G-assignment in Introduction to Compilers'14

A Compiler for the FASTO Language

Deadline: Monday, December 22, 17:00

Version 1.0

Preamble

This is the G-assignment[1] in Introduction to Compilers (Oversættere), B2-2014/15. The assignment should be solved in groups of 2-3 students. The task is made available on Tuesday, November 18, 23:59, and your solution must be handed in by Monday, December 22, 17:00, by uploading it on Absalon. Use the group submission function on Absalon, and indicate the names of all group members in your report.

In addition to the final hand-in, there will also be a *milestone handin* on Wednesday, the 10th of December, where you have the chance to get feedback on your work and report so far.

This document is supplemented by a partial implementation of FASTO. Your task is to complete the implementation, as well as 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.

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

Your solution should demonstrate competence in the entire curriculum, 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.

Contents

[1] Also known as "Group project" or "Godekendelsesopgave".

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 required 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 four subfolders:

`src/`	Contains the implementation of the compiler.
`tests/`	Contains test inputs and expected output files.
`doc/`	Contains documentation, e.g. this document and `mips-module.pdf`.
`bin/`	Will contain the compiler executable (binary) file.

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

`Lexer.lex`	A lexer definition for FASTO, for use with `mosmllex`.
`Parser.grm`	A parser definition for FASTO, for use with `mosmlyac`.
`Interpreter.sml`	An interpreter for FASTO.
`TypeChecker.sml`	A type-checker for FASTO.

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

`CopyConstPropFold.sml`	Copy propagation and constant folding optimizations for FASTO.
`CodeGen.sml`	A translator from FASTO to MIPS assembler. The translation is done directly from FASTO to MIPS, i.e. without passing through a lower-level intermediate representation of the code.

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

`Fasto.sml`	Types and utilities for the abstract syntax of FASTO. This is a good place to start.
`SymTab.sml`	A polymorphic symbol table. A symbol table is useful for keeping track of information in the various compiler passes.
`RegAlloc.sml`	A register allocator for MIPS.
`Mips.sml`	Types and utilities for the abstract syntax of MIPS.
`CallGraph.sml`	Function for computing the call graph of a FASTO program. Used as a building block in some optimizations.
`DeadBindingRemoval.sml`	Optimization that removes unused ("dead") variables.
`DeadFunctionRemoval.sml`	Optimization that removes unused ("dead") functions.
`Inlining.sml`	Aggressive inlining of all non-recursive functions.

The main program `FastoC.sml`[3] 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 MIPS 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.

The compiler modules are described in detail in Appendix A.

1.2 What software to use

Please use the Moscow ML 2.10 [1]. Use `mosmlc` to compile the above modules, including the unchanged ones. For lexical and syntactical analysis, please use the accompanying Moscow ML tools — `mosmllex` and `mosmlyac`. (We have tested our implementation and will evaluate yours with these tools.)

On Linux and Mac, use the `make` command inside the `src` directory to rebuild all modules. On Windows, run the `make.bat` script instead. Be aware, when you develop, of any warnings or error messages introduced by your changes to the code — try to resolve them meaningfully.

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

To see the results of optimization, run `bin/fasto -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 -p icdcdD tests/file.fo` would, in order, inline functions, perform copy/constant propagation/folding,

[3]"`FastoC`" means "FASTO Compiler" and has nothing to do with the C programming language.

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 optimisation pipeline will be used, which is equivalent to `icdD`.

To run the programs compiled by the compiler, you should use the MARS simulator [2]. MARS is written in Java, so you need a Java Runtime Environment in order to use it.

One way to run MARS and get its output directly in the command-line is by typing `java -jar Mars4_5.jar tests/file.asm`. MARS also has a GUI available by typing `java -jar Mars4_5.jar` that is very useful for finding bugs.

1.3 Features to Implement

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

1. multiplication, division, numeric negation, boolean operators (`and`, `or`, `not`), and boolean literals;
2. the array combinators `scan` and `filter`;
3. using λ-expressions as arguments to second-order array combinators
4. two optimizations called "constant folding" and "copy propagation" (already partially implemented);
5. **(optional)** syntactical sugar for array comprehensions.

