G-assignment in "Implementation of Programming Languages" 2025

A Compiler for the FASTO Language

Deadline: Monday, June 2

Version 1.0

Preamble

This is the G-assignment[1] for the Implementation of Programming Languages course (*Implementering af programmeringssprog (IPS)*, NDAB16006U). The assignment should be solved in groups of 1–3 students. The task is made available on Monday, April 28, 2025, and your solution must be handed in by 23:59 on Monday, June 2, by uploading it on Absalon. Please explicitly state the names of all group members on the first page of your report.

In addition to the final hand-in, there will also be a *milestone handin* on Friday, May 23, where you have the chance to get feedback on your work and report so far. (It is not mandatory, but we encourage you to submit whatever you have by then.)

This document is supplemented by a partial implementation of FASTO. Your task is to complete the implementation, as well as to document and evaluate your work in a report. Your submission should include the full FASTO compiler, including your tests, in a .ZIP archive, *as well as* a report as a PDF document in which you present your solution.

This assignment will be assessed as either *pass* or *fail*. Passing this assignment is a prerequisite for participation in the final exam (in addition to passing the required fraction of the five weekly assignments), and it *cannot be resubmitted*.

Your solution should demonstrate competence in the entire course syllabus, understanding of all compiler phases, and the ability to thoroughly document your solution. Partial solutions will be considered if they are convincing and well-documented.

Please read through this entire document before you start working on the project.

Contents

1 Project / Task Description **2**

1.1 Overview . . . 2

1.2 What software to use . . . 3

1.3 Features to Implement . . . 3

1.4 Submitting Your Solution . . . 4

1.5 Accepted Limitations of the Compiler . . . 5

1.6 `fslex` and `fsyacc` . . . 5

2 The FASTO Language **5**

2.1 Lexical and Syntactical Details . . . 5

2.2 Semantics . . . 7

2.2.1 FASTO Basics . . . 7

2.2.2 FASTO Built-In Functions . . . 7

2.2.3 (Multidimensional) Arrays in FASTO . . . 7

2.2.4 Map-Reduce-Scan-Filter Programming with FASTO Arrays . . . 8

2.2.5 Map-Reduce-Scan With Lambda Expressions (Anonymous Functions) . 10

[1] Also known as “Group project” or “Godkendelsesopgave”.

2.2.6 More About Second-Order Array Combinators (SOACs) 10
2.2.7 Array Layout Used in RISC-V Code Generation 11

3 Project Tasks **11**
3.1 FASTO Features to Implement 11
3.2 Testing your Solution, Input (FASTO) Programs 14
3.3 Partitioning Your Work 15

1 Project / Task Description

1.1 Overview

The task is to complete an optimizing compiler for the FASTO language[2], described in detail in Section 2. In summary, FASTO is a simple, strongly-typed, first-order functional language, which supports arrays by special array constructors and combinators (e.g. `map` and `reduce`).

You are not expected to implement the whole compiler from scratch: We provide a partial implementation of FASTO, and you are asked to add the missing parts. The partial implementation, as well as a couple FASTO programs with their expected outputs, can be found in the archive `fasto.zip`. The archive contains five subfolders:

`Fasto/`	Contains the implementation of the compiler, ready for building under .NET 9.0.
`tests/`	Contains test programs, and corresponding input and expected output files for each.
`doc/`	Contains documentation, e.g. this document.
`bin/`	Contains a few helper scripts for compiling and executing Fasto programs, running some standardized tests, etc.
`tools/`	Contains a few non-essential tools you may find useful, such as Emacs and Vim modes for programming in FASTO.

To complete the tasks you will need to modify the following files in the `Fasto/` folder:

`Lexer.fsl`	A lexer definition for FASTO, for use with `fslex`.
`Parser.fsp`	A parser definition for FASTO, for use with `fsyacc`.
`Interpreter.fs`	An interpreter for FASTO.
`TypeChecker.fs`	A type-checker for FASTO.
`CodeGen.fs`	A translator from FASTO to RISC-V assembler. The translation is done directly from FASTO to RISC-V, i.e. without passing through a lower-level intermediate representation of the code.
`CopyConstPropFold.fs`	Copy propagation and constant folding optimizations for FASTO.
`DeadBindingRemoval.fs`	A FASTO optimization that removes unused (hence dead) `let` bindings.

There are several other modules in the compiler, which you should not need to modify, although you may need to read them and understand what they do. For completion, these are:

`AbSyn.fs`	Types and utilities for the abstract syntax of FASTO.
	This is a good place to start.

[2] FASTO stands for "Funktionelt Array-Sprog Til Oversættelse".

`SymTab.fs(i)`	A polymorphic symbol table. A symbol table is useful for keeping track of information in the various compiler passes.
`RegAlloc.fs`	A register allocator for RISC-V. You will probably not need to touch this.
`RiscV.fs`	Types and utilities for the abstract syntax of RISC-V.
`CallGraph.fs`	Function for computing the call graph of a FASTO program. Used as a building block in some optimizations.
`DeadFunctionRemoval.fs`	Optimization that removes unused ("dead") functions.
`Inlining.fs`	Aggressive inlining of all non-recursive functions.

The main program `Fasto.fs` is the driver of the compiler: it runs the lexer and parser, and then either interprets the program or validates the abstract syntax in the type checker, rearranges it in the optimizer, and compiles it to RISC-V code. After the type checker validates the program, the other stages assume that the program is type-correct. Some type information needs to be retained for code generation, which is added by the type checker.

1.2 What software to use

Instructions regarding the installation of F# and other prerequisites (.NET SDK, RARS, etc.) on Linux, macOS, and Windows are available on Absalon.

