDecls: Set[TypeAst.TypeDecl.TyDef] = liftDecls.map(fb
            ↪  => getAdaptedDecl(fb))
        val liftedProg: TypeAst.TypeProg = adaptedDecls.toSeq++:prog
        liftedProg
    }

    def getLiftedFunctions(prog: TypeAst.TypeProg, acc: Set[FunctionBundle]
        ↪ =Set()): Set[FunctionBundle] = prog match {
        case decl::xs => getLiftedFunctions(xs,LiftDecl.liftDecl(decl)++
            ↪ acc)
        case Nil => acc
    }

    def getAdaptedDecl(fb: FunctionBundle) : TypeAst.TypeDecl.TyDef = {

        val funcArgsType: Seq[FLType] = getFunctionArgTypes(fb.typ)
        val returnType: FLType = getFunctionReturnType(fb.typ)
        val modifiedArgs: Seq[(String,FLType)] = modifyArgsType(
            ↪ funcArgsType)
        TypeAst.TypeDecl.TyDef(fb.name+"$adapted",fb.pack,modifiedArgs,
            ↪ Seq(createBody(fb)),FLFunc(modifiedArgs.map(tup => tup._2
            ↪ ),modifyReturnType(returnType)))
    }

    def modifyReturnType(typ: FLType) : FLType = typ match {
        case FLInt|FLBoolean|FLFloat => FLObject
        case _ => typ
}

def modifyArgsType(types: Seq[FLType], name: Int=0): Seq[(String,FLType
    ↪ )] = types match {
        case FLInt::xs => (("_"+name),FLObject) +: modifyArgsType(xs,
            ↪ name+1)
        case FLFloat::xs => (("_"+name),FLObject) +: modifyArgsType(xs,
            ↪ name+1)
        case FLBoolean::xs => (("_"+name),FLObject) +: modifyArgsType(xs
            ↪ ,name+1)
        case FLString::xs => (("_"+name),FLString) +: modifyArgsType(xs,
            ↪ name+1)
        case FLIO(FLUnit)::xs => (("_"+name),FLIO(FLUnit)) +:
            ↪ modifyArgsType(xs,name+1)
        case _::xs => modifyArgsType(xs,name)
        case Nil => Seq()
}

def createBody(fb: FunctionBundle): TypeExpr = {
        val argsType: Seq[FLType] = FLType.getFunctionArgTypes(fb.typ)
        val returnType: FLType = FLType.getFunctionReturnType(fb.typ)

        val argsFunctionCall: Seq[TypeExpr] = processUnboxing(argsType)
        val functionCall: TypeExpr = processFunctionCall(fb.name, fb.
            ↪ pack, argsFunctionCall,returnType)
        processBoxing(functionCall,returnType)
}

def processUnboxing(types: Seq[FLType], name: Int=0): Seq[TypeExpr] =
    ↪ types match {
        case FLInt::xs => TypeAst.TypeExpr.TyAssign("unboxToInt","scala/
            ↪ runtime/BoxesRunTime",Seq(TypeAst.TypeExpr.TyValue("_"+
            ↪ name,FLObject)),FLInt) +: processUnboxing(xs,name+1)
        case FLFloat::xs => TypeAst.TypeExpr.TyAssign("unboxToFloat","
            ↪ scala/runtime/BoxesRunTime",Seq(TypeAst.TypeExpr.TyValue(
            ↪ "_"+name,FLObject)),FLFloat) +: processUnboxing(xs,name
            ↪ +1)
        case FLBoolean::xs => TypeAst.TypeExpr.TyAssign("unboxToBoolean"
            ↪ ,"scala/runtime/BoxesRunTime",Seq(TypeAst.TypeExpr.
            ↪ TyValue("_"+name,FLObject)),FLBoolean) +: processUnboxing
            ↪ (xs,name+1)
        case FLString::xs => TypeAst.TypeExpr.TyValue("_"+name,FLString)
            ↪  +: processUnboxing(xs,name+1)
        case FLIO(FLUnit)::xs => TypeAst.TypeExpr.TyValue("_"+name,FLIO(
            ↪ FLUnit)) +: processUnboxing(xs,name+1)
        case _::xs => processUnboxing(xs,name)
        case Nil => Seq()
}