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 `src/`, tests in `tests/`, etc.). **Do not submit any binaries**, but make sure that `Make.sml` still works and produces your compiler as `bin/fasto` (or `bin/fasto.exe`).
2. A *report* as a PDF document, which describes and evaluates your work and the main design decisions you took. Your report 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 mark all the group members accordingly.

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 will be penalized, as they testify either wrong priorities or lack of understanding.

You should not include the whole compiler in your report, but you *must* include the parts that were either added, i.e. new code, or substantially modified. Add them as code listings, and use the appendix if they get too big. Ideally, we should not need to read your 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 than the test program(s) should illustrate the implementation shortcomings to your best extent.

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

The report 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 MIPS jump instructions. (i.e. label addresses fit into 16 bit, for branch instructions).

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 `mosmllex` and `mosmlyacc`

Documentation related to these tools is available in Moscow ML Owner's Manual. The manual can be found on the Moscow ML homepage [1], which also provides installation information for Windows and Linux. The course website contains a direct link to the manual. *Instructions related to the use of these tools will be given in the lectures, exercise and lab sessions.*

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) is given in Figure 1. The following rules characterise 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.

Prog	→	*FunDecs*
FunDecs	→	*Fun FunDecs*
FunDecs	→	*Fun*
Fun	→	`fun` *Type* ID (*TypeIds*) = *Exp*
Fun	→	`fun` *Type* ID () = *Exp*
TypeIds	→	*Type* ID , *TypeIds*
TypeIds	→	*Type* ID
Type	→	`int`
Type	→	`char`
Type	→	`bool`
Type	→	[*Type*]
Exp	→	ID
Exp	→	NUM
Exp	→	`true`
Exp	→	`false`
Exp	→	CHARLIT
Exp	→	STRINGLIT
Exp	→	{ *Exps* }
Exp	→	*Exp* + *Exp*
Exp	→	*Exp* - *Exp*
Exp	→	*Exp* == *Exp*
Exp	→	*Exp* < *Exp*
Exp	→	~ *Exp*
Exp	→	`not` *Exp*
Exp	→	*Exp* && *Exp*
Exp	→	*Exp* || *Exp*

(. . . continued on the right)

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	→	ID [*Exp*]
Exp	→	`iota` (*Exp*)
Exp	→	`replicate` (*Exp* , *Exp*)
Exp	→	`map` (*FunArg* , *Exp*)
Exp	→	`reduce` (*FunArg* , *Exp* , *Exp*)
Exp	→	`filter` (*FunArg* , *Exp*)
Exp	→	`scan` (*FunArg* , *Exp* , *Exp*)
Exp	→	{ *Exp* | *Bindings* }
Exp	→	{ *Exp* | *Bindings* | *Exps* }
Bindings	→	*Binding*
Bindings	→	*Binding* , *Bindings*
Binding	→	ID <- *Exp*
Exps	→	*Exp* , *Exps*
Exps	→	*Exp*
FunArg	→	ID
FunArg	→	`fn` *Type* ( ) => *Exp*
FunArg	→	`fn` *Type* (*TypeIds*) => *Exp*

Figure 1: Syntax of the FASTO Language.

- 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 Moscow ML. A possible minus sign is *not* considered as part of the number literal (negative number literals are not supported in the handed-out version, one would need to write `0-1` for `-1`).
- A character literal (**CHARLIT**) consists of a character surrounded by single quotes (`'`). A character is:
  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: `a, b, f, n, r, t, v, ?, ', "`, or by an octal value between 0 and 0177, or by `x` and a hexadecimal value between 0 and 0x7f.
- 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 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`. (Similar for a let expression.)
- Whitespace is irrelevant for FASTO, and no lexical atoms contain whitespace.