Be aware, when you develop, of any warnings or error messages introduced by your changes to the code — try to resolve them meaningfully. Your final submitted F# code should build without any warnings about incomplete pattern matches, dodgy indentation, shift/reduce-conflicts in the parser, or similar.

To run the interpreter on a file located in `tests/file.fo`, open a terminal, go to the FASTO directory and type `bin/fasto.sh -i tests/file.fo`. To compile the file without optimizations, type `bin/fasto.sh -c tests/file.fo`. This produces the file `tests/file.asm`. To compile an optimized version of the file, type `bin/fasto.sh -o tests/file.fo`.

To see the results of optimization, run `bin/fasto.sh -p <opts> tests/file.fo`, where `<opts>` is a sequence of characters referring to optimization passes. This will apply the optimizations in sequence, and print the resulting FASTO program to standard output. The valid characters in `<opts>` are `i` (for inlining), `c` (for copy propagation and constant folding), `d` (for removing dead variable bindings) and `D` (for removing dead functions).

Thus, `bin/fasto.sh -p icdcdD tests/file.fo` would, in order, inline functions, perform copy/constant propagation/folding, remove dead bindings, perform copy/constant propagation/folding again, remove dead bindings again, and finally remove dead functions. The `-o` option accepts a similar argument to explicitly specify the pipeline. If no options are passed to `-p` or `-o`, the default optimization pipeline will be used, which is equivalent to `icdD`.

To run the programs compiled by the compiler, you should use the RARS simulator [1]. RARS is written in Java, so you need a Java Runtime Environment in order to use it.

One way to run RARS and get its output directly in the command-line is by typing `java -jar bin/rars1_6.jar tests/file.asm`. (You may also use the short helper scripts `bin/rars.sh tests/file.asm` or `bin/compilerun.sh tests/file.fo`.) RARS also has a GUI, available by typing `bin/rars.sh g tests/file.asm`, which may be useful for tracking down bugs in the generated RISC-V code.

1.3 Features to Implement

In brief, you need to implement the following features in the assignment:

1. multiplication, division, negation, boolean operators (and, or, not), boolean literals; multi-variable `let`;
2. the array combinators `replicate`, `filter`, and `scan`;
3. three optimizations: "constant folding", "copy propagation" and "dead-code removal" (all are partially implemented, so you need to implement only a few important cases);
4. Bonus: an array-comprehension language construct.

Each project task is described in detail in Section 3, after the language description.

1.4 Submitting Your Solution

A solution to this assignment consists of two files to be uploaded to Absalon:

1. A .ZIP archive containing the full FASTO compiler, including your tests. Please use the same directory structure as in `fasto.zip` (source code in `Fasto/`, tests in `tests/`, etc.). **Do not submit any binaries** (remove `Fasto/bin` and `Fasto/obj`, or run `make clean`, before zipping up your solution), but do make sure that your submission builds cleanly from scratch first.

2. A *report* as a PDF document, which describes and evaluates your work and the main design decisions you took. We advise that your report (excluding any code-listing appendices) should not exceed this document in size, and should have an appropriate level of detail.

   Your report must start with a cover page stating the names of all group members.

Additionally, use the group submission submission feature on Absalon, i.e. submit only one copy per group, and make sure all group members (only) are properly included.

Contents of the Report

It is largely up to you to decide what you think is important to include in the report, as long as the following requirements are met:

Your report should justify all your changes to the compiler modules, in particular, the lexer, parser, interpreter, type checker, machine-code generator, and the optimization modules. All major design decisions should be presented and justified.

When evaluating your work, the main focus will be on verifying that your implementation of the language is *correct*. While we do not put particular emphasis on compiler optimizations in this course, we will also evaluate the *quality of the generated code*: if there are obvious inefficiencies that could have been easily solved, you may be penalized, as they testify either wrong priorities or lack of understanding.

You should not include the whole compiler text in your report, but you *must* include the parts that were either added, i.e. new code, or substantially modified. Use code listings, and discuss them in text, e.g., "Lines 4–6 implement this thing and lines 7–9 this other thing." If code listings get too big, discuss a simplified version, and add the full code listing to the appendix. Similarly, if two subtasks are similar enough, e.g., `&&` and `||`, you may just present the `&&` and say that `||` is handled similarly. Ideally, we should not need to read your raw source code.

Your report should describe whether the compilation and execution of your input/test (FASTO) programs results in the correct/expected behavior. If it does not, try to explain why this is. In addition, (i) it must be assessed to what extent the delivered test programs cover the language features, and (ii) if the implementation deviates from the correct/expected behavior, then the test program(s) should illustrate the implementation shortcomings to your best ability.

Known shortcomings in type checking, machine-code generation, and/or optimizations must be described, and, whenever possible, you need to make suggestions on how these might be corrected.

The report (excluding appendices) should not exceed this document in size, and should have an appropriate level of detail. You might be penalized if your report includes too many irrelevant details.

1.5 Accepted Limitations of the Compiler

It is perfectly acceptable that the lexer, parser, type checker, and code generator stops at the first error encountered.

It can be assumed that the translated program is small, so all target addresses for jump and branch instructions fit into constant fields of RISC-V jump instructions. (I.e., label offsets fit into 12 bits, for branch instructions.) Likewise, you may assume that no FASTO function takes more than 8 arguments, so that they can all be passed in registers.

It is not necessary to free memory in the heap while running the program. You do not need to consider stack or heap overflow in your implementation. The actual behavior of overflow is undefined, so if errors occur during execution, or you see strange results, it might be due to overflow.

1.6 `fslex` and `fsyacc`

Instructions related to the use of these tools will be given in the lectures, exercise and lab sessions. A few pointers to useful related tutorials are available from the page "Literature and Additional Resources" on Absalon.