def processFunctionCall(name: String, pack: String, args: Seq[TypeExpr
    ↪ ], typ: FLType): TypeExpr = {
        TypeAst.TypeExpr.TyAssign(name,pack,args,typ)
}

def processBoxing(funcCall: TypeExpr, returnType: FLType): TypeExpr =
    ↪ returnType match {
        case FLInt => TypeAst.TypeExpr.TyAssign("boxToInteger","scala/
            ↪ runtime/BoxesRunTime",Seq(funcCall),FLIntObject)
        case FLFloat => TypeAst.TypeExpr.TyAssign("boxToFloat","scala/
            ↪ runtime/BoxesRunTime",Seq(funcCall),FLFloatObject)
        case FLBoolean => TypeAst.TypeExpr.TyAssign("boxToBoolean","
            ↪ scala/runtime/BoxesRunTime",Seq(funcCall),FLBooleanObject
            ↪ )
        case _ => funcCall
```

```
        }


}
```

B.4.3 ByteCode Generation

Bytecode Generation helper functions

Listing B.26: Bytecode Generation helper functions. CodeGenHelper.scala file.

```
package backend.codegen;


import org.objectweb.asm._;

import org.objectweb.asm.ClassWriter._;

import org.objectweb.asm.Opcodes._;


import ast._;

import ast.TypeAst._;


object CodeGenHelper {


        def getFunctionDescriptor(typ: FLType): String = typ match {

                case FLInt|FLFloat|FLBoolean => "Ljava/lang/Object;"

                case FLIntObject => "Ljava/lang/Integer;"

                case FLFloatObject => "Ljava/lang/Float;"

                case FLBooleanObject => "Ljava/lang/Boolean;"

                case FLString => "Ljava/lang/String;"

                case FLIO(_) => "Lfula/helper$IO;"

                case FLObject => "Ljava/lang/Object;"

                case FLFunc(args,typ) => "(" + args.map(getFunctionDescriptor(_)
                    ↪ ).mkString + ")" + getFunctionDescriptor(typ)

                case _ => ""

        }


        def getSamDescriptor(typ: FLType): String = typ match {
                case FLFunc(args,typ) => "(" + args.map(getSamDescriptor(_)).
                    ↪ mkString + ")" + getSamDescriptor(typ)
                case _ => "Ljava/lang/Object;"
        }

        def getFunctionArity(typ: FLType): String = typ match {
                case FLFunc(args,_) => "()Lscala/Function" + args.length + ";"
                case _ => ""
        }

        def callLambdaArity(typ: FLType): String = typ match {
                case FLFunc(args,_) => "scala/Function" + args.length
                case _ => ""
        }

}
```

Compile array of bytes to file

Listing B.27: Compile array of bytes to file. CompileToFile.scala file.

```
package backend.codegen;

import cats.effect._;

import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

object CompileToFile {

        def writeByteCode(bytes: Array[Byte], filename: String) : IO[Either[
            ↪ String,Unit]] = IO {
                try {
                        val pth: Path = Paths.get("fula/" + filename + ".class")
```

```
                    ↪ ;
                val direct: Path = Paths.get("fula/");
                Files.createDirectories(direct);
                Right(Files.write(pth, bytes));
        } catch {case e: Exception => Left("Error:␣cannot␣write␣file.")}
    }

    def writeByteCode2(bytes: Array[Byte], filename: String) : IO[Either[
        ↪ Unit,Unit]] = IO {
        try {
                val pth: Path = Paths.get("fula/" + filename + ".class")
                    ↪ ;
                Right(Files.write(pth, bytes));
        } catch {case e: Exception => Left(e.printStackTrace())}
    }

}
```

Compile Expression to bytecode

Listing B.28: Compile expression functions. CompileExpr.scala file.