2.2 Semantics

FASTO implements a small functional language; unless otherwise indicated, the language semantics are similar to that of SML. 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. 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`** → **`char`** and **`ord`** : **`char`** → **`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 `int`, `read(char)` returns `char`, and `read(bool)` returns `bool`. These are all the 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`, `reduce`, `filter` and `scan`.

2.2.3 (Multidimensional) Arrays in FASTO

FASTO supports three basic types: `int`, `char` and `bool`. Comparisons are defined on values of all basic types, but addition, subtraction and the like 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 of integers, `[[[bool]]]` denotes the type of a three-dimensional array of booleans, etc.

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]`. If the index is outside the array bounds, the program will print an error and halt.

Arrays can be built in several ways:

- By the use of arrays literals, as exemplified in the following expression:
  `let x = 1 in { {1+x, 2+5}, {3-x, 4, 5} }`
  This represents a bi-dimensional arrays of integers, thus type `[[int]]`.
  Note that array elements can be 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` and `replicate` are array constructors (ACs): `iota(N)`≡`{0,1,..,N-1}`, i.e. it constructs the uni-dimensional array containing the first N natural integers starting with 0. Note that N can be an arbitrary expression of integer type, and that the result is always of type `[int]`.

- Similarly, `replicate(N, val)`≡`{val, val, ..., val}`, creates an array of `N` elements (hence `N` needs to be of type `int`), filled with the given element `val` as a constant. The result array has the element type of `val`, and adds an extra-dimension to those of `val`, e.g. if the type of `val` is `[[bool]]`, then the result is a 3D-array of booleans, i.e. `[[[bool]]]`.

2.2.4 Map-Reduce Programming with FASTO Arrays

We have seen so far how arrays are constructed from a (finite) set of literals, from a scalar (with `iota`), or by copying (with `replicate`). 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. This is also an excellent opportunity to familiarize ourselves with the programming style of FASTO. Figure 2 gives the definition of the second-order-array combinators (SOAC). They are named "second-order" because they receive arbitrary functions as parameters:

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.

$$
\begin{array}{lll}
\texttt{map}\ (f,\ \{\ a_1,\ldots,a_n\ \}) & \equiv & \{\ f(a_1),\ldots,f(a_n)\ \} \\
\texttt{reduce}\ (f,\ e,\ \{\ a_1,\ a_2,\ a_3\ \}) & \equiv & f(f(f(e,\ a_1),\ a_2),\ a_3) \\
\texttt{scan}\ (f,\ e,\ \{\ a_1,\ a_2,\ \ldots\ \}) & \equiv & \{\ e,\ f(e,\ a_1),\ f(f(e,\ a_1),\ a_2),\ \ldots\ \} \\
\texttt{filter}\ (f,\{\ a_1,\ a_2,\ \ldots\ \}) & \equiv & \{\ a_i\ \mid\ a_i \in \{\ a_1,\ a_2,\ \ldots\ \}\ \wedge\ f(a_i) \text{ is true }\}
\end{array}
$$

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

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. `reduce` computes the accumulated result of applying the operator across all input array elements (and the neutral element) from left to right.

Next, `scan` receives as parameters (i) a function of two arguments f (again of the same type), (ii) a start element e, and (iii) an array. `scan` computes all partial prefixes of the array elements under that operator, i.e. e, $f(e,a_1)$, $f(f(e,a_1),a_2)$, and so on.