2 The FASTO Language

FASTO is a simple, first-order functional language that allows recursive definitions. In addition to simple types (`int, bool, char`), FASTO supports arrays, which can also be nested, by providing array constructor functions (ACs) and second-order array combinators (SOACs) to modify and collapse arrays. Before we give details on the array constructors and combinators, we present the syntax and an informal semantics of FASTO's basic constructs.

2.1 Lexical and Syntactical Details

A context-free grammar of the full FASTO language (including everything you have to implement, with the exception of "bonus" task 4) is given in Figure 1. The following rules characterize the FASTO lexical atoms and clarify the syntax.

- A name (**ID**) consists of (i) letters, both uppercase and lowercase, (ii) digits and (iii) underscores, but it *must* begin with a letter. Letters are considered to range from *A* to *Z* and from *a* to *z*, i.e. English letters. Some words (`if, then, fun,...`) are reserved keywords and *cannot* be used as names.
- Numeric constants, denoted by **NUM**, take positive values, and are formed from digits 0 to 9. Numeric constants are limited to numbers that can be represented as positive integers in F#. A possible minus sign is *not* considered as part of the number literal. (Unary negation ˜ is not supported in the handed out version, but if you need a negative constant, you can initially write it as `(0-3)` instead of `~3`.)
- A character literal (**CHARLIT**) consists of a character surrounded by single quotes (`'`). A character is:

Prog	→	*FunDecs*
FunDecs	→	`fun` *Fun* *FunDecs*
FunDecs	→	`fun` *Fun*
Fun	→	*Type* ID `(` *Params* `)` `=` *Exp*
Fun	→	*Type* ID `(` `)` `=` *Exp*
Params	→	*Type* ID `,` *Params*
Params	→	*Type* ID
Type	→	`int`
Type	→	`char`
Type	→	`bool`
Type	→	`[` *Type* `]`
Exp	→	ID
Exp	→	ID `[` *Exp* `]`
Exp	→	NUM
Exp	→	`true`
Exp	→	`false`
Exp	→	CHARLIT
Exp	→	STRINGLIT
Exp	→	`{` *Exps* `}`
Exp	→	*Exp* `+` *Exp*
Exp	→	*Exp* `-` *Exp*
Exp	→	*Exp* `*` *Exp*

(... continued on the right)

Exp	→	*Exp* `/` *Exp*
Exp	→	*Exp* `==` *Exp*
Exp	→	*Exp* `<` *Exp*
Exp	→	`~` *Exp*
Exp	→	`not` *Exp*
Exp	→	*Exp* `&&` *Exp*
Exp	→	*Exp* `||` *Exp*
Exp	→	`(` *Exp* `)`
Exp	→	`if` *Exp* `then` *Exp* `else` *Exp*
Exp	→	`let` ID = *Exp* `in` *Exp*
Exp	→	ID `(` *Exps* `)`
Exp	→	ID `(` `)`
Exp	→	`read` `(` *Type* `)`
Exp	→	`write` `(` *Exp* `)`
Exp	→	`iota` `(` *Exp* `)`
Exp	→	`length` `(` *Exp* `)`
Exp	→	`replicate` `(`*Exp* `,` *Exp* `)`
Exp	→	`map` `(` *FunArg* `,` *Exp* `)`
Exp	→	`filter` `(` *FunArg* `,` *Exp* `)`
Exp	→	`reduce` `(` *FunArg* `,` *Exp* `,` *Exp* `)`
Exp	→	`scan` `(` *FunArg* `,` *Exp* `,` *Exp* `)`
Exps	→	*Exp* `,` *Exps*
Exps	→	*Exp*
FunArg	→	ID
FunArg	→	`fn` *Type* `(` `)` `=>` *Exp*
FunArg	→	`fn` *Type* `(` *Params* `)` `=>` *Exp*

Figure 1: Syntax of the FASTO Language.

1. A character with ASCII code between 32 and 126 *except* for characters `'`, `"` and \.
2. An *escape sequence*, consisting of character \, followed by one of the following characters: `t`, `n`, `'`, `"`, or \, all with the usual interpretations.

- A string literal (**STRINGLIT**) consists of a sequence of characters surrounded by double quotes (`"`). Escape sequences as described above can be used in string literals.
- Except within a string literal, any sequence of characters starting with `//` and ending at the end of the respective line is a comment and will be ignored by the lexer.
- The `+` and `-` operators have the same precedence and are both left-associative. The `<` and `==` operators have the same precedence and are both left-associative, but they both bind weaker than `+`. (The precedences and associativities of the other infix operators are discussed in Section 3.)
- The rules for the `if-then-else` and `let` expressions have the weakest precedence. For example `if a<3 then 1 else 2+x` is to be parsed as `if a<3 then 1 else (2+x)` and *not* as `(if a < 3 then 1 else 2) + x`. (Similarly for a let expression.)
- Whitespace is irrelevant for FASTO, and no lexical atoms (except string literals) contain whitespace. However, some whitespace may be needed to separate adjacent keywords and/or names that would otherwise run together.

2.2 Semantics

FASTO implements a small functional language; unless otherwise indicated, the language semantics are similar to that of F#, but note that components of array literals are separated by "," rather than ";". FASTO does not support modifying variables. That is, with the exception of its I/O read and write operations, FASTO is *purely functional*.

2.2.1 FASTO Basics

As can be seen in Figure 1, a FASTO program is a list of function declarations. Any program must contain a function called `main` that does not take any parameters. The execution of a program always starts by calling the `main` function. The return type of `main` may be any valid FASTO type. Beware, however, that while the supplied *interpreter* will show the result from `main` for ease of debugging, the code generated by the *compiler* will ignore the program result, and only produce output from explicit calls to the built-in function `write` (see below), which only allows a few specific output types.