```
package backend.codegen;


import ast._;
import ast.FLType._;


import ast.TypeAst._;
import ast.TypeAst;


import org.objectweb.asm._;
import org.objectweb.asm.ClassWriter;
import org.objectweb.asm.ClassWriter._;
import org.objectweb.asm.Opcodes._;
import org.objectweb.asm.Type;
import cats.data.State;

object CompileExpr {

        val MODULE_NAME = "fula/"

        val nameToAddress = (name: String) => name.drop(1).toInt

        def compileIf(expr: TypeAst.TypeExpr.TyIf, mv: MethodVisitor) :
            ↪ MethodVisitor = expr match {
                case TypeAst.TypeExpr.TyIf(bexp, b1, b2, typ) => {
                        val mv_mod1 = compileBexp(bexp,mv)
                        val labelFalse = new Label();
                        val labelEnd = new Label();
                        mv.visitJumpInsn(IFEQ, labelFalse);
                        val mv_mod2 = compileBody(b1,mv_mod1);
                        mv_mod2.visitJumpInsn(GOTO, labelEnd);
                        mv.visitLabel(labelFalse);
                        val mv_mod3 = compileBody(b2,mv_mod2);
                        mv_mod3.visitLabel(labelEnd);
                        mv_mod3
                }
        }

        def compileBexpOperator(op: String, typ: FLType, mv: MethodVisitor):
            ↪ MethodVisitor = {

                val labelFalse = new Label();
                val labelEnd = new Label();

                typ match {
                        case FLInt|FLBoolean => mv.visitInsn(ISUB);
                        case FLFloat => mv.visitInsn(FCMPL);
                        case FLString => mv.visitMethodInsn(INVOKEVIRTUAL, "java
                            ↪ /lang/Object", "equals", "(Ljava/lang/Object;)Z",
                            ↪ false); mv.visitLdcInsn(1); mv.visitInsn(IXOR);
                }

                op match {
                        case "==" => mv.visitJumpInsn(IFNE, labelFalse)
                        case "!=" => mv.visitJumpInsn(IFEQ, labelFalse)
                        case "<" => mv.visitJumpInsn(IFGE, labelFalse)
                        case "<=" => mv.visitJumpInsn(IFGT, labelFalse)
                }

                mv.visitLdcInsn(1);
                mv.visitJumpInsn(GOTO, labelEnd);
                mv.visitLabel(labelFalse);
                mv.visitLdcInsn(0);
                mv.visitLabel(labelEnd);

                mv;
        }

        def compileBexp(bexp: TypeAst.TypeExpr.TypeBexp, mv: MethodVisitor) :
            ↪ MethodVisitor = bexp match {
                case TypeAst.TypeExpr.TypeBexp.TyBop(op, aexp1, aexp2, exprTyp,
                    ↪ typ) =>
                        val mv_mod1 = compileExpr(aexp1,mv)
                        val mv_mod2 = compileExpr(aexp2,mv_mod1)
                        val mv_mod3 = compileBexpOperator(op,exprTyp, mv_mod2)
                        mv_mod3
        }

        def compileAexpOperator(op: String, typ: FLType, mv: MethodVisitor):
            ↪ MethodVisitor = (op,typ) match {
                case ("+",FLInt) => {mv.visitInsn(IADD);mv}
                case ("-",FLInt) => {mv.visitInsn(ISUB);mv}
                case ("/",FLInt) => {mv.visitInsn(IDIV);mv}
                case ("%",FLInt) => {mv.visitInsn(IREM);mv}
                case ("*",FLInt) => {mv.visitInsn(IMUL);mv}
                case ("&",FLInt) => {mv.visitInsn(IAND);mv}
                case ("+",FLFloat) => {mv.visitInsn(FADD);mv}
                case ("-",FLFloat) => {mv.visitInsn(FSUB);mv}
                case ("/",FLFloat) => {mv.visitInsn(FDIV);mv}
                case ("%",FLFloat) => {mv.visitInsn(FREM);mv}
                case ("*",FLFloat) => {mv.visitInsn(FMUL);mv}
                case (_,_) => mv
        }