Finally, `filter` receives as parameters a predicate function f and an array, applies f to each element of the array and creates a new array that only contains the elements for which f returned true.

```
                           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 keyboard
   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`.

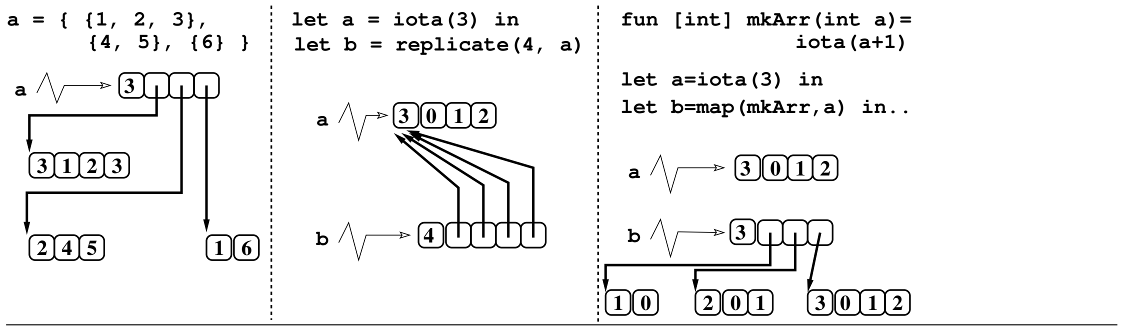
One last observation is that `map` and `write` can be used together to print arbitrary arrays: For example, given `fun int writeInt(a) = write(a)`, then `map(writeInt, a)` prints an array of integers `a`. The shortcoming is that `map(writeInt, a)` will create a duplicate of `a`, because every call to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

2.2.5 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 Standard ML's, and its type in a Standard ML-like notation 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]]`, a 2D-array of integers, and `map(f, arr)` will return `[[char]]`, a 2D-array of characters (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 in a Standard ML-like notation are:

```
               SOAC types in FASTO, SML-like notation
map      : (a → b) * [a] → [b]
reduce   : (a * a → a) * a * [a] → a
scan     : (a * a → a) * a * [a] → [a]
filter   : (a → bool) * [a] → [a]
```

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



```
a = { {1, 2, 3},
      {4, 5}, {6} }
```

```
let a = iota(3) in
let b = replicate(4, a)
```

```
fun [int] mkArr(int a)=
              iota(a+1)

let a=iota(3) in
let b=map(mkArr,a) in..
```

Figure 4: Array Layout.

Function types like these cannot be expressed in FASTO, so we cannot write the argument type of `map` or `reduce`. 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` $\rightarrow$ `a` for some type `a`, and that the other arguments correspondingly 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 a function f and thereby determines the type of the current `map` where f is used. (At this point we "trust" the type of f because it has been already type checked in an earlier compilation phase.)

2.2.6 Array Layout Used in MIPS Code Generation

Figure 4 illustrates the array representation used in the MIPS32 code generator on several code examples. In the following we consider that the word size is `4` bytes.

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 4.

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 MIPS code correctly, and for (ii) all illegal programs to be rejected by the compiler.

Except in the lexer and parser, the code handout contains comments of the form `TODO TASK` *n* whereever you need to make changes.

1. 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, MIPS 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.

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

As usual, multiplication and division bind stronger than addition and subtraction. Likewise, the `&&` operator binds stronger than the `||` operator. All four should be left-associative. Logical negation binds stronger than `&&` and `||`. Logical operators should not bind stronger than comparisons. 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.

When implementing division in the interpreter, note that the Standard ML function `div` rounds towards negative infinity, while the MIPS division instruction (and FASTO) rounds towards zero. Instead of `div`, you should use the `Int.quot` function, which rounds towards zero.

2. Implement `filter` and `scan`

The handed-out FASTO implementation only supports `map` and `reduce` as second-order array combinators. One part of the task is to add support for `filter` and `scan`, as described in Sections 2.2.4 and 2.2.5. These operations must be added to all compiler phases. Make sure you understand how `map` and `reduce` are implemented; `filter` and `scan` should work in much the same way.

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, and write MIPS code based on it (replacing variable names with symbolic registers and loops with conditional jumps).

3. λ-expressions in SOACs: This task is about implementing support for passing anonymous functions to `map`, `reduce`, `scan` and `filter`, with the following syntax.

$$\begin{array}{lll} \mathit{FunArg} & \rightarrow & \texttt{fn}\ \mathit{Type}\ (\ )\ \texttt{=>}\ \mathit{Exp} \\ \mathit{FunArg} & \rightarrow & \texttt{fn}\ \mathit{Type}\ (\ \mathit{TypeIds}\ )\ \texttt{=>}\ \mathit{Exp} \end{array}$$

The handed-out FASTO implementation already supports passing named functions, but this can be 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. For example, the program in Figure 5 could not be written without the use of anonymous functions.