Function scope spans through the entire program, so any function can call another one and, for instance, functions can be mutually recursive without special syntax. It is illegal to declare two functions with the same name.

Each function declares its result type and the types and names of its formal parameters. It is illegal for two parameters of the same function to share the same name. Functions and variables live in separate namespaces, so a function can have the same name as a variable. The body of a function is an expression, which can be a constant (for instance `5`), a variable name (`x`), an arithmetic expression or a comparison (`a<b`), a conditional (`if e1 then e2 else e3`), a function call (`f(1,2,3)`), an expression with local declarations, (`let x = 5 in x + y`), etc.

2.2.2 FASTO Built-In Functions

Since FASTO is strongly typed and does not support implicit casting, the built-in functions `chr` : `int` $\to$ `char` and `ord` : `char` $\to$ `int` allows one to convert explicitly between integer and character values. They are represented internally as "regular" functions, because their types are expressible in FASTO.

The functions `read` and `write` will operate on standard input / standard output. They are the only FASTO constructs that have side effects (I/O). Since `read` and `write` are polymorphic, their types are not expressible in FASTO. For this reason, the parser does not treat calls to them as regular function calls, but instead represents them by special `Read` and `Write` nodes in the abstract syntax.

The function `read` receives a type parameter that indicates the type of the value to be read: `read(int)` returns an `int`, and `read(char)` returns a `char`. These are the only valid uses of `read`.

The function `write` outputs the value of its parameter expression, and returns this value. Its valid argument types are `int`, `char`, `bool`, and `[char]` (the type of string literals and arrays of characters), e.g. `write("Hello World!")`.

Because of the special status of `read` and `write`, it is also not possible to use them in a curried form for `map` and `reduce`.

2.2.3 (Multidimensional) Arrays in FASTO

FASTO supports three basic types: `int`, `char` and `bool`. Comparisons are defined on all values of the same basic type (with `false` considered less than `true`, but addition, subtraction and other arithmetic are *only* defined on integers. As a rule, no automatic conversion between types is carried out, e.g. `if(cond) then 'c' else 1` should be rejected by the type checker.

In addition, FASTO supports an array type constructor, denoted by `[]`. Arrays can be nested. For example, `[char]` denotes the type of a vector of characters, `[[int]]` denotes the type of a two-dimensional array (i.e., array of arrays) of integers, `[[[bool]]]` denotes the type of a three-dimensional array of booleans, etc. Note that, like for lists of lists in F#, the subarrays in an array of arrays need not have the same length. The function `length` returns the length of an array.

Single-dimension indexing can be applied on arrays: if `x : [[int]]`, then `x[0]` yields the first element of array `x`, and `x[i]` yields the `i+1` element of `x`. Both `x[0]` and `x[i]` are arrays of integers, i.e. have type `[int]`. To index further into a multidimensional array, you can use a `let`-binding, as in `let v = m[3] in v[4]`. If an index falls outside the array bounds, the program will print an error and halt.

Arrays can be built in several ways:

- By the use of array literals, as exemplified in the following expression:
  `let x = 1 in { {1+x, 2+5}, {3-x, 4, 5} }`
  This represents a two-dimensional array of integers, thus type `[[int]]`.
  Note that the element values can be given by arbitrary expressions, not just constants.

- String literals are supported and they are identical to one-dimensional arrays of characters, i.e. `"abcd"` is the same as `{ 'a', 'b', 'c', 'd' }`.

- `iota` array constructor (ACs): `iota(N)`≡`{0,1,..,N-1}`, i.e. it constructs the uni-dimensional array containing the first `N` natural integers starting with `0`. Hence `N`'s value must be greater or equal to `0`. Note that `N` can be an arbitrary expression of integer type, and that the result is always of type `[int]`.

- `replicate` ACs: Assuming integer `n`, `replicate(n, a)` constructs an array of length `n` in which all (`n`) elements are `a`. Note that `a` can be an array, e.g., `a:[int]`, in which case the result of `replicate` will be a two-dimensional array (in which all elements of the outermost dimension point to `a`). The implementation of `replicate` is part of task 2, and more detail is given there.

2.2.4 Map-Reduce-Scan-Filter Programming with FASTO Arrays

We have seen so far how arrays are constructed from a (finite) set of literals, or from a scalar (with `iota`). In the following, we show how to *transform* an array in a computation, and how to *reduce* it back to a scalar or an array of smaller dimensionality. The implementation of `map` and `reduce` are provided in the hand-in code; task 2 requires you to implement `filter` and `scan` throughout the compiler (and the same for `replicate`). Figure 2 defines the second-order-array combinators (SOAC). They are named "second-order" because they receive arbitrary functions as parameters.

$$
\begin{array}{lll}
\texttt{map}\ \ (f,\ \{\ a_1,\ldots,a_n\ \}) & \equiv & \{\ f(a_1),\ldots,f(a_n)\ \} \\
\texttt{filter}\ (f,\ \{\ a_1,\ldots,a_n\ \}) & \equiv & \{\ a_{i_1},\ldots,a_{i_m}\ \},\text{such that } m \le n \text{ and } f(a_{i_k}) = \mathit{true} \\
 & & \text{i.e., } a_{i_k} \text{ are all the elements that succeed under predicate } f \\
\texttt{reduce}\ (f,\ e,\ \{\ a_1,\ a_2,\ \ldots,\ a_n\ \}) & \equiv & f(\ldots f(f(e,\ a_1),\ a_2),\ \ldots\ a_n) \\
\texttt{scan}\ \ (f,\ e,\ \{\ a_1,\ a_2,\ \ldots,\ a_n\ \}) & \equiv & \{\ f(e,\ a_1),\ \ f(f(e,\ a_1),\ a_2),\ \ldots, \\
 & & \quad f(\ldots f(f(e,\ a_1),\ a_2),\ \ldots\ a_n)\ \}
\end{array}
$$

Figure 2: Second-Order Array Combinators (SOACs) in FASTO

For example, `map` receives as parameters a function f and an array, applies f to each element of the array and creates a new array that contains the return values.