        def compileAop(expr: TypeAst.TypeExpr.TyAop, mv: MethodVisitor) :
            ↪ MethodVisitor = expr match {
                case TypeAst.TypeExpr.TyAop(op, aexp1, aexp2, typ) => {
                        val mv_mod1 = compileExpr(aexp1,mv)
                        val mv_mod2 = compileExpr(aexp2,mv_mod1)
                        val mv_mod3 = compileAexpOperator(op,typ, mv_mod2)
                        mv_mod3
                }
        }


        def compileAssign(expr: TypeAst.TypeExpr.TyAssign, mv: MethodVisitor) :
            ↪  MethodVisitor = expr match {
                case TypeAst.TypeExpr.TyAssign(name, pack, args, typ) => {
                        val mv_mod = compileBody(args,mv)
                        val pack_name: String = if (pack.startsWith("scala/"))
                            ↪ pack else MODULE_NAME+pack
                        mv.visitMethodInsn(INVOKESTATIC, pack_name, name,
                            ↪ CompileDecl.funcTypToString( FLFunc(args.map(expr
                            ↪ => modifyArg(getNodeType(expr))),typ) ), false);
                        mv;
                }
        }

        def modifyArg(typ: FLType): FLType = typ match {
                case FLFunc(_,_) => FLWrap(typ)
                case _ => typ
        }

        def compileApply(expr: TypeAst.TypeExpr.TyApply, mv: MethodVisitor) :
            ↪ MethodVisitor = expr match {
                case TypeAst.TypeExpr.TyApply(name, args, typ) => {
                        mv.visitVarInsn(ALOAD,nameToAddress(name))
                        val mv_mod = compileApplyArgs(args,mv)
                        mv.visitMethodInsn(INVOKEINTERFACE, CodeGenHelper.
                            ↪ callLambdaArity(FLFunc(args.map(expr =>
                            ↪ getNodeType(expr)),typ)), "apply", CodeGenHelper.
                            ↪ getSamDescriptor(FLFunc(args.map(expr =>
                            ↪ getNodeType(expr)),typ)), true);
                        unboxReturn(typ,mv)
                        mv;
                }
        }

        def compileApplyArgs(args: TypeAst.TypeBlock, mv: MethodVisitor) :
            ↪ MethodVisitor = args match {
                case expr::xs => {
                        val mv_mod = compileExpr(expr, mv)
                        val mv_mod2 = boxArg(getNodeType(expr),mv_mod)
                        compileApplyArgs(xs,mv_mod2)
                }
                case Nil => mv
        }

        def boxArg(typ: FLType, mv: MethodVisitor): MethodVisitor = typ match {
                case FLInt => { mv.visitMethodInsn(INVOKESTATIC, "scala/runtime/
                    ↪ BoxesRunTime", "boxToInteger", CompileDecl.
                    ↪ funcTypToString(FLFunc(Seq(FLInt),FLIntObject)), false);
                    ↪ mv }
                case FLFloat => { mv.visitMethodInsn(INVOKESTATIC, "scala/
                    ↪ runtime/BoxesRunTime", "boxToFloat", CompileDecl.
                    ↪ funcTypToString(FLFunc(Seq(FLFloat),FLFloatObject)),
                    ↪ false); mv }
                case FLBoolean => { mv.visitMethodInsn(INVOKESTATIC, "scala/
                    ↪ runtime/BoxesRunTime", "boxToBoolean", CompileDecl.
                    ↪ funcTypToString(FLFunc(Seq(FLBoolean),FLBooleanObject)),
                    ↪ false); mv }
                case _ => mv
        }