```
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 5: A FASTO program using anonymous functions

Within an anonymous function, all variables are in scope that were also in scope outside of the SOAC containing the anonymous function.

To fully implement anonymous functions, you will need to modify the interpreter, type checker and code generator. These files contain comments describing a suggested strategy.

4. Copy propagation and constant folding

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 efficently. The FASTO compiler already comes with a number of optimization passes and a framework for running them, but the pass that does copy-propagation and constant-folding, `CopyConstPropFold`, is unfinished. For this task, you must finish the implementation of the `CopyConstPropFold` module - this involves work on the cases for variables and `let`-bindings, as well as adding cases for the language constructs added in task 1. Comments in the handed-out code mention where you need to make additions.

See Appendix A.5 for more on how the optimizer works, and Appendix A.5.3 in particular for details on what constant-folding you must perform.

(Optional – handle shadowing): The `CopyConstPropFold` design makes use of a simple symbol table mapping variable names to constants or variables. This approach does not handle well 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 is not required to work properly on such programs. If you wish, you may describe (or even implement) a possible solution to the problem.

5. Array Comprehensions *(optional)*

This task is optional: you do not need to complete it for your submission to pass.

Extend the lexer, parser, typechecker and compiler with array comprehensions as specified below. Array comprehensions allow us to define arrays using a very compact syntax instead of using map and filter directly.

The relevant grammar rules are listed here:

$$\begin{array}{lcl} \mathit{Exp} & \rightarrow & \{\ \mathit{Exp} \mid \mathit{Bindings}\ \} \\ \mathit{Exp} & \rightarrow & \{\ \mathit{Exp} \mid \mathit{Bindings} \mid \mathit{Exps}\ \} \\ & & \\ \mathit{Bindings} & \rightarrow & \mathit{Binding} \\ \mathit{Bindings} & \rightarrow & \mathit{Binding}\ ,\ \mathit{Bindings} \\ & & \\ \mathit{Binding} & \rightarrow & \text{ID <- } \mathit{Exp} \end{array}$$

For example, an array comprehension could look like so:

```
{ x + y | x <- iota(2), y <- iota(3) }
```

This should result in the following two-dimensional array:

```
{ { 0 + 0, 0 + 1, 0 + 2},
  { 1 + 0, 1 + 1, 1 + 2} }
```

We can add a condition that must be true, like this:

```
{ x + y | x <- iota(2), y <- iota(3) | not (x == y) }
```

Which will cause some elements to be filtered out:

```
{ { 0 + 1, 0 + 2},
  { 1 + 0, 1 + 2} }
```

Written with map and filter, the above would look like this:

```
map(fn [int] (int x) =>
        map(fn int (int y) => x + y,
            filter(fn bool (int y) => not (x == y),
                   iota(3))),
    iota(2))
```

As we can see, we map first over the array from the first binding, then the second binding and so on. The last binding is filtered using the given conditions before we finally map over it with the actual expression.

If there's more than one condition, we simply AND all conditions together:
`{ ... | not (x == y), x > 0 }` becomes the condition `not (x == y) && x > 0`.

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`. Among them:

- `fib.fo` computes the nth Fibonacci number.
- `iota.fo` and `replicate.fo` use the array constructors.
- `reduce.fo` uses the reduce (array) combinator.
- `ordchr.fo` maps with built-in functions `ord` and `chr`.
- `projFigure3.fo` is the program depicted in Figure 3.

- `mapRedIO.fo` maps and reduces `int` and `char` arrays and performs IO.
- `inlinemap.fo` tests the optimizations.
- `testIOMSSP.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.

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 is invalid, and the compiler is expected to report the error in *foo*`.err`.