The `filter` SOAC receives as parameters a predicate f (i.e., a function of type $\alpha \rightarrow$ `Bool`) and an array (of type `[`α`]`), and returns another array, which (i) is formed by all the elements of the input array which succeed under predicate f, i.e., $f(a_{i_k}) = true$, and which (ii) maintains the partial order in which these elements appear in the input array. For example, the predicate that results in value `true` for odd numbers and `false` for even numbers can be written as `fun bool odd(int a) = not (a == (a/2)*2)`, and computing all the odd positive numbers up to `N-1` can be achieved by calling `filter(odd, iota(N))`.

Similarly, `reduce` receives as parameters (i) a function f that accepts two arguments of the same type, (ii) a start element e, and (iii) an array. (The start element e will usually, but not necessarily, be the neutral element of the operation f, e.g., 0 for addition.) `reduce` computes the accumulated result of applying the operator across all input array elements (and the start element) from left to right. For example, assuming function `plus` that adds two integers (see Figure 3 below), `reduce(plus,0,{1,2,3,4})` $\equiv$ `0+1+2+3+4` $\equiv$ `10`.

Finally, `scan` receives the same arguments as `reduce` but produces an array of the same length as the input array, by computing all prefix-sums under the given operator. For example, `scan(plus,0,{1,2,3,4})` $\equiv$ `{0+1,0+1+2,0+1+2+3,0+1+2+3+4}` $\equiv$ `{1,3,6,10}`. However, the implementation should reuse the already computed partial sums, instead of recomputing each element of the output array from scratch, as a literal reading of the definition would suggest. That is, f should only be called as many times as there are elements in the array.

Example

```
fun int plus100(int x) = x + 100
fun int plus (int x, int y) = x + y

fun [char] main() =
   let N = read(int) in        // read N from the standard input
   let a = iota(N) in          // produce a = {0,1,... N-1}
   let b = map(plus100, a)  in //  b = {100,101,...,N+99}
   let d = reduce(plus,0,a) in //  d = 0+0+1+2+...+(N-1)
   let c = map(chr, b) in      //  c = {'d','e','f',...}
   let e = write(ord(c[1])) in //  c[1] =  'e',  ord('e') = 101
   write(c)                    // output "def..." to screen
```

Figure 3: Code Example for Array Computation in FASTO

The code example in Figure 3 illustrates a simple use of arrays: First integer `N` is read from keyboard, via `read`. Then, array `a`, containing the first `N` consecutive natural numbers, is produced by `iota`. The first `map` will add each number in array `a` with 100 and will store the result in array `b`, see the implementation of `plus100`. The values in array `a` are then summed up using `reduce`. Next, `map` is called again with built-in function `chr` to convert array `b` to an array of characters, stored in `c`.

Expression `write(ord(c[1]))` (i) retrieves the second element of array `c` (`'e'`), (ii) converts it to an integer via built-in function `ord`, and (iii) prints it (`101`).

Finally, the last line prints array `c` (as a string). Note that, since `write` returns its parameter, the result of `main` is `c`, which is of type `[char]`, and matches the declared-result type of `main`.

One last observation is that `map` and `write` can be used together to print arbitrary arrays: For example, given `fun int writeInt(a) = write(a)`, then `map(writeInt, a)` prints an array of integers `a`. The shortcoming is that `map(writeInt, a)` will create a duplicate of `a`, because every call to `map` creates a new array.

[2]As a side note, if the function parameter f in `reduce` is *associative*, these constructs have well-known, efficient parallel implementations, and are known as "map–reduce" programming.

2.2.5 Map-Reduce-Scan With Lambda Expressions (Anonymous Functions)

So far, we have presented how `map`, `reduce` and `scan` work when user-defined functions are provided as arguments. This is often inconvenient, as it requires the creation of many trivial top-level functions. Furthermore, these functions cannot access variables bound at the point where the SOAC is invoked, which severely limits their usefulness.

To counter this, FASTO also allows *lambda-expressions*, a.k.a., anonymous functions, to be passed as arguments to `map`, `reduce`, `scan`. (This feature is already implemented for you.) Anonymous functions have the following syntax:

$FunArg \rightarrow$ `fn` *Type* `( ) =>` *Exp*
$FunArg \rightarrow$ `fn` *Type* `(` *TypeIds* `) =>` *Exp*

Figure 4 demonstrate the use of anonymous functions: Note that the last `map` cannot possibly be written without the use of anonymous functions because it uses the value of variable `x`, which is bound in the scope of `main`. Within an anonymous function, all variables are in scope that were also in scope outside of the SOAC containing the anonymous function.

Example

```
fun [char] main() =
  let n = read(int) in
  let a = map(fn int (int i) => read(int), iota(n)) in
  let x = read(int) in
  let b = map(fn int (int y) => x + y, a) in
  write(b)
```

Figure 4: A FASTO program using anonymous functions

2.2.6 More About Second-Order Array Combinators (SOACs)

So, what is the type of `map`? First of all, we note that the type of the result and the second argument depend on the type of the parameter function (the first argument); `map` is *polymorphic*. In fact, the `map` function is very similar to F#, and its type is $(a \rightarrow b) * [a] \rightarrow [b]$ where a and b are arbitrary types. In presence of an expression `map(f, arr)`, if `f` is a function that takes an array of type `[int]` as an argument and returns an array of type `[char]`, then the second argument `arr` must have type `[[int]]` (i.e., a 2D-array of integers), and `map(f, arr)` will return `[[char]]` (i.e., a 2D-array of characters or an array of strings).