        def unboxReturn(typ: FLType, mv: MethodVisitor): MethodVisitor = typ
            ↪ match {
                case FLInt => { mv.visitMethodInsn(INVOKESTATIC, "scala/runtime/
                    ↪ BoxesRunTime", "unboxToInt", CompileDecl.funcTypToString(
                    ↪ FLFunc(Seq(FLObject),FLInt)), false); mv }
                case FLFloat => { mv.visitMethodInsn(INVOKESTATIC, "scala/
                    ↪ runtime/BoxesRunTime", "unboxToFloat", CompileDecl.
                    ↪ funcTypToString(FLFunc(Seq(FLObject),FLFloat)), false);
                    ↪ mv }
                case FLBoolean => { mv.visitMethodInsn(INVOKESTATIC, "scala/
                    ↪ runtime/BoxesRunTime", "unboxToBoolean", CompileDecl.
                    ↪ funcTypToString(FLFunc(Seq(FLObject),FLBoolean)), false);
                    ↪  mv }
                case FLString => { mv.visitTypeInsn(CHECKCAST,"java/lang/String"
                ↪ ); mv }
              case FLIO(FLUnit) => { mv.visitTypeInsn(CHECKCAST,"fula/
                ↪ helper$IO"); mv }
              case _ => mv
        }

        def compileFunction(expr: TypeAst.TypeExpr.TyFunction, mv:
            ↪ MethodVisitor) : MethodVisitor = expr match {
              case TypeAst.TypeExpr.TyFunction(name, pack, typ) => {

                    val hdl1: Handle = new Handle(Opcodes.H_INVOKESTATIC,"
                        ↪ java/lang/invoke/LambdaMetafactory","
                        ↪ altMetafactory", "(Ljava/lang/invoke/
                        ↪ MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/
                        ↪ invoke/MethodType;[Ljava/lang/Object;)Ljava/lang/
                        ↪ invoke/CallSite;",false)

                    val lambdaDescriptor: String = CodeGenHelper.
                        ↪ getFunctionDescriptor(typ)

                    val hdl2: Handle = new Handle(Opcodes.H_INVOKESTATIC,
                        ↪ MODULE_NAME+pack,name+"$adapted", lambdaDescriptor
                        ↪ ,false)

                    val samSignature = Type.getMethodType(CodeGenHelper.
                        ↪ getSamDescriptor(typ))
                    val instSignature = Type.getMethodType(lambdaDescriptor)

                    val functionArity = CodeGenHelper.getFunctionArity(typ)

                    mv.visitInvokeDynamicInsn("apply",functionArity,hdl1,
                        ↪ samSignature,hdl2,instSignature,1);
                    mv;
        }
}

def compileValue(expr: TypeAst.TypeExpr.TyValue, mv: MethodVisitor) :
    ↪ MethodVisitor = expr match {
        case TypeAst.TypeExpr.TyValue(str, typ) => typ match {
                case FLInt|FLBoolean => mv.visitVarInsn(ILOAD,
                    ↪ nameToAddress(str))
                case FLFloat => mv.visitVarInsn(FLOAD,nameToAddress(str)
                    ↪ )
                case FLIO(_)|FLString|FLWrap(_)|FLObject => mv.
                    ↪ visitVarInsn(ALOAD,nameToAddress(str))
        }
        mv;
}

def compileVal(expr: TypeAst.TypeExpr.TyVal, mv: MethodVisitor) :
    ↪ MethodVisitor = expr match {
        case TypeAst.TypeExpr.TyVal(name, expr, exprTyp, typ) => exprTyp
            ↪  match {
                case FLInt|FLBoolean => {
                        val mv_mod = compileExpr(expr,mv)
                        mv_mod.visitVarInsn(ISTORE,nameToAddress(name))
                        mv_mod
                }
                case FLFloat => {
                        val mv_mod = compileExpr(expr,mv)
                        mv_mod.visitVarInsn(FSTORE,nameToAddress(name))
                        mv_mod
                }
                case FLIO(_)|FLString|FLWrap(_)|FLObject => {
                        val mv_mod = compileExpr(expr,mv)
                        mv_mod.visitVarInsn(ASTORE,nameToAddress(name))
                    mv_mod
                }
        }
}