You should extend and modify these test cases to test the new features you have added. You can add new test programs by following the naming convention outlined above. Tests can be run on a Unix-compatible platform using the `runtests.sh` script in the `src/` directory. Standing in the `src/` directory, run

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

This will recompile the compiler, then compile and run every test program found in the `tests` directory, comparing actual output with expected output. If `runtests.sh` complains that it cannot find MARS, run the script as follows:

```
$ MARS=/path/to/Mars4_5.jar ./runtests.sh
```

where `/path/to/Mars4_5.jar` is the path to MARS.

On Windows, you may have luck running `runtests.sh` with MSYS[4] or Cygwin[5]. Of these, Cygwin is more heavyweight, but also more likely to work.

3.3 Partitioning Your Work

The solution and report have to be completed within 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*,* 2*,* 3 *and* 5 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.

Tasks 2 *and* 3 can be completed immediately after the intermediate/machine-code generation lecture. *Task* 4 can be completed immediately after building the abstract-syntax tree (ABSYN) of the input program, i.e. after parsing, as it requires (only) an ABSYN-level translation. *Task* 5 requires that task 3 has been implemented.

References

[1] Moscow ML — a light-weight implementation of Standard ML (SML), a strict functional language used in teaching and research. http://mosml.org/, 2013.

[2] MARS, a MIPS Assembler and Runtime Simulator, version 4.5. http://courses.missouristate.edu/kenvollmar/mars/, 2002-2014. Manual: http://courses.missouristate.edu/KenVollmar/MARS/Help/MarsHelpIntro.html.

[4] http://mingw.org/wiki/msys

[5] http://cygwin.com/

[3] David A. Patterson and John L. Hennessy. *Computer Organization & Design, the Hardware/Software Interface*. Morgan Kaufmann, 1998. Appendix A is freely available at `http://www.cs.wisc.edu/~larus/HP_AppA.pdf`.

[4] Torben Ægidius Mogensen. *Introduction to Compiler Design*. Springer, London, 2011. Previously available as [6].

[5] Jost Berthold. MIPS, Register Allocation and MARS simulator. Based on an earlier version in Danish, by Torben Mogensen. Available on Absalon., 2012.

[6] Torben Ægidius Mogensen. *Basics of Compiler Design*. Self-published, DIKU, 2010 edition edition, 2000-2010. First published 2000. Available at `http://www.diku.dk/~torbenm/Basics/`. Will not be updated any more.

Appendix

A Description of the FASTO Compiler Modules

We provide an implementation of the FASTO compiler as a basis for the tasks described above. This section introduces the compiler's structure and explains some details and reasons for particular choices.

A.1 Lexer, Parser and Symbol Table

The lexer and parser are implemented in files `Lexer.lex` and `Parser.grm`, respectively. The lexical rules have been already described in Section 2.1, and the grammar has been shown in Figure 1.

File `SymTab.sml` provides a polymorphic implementation of the symbol table. The symbol table is simply a list of tuples, in which the first term is a string, denoting the name of a variable/function, and the second term is the information associated to that name. The type of the second term is polymorphic, i.e. different types of information will be stored in different tables during processing. The main functions are: (i) `bind`, which adds a name-value pair to the symbol table (potentially hiding an older entry), and (ii) `lookup`. Function `lookup` receives as input the name of a variable, `n`, and searches the symbol table to find the closest binding for `n`. If no such binding exists then option `NONE` is returned, otherwise `SOME m`, where `m` is the information associated with `n`.

A.2 Abstract Syntax Representation

The abstract syntax representation is defined in the file `Fasto.sml`: the entire program has either type `Fasto.UnknownTypes.Prog` or `Fasto.KnownTypes.Prog`, depending on which phase the compiler is in. A program is a list of function declarations. A function declaration has type `FunDec`. Expressions and types are built by the constructors of types `Exp` and `Type`, respectively.