In contrast, if `g` takes a single `bool` to an `int`, the type of `map(g, arr)` will be `[int]` and the type of `arr` has to be `[bool]`. Similar thoughts apply to the other SOACs, whose types are:

SOAC types in FASTO, F#-like notation

```
map     : (a → b) * [a] → [b]
filter  : (a → bool) * [a] → [a]
reduce  : (a * a → a) * a * [a] → a
scan    : (a * a → a) * a * [a] → [a]
```

Function types like these cannot be expressed in FASTO, so we cannot write the argument type of `map`, `filter`, `reduce` or `scan`. SOACs are therefore fixed in the language syntax. However, the type-checker verifies that the argument types of a SOAC satisfy the requirements implied by the types given above; for instance checking that the function used inside `reduce` indeed has type `a * a → a` for some type `a`, and that the other two arguments have type `a` and `[a]`.

A second concern is code generation. The code generated for `map` steps through an array in memory. However, different calls to `map` operate on different element types which take up different sizes in memory (a `char` is stored in one byte, an `int` takes up four bytes, and the elements might be arrays again, whose representation is a heap address taking up four bytes). Therefore, the respective element types must be remembered for code generation, – it is not possible to define a single function that handles all `map` calls in one and the same way. Instead, code is *generated individually* for every `map` call. The types involved in each use of `map` can be found out (i) by annotating the abstract-syntax node of each call to map during the type-checking phase, or (ii) by maintaining and inspecting the function symbol table, which provides the type of the function f and thereby determines the type of the current `map` where f is used. (At this point we “trust” the type of f, as it has been already type checked in an earlier compilation phase.)

2.2.7 Array Layout Used in RISC-V Code Generation

Figure 5 illustrates the array representation used in the RISC-V32 code generator on several code examples. In the following we consider that the word size is `4` bytes.

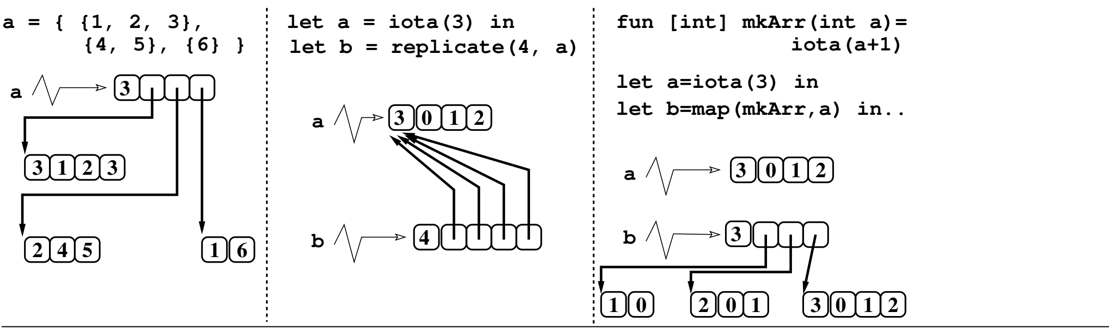

Figure 5: Array Layout.

In essence, a FASTO array is implemented with a one-word header that holds the size of the outer dimension of the array, followed by the content data. Arrays are thus contiguous in memory, and they are *word-aligned* so the address of the header is located on a memory address divisible by 4.

The amount of space an array uses depends on its type. While the array header always uses one word (4 bytes), arrays of type `[bool]` and `[char]` with n elements will use n additional bytes, and arrays of type `[int]` and `[[a]]` for some type `a` will use $4 \cdot n$ additional bytes. If the last element of an array is not on a word-aligned address, additional memory is wastefully allocated but not used, to make subsequent memory allocation more convenient. This is achieved by rounding up allocation size to the nearest multiple of 4.

For multi-dimensional arrays, the content of the array holds (one-word) pointers to arrays of one dimension lower. Several elements of an array may hold pointers to the same lower-dimensional array, as shown in the example in the middle of Figure 5.

3 Project Tasks

3.1 FASTO Features to Implement

Your task is to extend the implementation of the FASTO compiler in several ways, which are detailed below. To “extend the implementation” means to do whatever is necessary for (i) legal programs to be *interpreted* and *translated* to RISC-V code correctly, and for (ii) all illegal programs to be rejected by the compiler. The code handout contains comments of the form `TODO project task` *n* wherever you need to make changes. Where relevant, you are also free to modify other parts of the compiler, in support of your implementation of the missing features; just be sure to clearly identify your additions or changes by suitable comments in the code, and to explain and justify them in the report.

1. Part (a) Warm-Up: Multiplication, division, boolean operators and literals.

Add the operators and boolean literals given below to the expression language of FASTO, and implement support for them in all compiler parts: lexer, parser, interpreter, type checker, RISC-V code generator. This task aims to get you acquainted with the compiler internals. The implementation of these operators will be very similar to other, already provided, operators.

$$\begin{array}{lcll}
Exp & \rightarrow & Exp \texttt{ * } Exp & (*\texttt{integer} - \texttt{multiplication operator}*) \\
Exp & \rightarrow & Exp \texttt{ / } Exo & (*\texttt{integer} - \texttt{division operator}*) \\
Exp & \rightarrow & Exp \texttt{ \&\& } Exp & (*\texttt{boolean} - \texttt{and operator}*) \\
Exp & \rightarrow & Exp \texttt{ || } Exp & (*\texttt{boolean} - \texttt{or operator}*) \\
Exp & \rightarrow & \texttt{true} & (*\texttt{boolean} - \texttt{true value}*) \\
Exp & \rightarrow & \texttt{false} & (*\texttt{boolean} - \texttt{false value}*) \\
Exp & \rightarrow & \texttt{not } Exp & (*\texttt{boolean} - \texttt{negation unary operator}*) \\
Exp & \rightarrow & \texttt{\~{} } Exp & (*\texttt{integer} - \texttt{negation unary operator}*)
\end{array}$$