def compileExpr(expr: TypeAst.TypeExpr, mv: MethodVisitor) :
    ↪ MethodVisitor = expr match {
        case expr: TypeAst.TypeExpr.TyIf => compileIf(expr,mv)
        case expr: TypeAst.TypeExpr.TyAssign => compileAssign(expr,mv)
        case expr: TypeAst.TypeExpr.TyApply => compileApply(expr,mv)
        case expr: TypeAst.TypeExpr.TyFunction => compileFunction(expr,
            ↪ mv)
        case expr: TypeAst.TypeExpr.TyAop => compileAop(expr,mv)
        case expr: TypeAst.TypeExpr.TyValue => compileValue(expr,mv)
        case expr: TypeAst.TypeExpr.TyVal => compileVal(expr,mv)
        case expr: TypeAst.TypeExpr.TypeBexp.TyBop => compileBexp(expr,
            ↪ mv)
        case TypeAst.TypeExpr.TyFloatExpr(num,typ) => { mv.visitLdcInsn(
            ↪ num); mv}
        case TypeAst.TypeExpr.TyIntExpr(num,typ) => { mv.visitLdcInsn(
            ↪ num); mv}
        case TypeAst.TypeExpr.TyBooleanExpr(bool,typ) => { if (bool) mv.
            ↪ visitLdcInsn(1) else mv.visitLdcInsn(0); mv}
        case TypeAst.TypeExpr.TyStringExpr(str,typ) => { mv.visitLdcInsn
            ↪ (str); mv}
}

def compileBody(block: TypeAst.TypeBlock, mv: MethodVisitor) :
    ↪ MethodVisitor = block match {
        case expr::xs => {
                val mv_mod = compileExpr(expr, mv)
                compileBody(xs,mv_mod)
        }
```

```
            case Nil => mv
        }
}
```

Compile Declaration to bytecode

Listing B.29: Compile declaration functions. CompileDecl.scala file.

```
package backend.codegen;


import cats.data.State;


import org.objectweb.asm._;
import org.objectweb.asm.ClassWriter._;
import org.objectweb.asm.Opcodes._;


import ast._;
import ast.TypeAst._;



object CompileDecl {


        def funcTypToString(typ: FLType): String = typ match {
                case FLInt => "I"
                case FLIntObject => "Ljava/lang/Integer;"
                case FLFloat => "F"
                case FLFloatObject => "Ljava/lang/Float;"
                case FLUnit => "V"
                case FLBoolean => "Z"
                case FLBooleanObject => "Ljava/lang/Boolean;"
                case FLString => "Ljava/lang/String;"
                case FLIO(_) => "Lfula/helper$IO;"
                case FLObject => "Ljava/lang/Object;"
                case FLWrap(FLFunc(args,typ)) => "Lscala/Function" + args.length
                    ↪ + ";"
        case FLFunc(args,typ) => "(" + args.map(funcTypToString(_)).
            ↪ mkString + ")" + funcTypToString(typ)
}

def compileReturn(typ: FLType) : Int = typ match {
        case FLInt => IRETURN
        case FLBoolean => IRETURN
        case FLFloat => FRETURN
        case FLUnit => RETURN
        case FLIO(_)|FLObject|FLWrap(_)|FLString => ARETURN
        case FLFunc(_,funcReturnTyp) => compileReturn(funcReturnTyp)
}