`Exp`, `FunDec`, `Prog` and other central constructors are located inside the `FastoFn` functor. This means that whenever `TypeAnnot` occurs, the actual type is `unit` (the SML type of the value `()`) if they are used in the `UnknownTypes` structure, or `Type` if in the `KnownTypes` structure. After parsing, a program uses `UnknownTypes` version of `TypeAnnot`, namely `unit`, and after type checking, a program uses the `KnownTypes` version – `Type`.

Note that several type constructors of `Exp` contain `Type` nodes, for instance the indexing constructor is declared as `Index of string * Exp * TypeAnnot * pos`. These type nodes are replaced by the correct type by the type checker, and used in the MIPS code generation phase. For example, generating MIPS code for array indexing (`a[i]`) requires to know whether the element type is stored in one byte or in one word (four bytes), i.e. whether to use a load-byte or load-word instruction.

`Fasto.sml` also provides functions to pretty print a program (`prettyPrint`), a function (`ppFun`), a type (`ppType`), or an expression (`ppExp`). Pretty printing can be used for user-friendlier error messages and for debugging. Also, when a correct abstract syntax tree is `prettyPrint`ed, the resulting string should again be valid input to the compiler and lead to the same abstract syntax tree.

A.3 Interpreter

The Interpreter is implemented in file `Interpret.sml`. The entry point is function `evalPgm`, which builds the symbol table for functions and starts the interpretation of the program by interpreting function `main`. Interpreting a (call to a) function is implemented by function `callFun`, which: (i) evaluates the function arguments, (ii) creates a new symbol table for variables and binds the evaluated arguments to the function's formal arguments, and (iii) continues by interpreting the expression that corresponds to the body of the function. Finally, interpreting an expression is implemented by `evalExp` via case analysis on the type constructors of `Exp`.

A.4 Type Checking

The type checker is implemented in file `Type.sml`. The entry point is
`checkProgram : Fasto.UnknownTypes.Prog → Fasto.KnownTypes.Prog`, which verifies that all functions in the program have the declared type and annotates the nodes.

Type checking a function means to verify that the result type of the function matches the type of the function-body expression, given the formal parameters have the declared types. Typechecking a function *application* means to verify that the types of the actual and the formal parameters match. Expressions are type-checked recursively, building types in a bottom-up fashion. For instance, type-checking an `if-then-else` means to check that the condition is boolean, and to determine the types of the `then` and `else` sub-expressions and to verify that these types are the same. Type checking an expression is mainly implemented by `expType` via case analysis on the type constructors of `Exp`. *Your changes to the type checker will mainly extend* `expType`*'s implementation.*

As mentioned, type information also needs to be passed to the MIPS code generator, especially for arrays and for the built-in `write` function. Therefore, `checkProgram` builds a new abstract-syntax representation in which the type information of various `Fasto.UnknownTypes.Exp` constructors are changed to the correct types, represented in `Fasto.KnownTypes.Exp` (for instance the type of the array for array-indexing `a[i]`).

A.5 High-Level Optimizations

An optimizing compiler is typically organised as a set of *passes*, each of which take as input a program, and produce as output an equivalent - but optimized - program. The FASTO compiler follows this design, and implements each pass as a distinct module. All but one of these are fully implemented for you, but will be covered for completeness.

A.5.1 `CallGraph`

This module is not an optimization pass per se, but is a module containing facilities for computing the *call graph* of a FASTO program, which is used in some optimizations. The call graph contains information about which functions are called directly or indirectly by a given function. As an important special case, if a function is recursive (even indirectly), the call graph will register it as calling itself.

A.5.2 `Inlining`

Function inlining is a transformation in which the body of a function is substituted for a call to the function. This has two main benefits: first, it removes the overhead of the function call. Second, and more importantly, it acts as an *enabling optimization*, in that most optimizations are not able to cross function call boundaries, and thus can only be applied once a function has been inlined.