As usual, multiplication and division should bind stronger than addition and subtraction (which already bind stronger than the comparison operators). Likewise, the `&&` operator binds stronger than the `||` operator (and both bind even weaker than the comparisons). All four should be left-associative. Boolean negation binds stronger than `&&` and `||`, but weaker than the comparisons, and integer negation binds the strongest of all. Examples:

- `to == be || not to && be == true` means `(to==be) || ((not to)&&(be==true))`
- `~ a + b * c == b * c - a` means `((~a)+(b*c)) == ((b*c)-a)`

The boolean operators `&&` and `||` must be *short-circuiting*, as they are in C. This means that the right-hand operand of `&&` is only evaluated if the left-hand operand is true, and the right-hand operand of `||` is only evaluated if the left-hand operand is false.

Also, attempts to divide by zero should be caught and reported nicely (with an error message identifying the FASTO source location, as usual), not crash the interpreter, or cause a machine trap (or return a meaningless result) in the compiled code.

Part (b) Multiple-declaration `lets` We can generalize the syntax of FASTO so as to allow a single `let`–`in` to declare multiple variables, e.g..

```
let x = 3; y = 2*x; z = iota(y+4) in z[y]
```

The individual declarations are separated by semicolons, and there must always be at least one declaration in the sequence. The above expression is completely equivalent to a series of nested single-declaration `let`s, i.e.:

```
let x = 3 in let y = 2*x in let z = iota(y+4) in z[y]
```

Extend the grammar to support this generalization. You may need to add additional nonterminals with suitable productions, as well as modify some of the existing ones. Describe all new and modified productions (in the same style as in Figure 1) in your report, and implement them in the lexer/parser. Remember that part of the parser's job is also to *reject* syntactically ill-formed programs, such as `let in 3`, or `let x=3;;y=4 in x+y`.

Note: the more general syntax for `let` should be *desugared* into the existing one. That is, you should **not** modify the definition of the abstract syntax in `AbSyn.fs`, but simply construct nested let-bindings in the parser output, like in the example above. This means that none of the compiler phases after the parser will need to be modified to support the new syntax.

2. Implement `replicate`, `filter` and `scan`

This task is about implementing `replicate`, `filter` and `scan`.

Recall that `replicate(n,a)` builds an array of outermost length `n` by replicating `n` times the element `a`. As such, `replicate` has type `int * τ → [τ]`, where τ can be a scalar or an array type. Please note that the parameter `n` must be greater or equal to zero (since it is the size of the resulted array); otherwise program should immediately terminate with an error (message).

Recall that `filter` and `scan` types and semantics have been described in Sections 2.2.4 and 2.2.5 and 2.2.6. These operations must be added to all compiler phases. When extending the type-checker, consider making a list of things that must be checked and cross each item off once it has been checked. When extending the code generator, consider writing the generated code blocks as imperative pseudocode, e.g. with C-like syntax with explicit loops and array iterators, and write RISC-V code based on it (replacing variable names with symbolic registers and loops with conditional jumps).

3. Copy propagation, constant folding, and dead-binding removal

High-level optimizations are usually structured as a set of *optimization passes*, that each take as input a program and produce a new program that computes the same results as the old one, but sometimes more efficiently. The FASTO compiler already comes with a number of optimization passes and a framework for running them, but the passes that implement copy-propagation and constant-folding, `CopyConstPropFold.fs` and dead-binding (code) removal `DeadBindingRemoval.fs` are unfinished.

For this task, you must finish the implementation of the `CopyConstPropFold` and `DeadBindingRemoval` modules. For copy/constant propagation and dead-binding removal this corresponds to implementing the cases when the expression is (i) a variable, (ii) an array index, and (iii) a `let` binding.

For constant folding, you should implement the cases when the expression is a multiplication (e_1 `*` e_2) or a logical and (e_1 `&&` e_2), but you are encouraged to extend the rules for the other cases of expressions (plus, minus, or, not expressions, etc). Comments in the handed-out code mention where you need to write your code, and the lecture/lab slides provide more information about how the optimizer works and hints/details about what copy/constant propagation, constant folding and dead-binding removal are, and how are they to be implemented.

Shadowing The `CopyConstPropFold` design makes use of a simple symbol table, mapping variable names to constants or variables. This approach does not direcly handle the case where variables may shadow others, as seen in the function on Figure 6.

Program with shadowing

```
fun int f(int a) =
  let b = a in
  let a = 4 in // Shadows the previous 'a'.
  b            // Cannot replace 'b' with 'a' now.
```

Figure 6: A problematic FASTO function

Your solution should preferably also deal sensibly with such programs. Note that this may be achieved by either (1) detecting the problematic situations and suppressing an optimization in (only) those cases; (2) systematically renaming the shadowing variable (i.e., the `let`-bound `a` in the example) to a freshly generated (much like in the Inlining module) name, so as to avoid the problem when it is detected; or (3) rename *all* such variables proactively, so that they can never shadow existing ones. For approaches (2) and (3), note that you may be able to take advantage of the copy-propagation table to help perform the renamings. Explain and justify your design and implementation choices in the report.

4. Bonus: Array comprehension

This is a bonus task; as such not many hints will be given in written form but you are encouraged to come to discuss them with the teacher/TAs.