def funcSignature(typ: FLType) : String = typ match {
        case FLInt => "I"
        case FLIntObject => "Ljava/lang/Integer;"
        case FLFloat => "F"
        case FLFloatObject => "Ljava/lang/Float;"
        case FLUnit => "V"
        case FLBoolean => "Z"
        case FLBooleanObject => "Ljava/lang/Boolean;"
        case FLString => "Ljava/lang/String;"
        case FLIO(_) => "Lfula/helper$IO;"
        case FLObject => "Ljava/lang/Object;"
        case FLWrap(FLFunc(args,typ)) => "Lscala/Function" + args.length
            ↪  + "<" + args.map(wrapSignature(_)).mkString +
            ↪ wrapSignature(typ) + ">" + ";"
        case FLFunc(args,typ) => "(" + args.map(funcSignature(_)).
            ↪ mkString + ")" + funcSignature(typ)
}

def wrapSignature(typ: FLType) : String = typ match {
        case FLInt|FLFloat|FLBoolean => "Ljava/lang/Object;"
        case _ => funcSignature(typ)
}



def compileDecl(decl: TypeAst.TypeDecl, cw: ClassWriter): ClassWriter =
    ↪ decl match {
        case TypeAst.TypeDecl.TyDef(name,_, args, body, typ) => {
                val mv: MethodVisitor = cw.visitMethod(ACC_PUBLIC +
                    ↪ ACC_STATIC, name, funcTypToString(typ),
                    ↪ funcSignature(typ), null)
                mv.visitCode();
                val mv_mod = CompileExpr.compileBody(body, mv);
                mv_mod.visitInsn(compileReturn(typ));
                mv.visitMaxs(0, 0);
                mv_mod.visitEnd();
                cw;
        }
        case TypeAst.TypeDecl.TyMain(name, _, body, typ) => {
                val mv: MethodVisitor = cw.visitMethod(ACC_PUBLIC +
                    ↪ ACC_STATIC, name, "([Ljava/lang/String;)V", null,
                    ↪ null)
                mv.visitCode();
                val mv_mod = CompileExpr.compileBody(body, mv);
                mv.visitMethodInsn(INVOKEINTERFACE, "fula/helper$IO", "
                    ↪ unsafeRunSync", "()V", true);
                mv.visitInsn(RETURN);
                mv.visitMaxs(0, 0);
                mv_mod.visitEnd();
                cw;
        }

}
```

```
}
```

Compile Program to bytecode

Listing B.30: Compile program functions. CompileProg.scala file.

```
package backend.codegen;


import cats.data.State;


import org.objectweb.asm._;
import org.objectweb.asm.ClassWriter._;
import org.objectweb.asm.Opcodes._;


import ast._;
import ast.TypeAst._;



object CompileProg {

       def compileProg(prog: TypeAst.TypeProg, filename: String) : Array[Byte]
           ↪  = {

              val cw: ClassWriter = new ClassWriter(COMPUTE_FRAMES +
                  ↪ COMPUTE_MAXS);
              cw.visit(V1_8, ACC_PUBLIC + ACC_SUPER, "fula/"+filename,null, "
                  ↪ java/lang/Object", null);
              val cw_mod = compileDecls(prog,cw)
              val bytes: Array[Byte] = cw_mod.toByteArray();
              bytes;
       }

       def compileDecls(prog: TypeAst.TypeProg, cw: ClassWriter) : ClassWriter
           ↪  = prog match {
              case decl::xs => {
                    val cw_mod = CompileDecl.compileDecl(decl, cw)
                    compileDecls(xs,cw_mod)
                }
                case Nil => cw
        }

}
```

B.5 Main

Listing B.31: The main file combining the compiler phases together. Main.scala file.