Array (list) comprehensions are used to provide a more compact and user-friendly notation for expressing operations on arrays. For the purpose of this task, we will use the (additional) syntax below for comprehension:

Exp	$\rightarrow$	*Compreh*
Compreh	$\rightarrow$	[*Type Exp* | *CphInps* ; *CphRes* ; *Exp*]
Compreh	$\rightarrow$	[*Type Exp* | *CphInps* ; *CphRes*]
CphInps	$\rightarrow$	*ID* <- *ID*
CphInps	$\rightarrow$	*ID* <- *ID* , *CphInps*
CphRes	$\rightarrow$	*Type ID* = *Exp*

Let us work it out on the example below (from file `tests/comprehension.fo`):

```
[int r*r | i <- x, j <-y; int r = (i+j) / 2; r - (r/5)*5 == 0 ]
```

that starts from some input arrays `x` and `y`, and eventually will produce as result an array whose length is less than or equal to the product of the lengths of input arrays (`length(x)*length(y)`).

In essence:

- `i <- x, j <-y` corresponds to the *CphInps* production and has the semantics that it selects any combination of elements, named `i` and `j`, from input arrays `x` and `y`, respectively. The elements of arrays `x` and `y` are iterated in order, such that the iterator of `y` is faster than that of `x`. Note that the constructs support an arbitrary number of input arrays.
- `int r = (i+j) / 2` corresponds to the *CphRes* production: it combines the current selection of elements into an expression that generates a result `r` of type `int`.
- `r - (r/5)*5 == 0` corresponds to the optional (last) *Exp* in the first production of *Compreh*; if this (implicitly-assumed) boolean expression evaluates to `false` than the current value of `r` is discarded, i.e., it will not generate a new element in the result array.
- Otherwise, if the boolean expression succeeds, then a new element is added to the result array by applying the formula before the bar (`|`), namely `r*r`, which is declared to have type `int`.

Note that comprehensions can be nested. Also note that the type of `r` and the type of the element of the resulted array are explicitly declared, which is not aesthetically pleasing. One can speculate that the rationale for that is the intention that this task should be solved in early-compilation stages, maybe at the parser and lexer level (without requiring modifications to the abstract-syntax tree, for example.)

Big Hint: the semantics of the comprehension construct can be implemented with a composition of SOACs! Which one?

3.2 Testing your Solution, Input (FASTO) Programs

It is your responsibility to test your implementation thoroughly. Please provide the test files in your group submission. As a starting point, some input programs can be found in folder `tests`. Note that some of these programs assume that you have already implemented multiplication, division, `filter`, `scan`, `replicate`, etc. Among them:

- `multilet.fo` tests that the *multi-let* syntax is supported.
- `fib.fo` computes the n^{th} Fibonacci number.
- `iota.fo` uses the array constructors.
- `replicate.fo` is a simple use of replicating a boolean seven times.
- `scan.fo` is a simple use of scan (with integer-addition operator).

- `reduce.fo` uses the reduce (array) combinator.
- `ordchr.fo` maps with built-in functions `ord` and `chr`.
- `proj_figure3.fo` is the program depicted in Figure 3.
- `map_red_io.fo` maps and reduces `int` and `char` arrays and performs IO.
- `inline_map.fo` tests the optimizations.
- `io_mssp.fo` implements the non-trivial algorithm for solving the "maximal segment sum" problem, which computes the maximal sum of the elements of a contiguous segment of an `[int]` array from all possible such segments.
- `filter.fo` uses a filter on a one-dimensional array (task 2).
- `filter-on-2darr.fo` uses a filter on a two-dimensional array (task 2).
- `copyConstPropFold0.fo` and `copyConstPropFold1.fo` are some simple tests for copy propagation and constant folding (task 3).
- `dead_bnd_rem.fo` can be used for testing dead-binding removal (task 3).
- `comprehension.fo` tests array comprehension (task 4).

If a test program *foo*`.fo` has a corresponding *foo*`.in` file, the program is intended to compile correctly, and produce the output in *foo*`.out` when run with the input from the input file. If no input file exist, the program is invalid, and the compiler is expected to report the error in *foo*`.err`. You can add new test programs by following the naming convention outlined above. **Tests can be run on a Unix-compatible platform using the** `bin/runtests.sh` script. You can run

```
$ ./bin/runtests.sh
```

This will compile and run every test program found in the `tests` directory, comparing actual output with expected output. Add the flag `-i` to run the tests in interpreted mode, or `-o` to turn on the compiler optimizations. You can also add a directory name containing the tests, if not `tests`.

Note that running the tests with `-o` only verifies that the optimized programs still execute correctly, but *not* that the relevant optimizations were actually performed at all. To check that, you will need to inspect the output of the optimizations maunally (using `bin/fasto.sh -p`, as described in Section 1.2.)

3.3 Partitioning Your Work

The solution and report have to be completed within approximately five weeks time. While you may be tempted to postpone work on the task towards the end of the period, this would be a bad idea. Instead, try to work on the parts described in the lecture, making the required changes in respective compiler modules and describing this part of your work in the report. It is a good idea to reserve the last week to report writing and testing.

In particular, *tasks* 1 *and* 2 require changes to all compiler phases. Try to implement each part of them immediately after the corresponding lecture. It is even possible to *start task* 1 *immediately* after you learn about parsing, because the rest of the code can be "pattern matched" from similar, already implemented, code.

Task 2 can be completed immediately after the intermediate/machine-code generation lecture. *Task* 3 can be started immediately after building the abstract-syntax tree (ABSYN) of the input program, i.e. after parsing, as it requires (only) ABSYN-level transformations, though for some of the finer points, you may want to wait until after the optimization lecture.

References

[1] RARS – RISC-V Assembler and Runtime Simulator, version 1.6. https://github.com/TheThirdOne/rars, 2023.