```
import ast.Ast;
import ast.TypeAst;

import frontend.parser.ProgParser;
import frontend.desugar.DesugarProg;
import frontend.typer.ProgTyper;
import backend.mangler.MangleProg;
import backend.lifter.LiftProg;
import backend.codegen.CompileProg;
import backend.codegen.CompileToFile;

import cats.effect._;
import cats.data._;
import scala.io.Source;
import cats.effect.unsafe.implicits.global;

object Main {

        def readFile(filename: String) : IO[Either[String,String]] = IO {
                try {
                Right(scala.io.Source.fromFile(filename).mkString)
                }
                catch {case e: Exception => Left("No␣file␣named␣" + filename + "
                    ↪ .")}
        }

        def getFileName(args: Array[String]) : Either[String, String] = {
                try {
                        val filename = args(0)
                        if (filename.endsWith(".fula")) Right(filename) else
                            ↪ Left("Error:␣Must␣be␣fula␣file.")
                }
                catch {case e: Exception => Left("You␣must␣enter␣a␣filename.")}
        }

        def getTree(prog: String, className: String) : Either[String, TypeAst.
            ↪ TypeProg] = for {
                tree <- ProgParser.parseProg(prog)
                desugaredTree <- DesugarProg.desugarProg(tree, className)
                typeTree <- ProgTyper.typeProg(desugaredTree)
        } yield (typeTree)

        def compileTree(tree: TypeAst.TypeProg, filename: String): Array[Byte]
            ↪ = {
                val mangledTree = MangleProg.mangleProg(tree)
                val liftedTree = LiftProg.liftProg(mangledTree)
                CompileProg.compileProg(liftedTree, filename)
        }

        def main(args: Array[String]) : Unit = {

                val filename : Either[String,String] = getFileName(args)

                val className : Either[String, String] = filename.map(name =>
                    ↪ name.replaceAll(".+/","").stripSuffix(".fula"))
                val progStr : IO[Either[String,String]] = filename match {
                        case Right(file) => readFile(file)
                        case Left(error) => IO{Left(error)}
                }

                val progCompiled: IO[Either[String,Array[Byte]]] = progStr.map(x
                    ↪  => for {
                        prog <- x
                        name <- className
                        tree <- getTree(prog,name)
                        jvmProg = compileTree(tree,name)
                } yield (jvmProg))



                val compileByteCode: EitherT[IO,String,Unit] = for {
                        bytes <- EitherT(progCompiled)
                        name <- EitherT(IO(className))
                        _ <- EitherT(CompileToFile.writeByteCode(bytes, name))
                } yield ()

                compileByteCode.value.flatMap({
                        case Right(io) => IO(io)
                        case Left(error) => IO(println(error))
                }).unsafeRunSync

        }
    }
```

B.6 Runtime IO Library

Listing B.32: IO Library that enables writing to the console in a pure functional way.

Helper.java file.

```
package fula;

class helper {

    interface IO {
        void unsafeRunSync();
    }

    static IO printFula(final int num) {
        return new IO() {
            public void unsafeRunSync() {
                System.out.print(num);
            }
        };
    };

    static IO printlnFula(final int num) {
        return new IO() {
            public void unsafeRunSync() {
                System.out.println(num);
            }
        };
    };

    static IO printFula(final float num) {
        return new IO() {
            public void unsafeRunSync() {
                System.out.print(num);
                return;
            }
        };
    };
```

```
static IO printlnFula(final float num) {
    return new IO() {
        public void unsafeRunSync() {
            System.out.println(num);
            return;
        }
    };
};

static IO printFula(final boolean bool) {
    return new IO() {
        public void unsafeRunSync() {
            System.out.print(bool);
            return;
        }
    };
};

static IO printlnFula(final boolean bool) {
    return new IO() {
        public void unsafeRunSync() {
            System.out.println(bool);
            return;
        }
    };
};

static IO printFula(final String bool) {
    return new IO() {
        public void unsafeRunSync() {
            System.out.print(bool);
            return;
        }
        };
    };

    static IO printlnFula(final String bool) {
        return new IO() {
            public void unsafeRunSync() {
                System.out.println(bool);
                return;
            }
        };
    };

    static IO join(IO a, IO b) {
        return new IO() {
            public void unsafeRunSync() {
                a.unsafeRunSync();
                b.unsafeRunSync();
                return;
            }
        };
    }
}
```