Inlining is not free, and can result in a expontential increase in code size, which can negatively impact performance. Production compilers use heuristics to determine when inlining should happen (usually based on the size of the function), but in FASTO we are more liberal: we aggressively inline all non-recursive functions.

```
fun int f(int x, int y) =
  (x+2) * (y-2)

fun int main() =
  let a = read(int) in
  let b = f(a, 2) in
  c
```

$\xrightarrow{Inlining}$

```
fun int f(int x, int y) =
  (x+2) * (y-2)

fun int main() =
  let a = read(int) in
  let b =
    let x = a in
    let y = 2 in
    (x + 2) * (y - 2)
  in
  b
```

The figure above illustrates the process of inlining a function `f` - note that the original function still exists in the program, and that the resulting code is somewhat awkward. It is the job of later optimizations to clean this up.

A.5.3 `CopyConstPropFold`

This module contains two conceptually separate, but implementation-wise heavily linked optimizations: *copy propagation* and *constant folding*. In copy propagation, we react to bindings such as `let x = a` or `let y = 2`, which merely "copy" a variable or constant, and replace subsequent uses of `x` and `y` with `a` and `2` respectively. In constant folding, we simplify expressions whose value can be determined at compile-time. In the running example, once we have performed the substitution `y` $\mapsto$ `2` in the expression $y - 2$, we obtain $2 - 2$, which, since it contains only constants, can be evaluated at compile-time to 0. In the original expression `(x + 2) * (y - 2)` we thus produce `(x + 2) * 0`, which can again be constant-folded to `0`.

```
fun int f(int x, int y) =
  (x+2) * (y-2)

fun int main() =
  let a = read(int) in
  let b =
    let x = a in
    let y = 2 in
    (x + 2) * (y - 2)
  in
  b
```

$\xrightarrow{Copy/constprop/fold}$

```
fun int f(int x, int y) =
  (x+2) * (y-2)

fun int main() =
  let a = read(int) in
  let b =
    let x = a in
    let y = 2 in
    0
  in
  b
```

Note that the program still contains the now-unused bindings for `a`, `x` and `y` - these are removed in the next pass.

Task 4 requires you to extend the implementation of function `copyConstPropFoldExp : (string*Propagatee) list → Exp → Exp` in file `copyConstPropFoldExp.sml` that implements both copy/constant-propagation AND constant folding.

- the 1st parameter denotes the `vtable`, which associates a variable name to its defining expression, represented via `datatype Propagatee`. The defining expression is restricted to be either (i) a constant (scalar) value, which is built with constructor `ConstProp` or (ii) another variable, i.e., `Var`, which is built with constructor `VarProp`.
- the 2nd argument denotes the expression to be optimized, and

- the expression result is the *new* optimized expression. Note that constant folding does not require the use of a symbol table.

The implementation uses case analysis on expression's constructors: **The key points in your implementation of constant/copy propagation** correspond to the `Var` and `Let` expressions:

- **For a let-binding expression** `let x =` $\texttt{e}_1$ `in` $\texttt{e}_2$:

  1 `copyConstPropFoldExp` is performed on $\texttt{e}_1$, resulting in a new expression $\texttt{enew}_1$. *Explain in your report why this is important.*

  2 If we are in a propagation case, for example $\texttt{enew}_1$ is a (constant) scalar value or a variable, then a new association is made in the symbol table. For example, if $\texttt{enew}_1$ is a variable named `"v"` then `("x", VarProp "v")` is added to `vtable`.

  3 With the updated `vtable`, `copyConstPropFoldExp` is (recursively) applied to $\texttt{e}_2$, resulting in a new expression $\texttt{enew}_2$, and the new `let`-expression `let x =` $\texttt{enew}_1$ `in` $\texttt{enew}_2$ is returned.

- **If the expression is a variable** `x`, i.e., `Fasto.Var("x", p)`, then `copyConstPropFoldExp` performs a lookup in `vtable`: If the lookup succeeds then the expression result is the expression (constant value or another variable) corresponding to the `vtable`-binding of `x`. Otherwise the result is the original expression, i.e., variable `x`. Note that `Fasto.Index` also refers to a variable that can be replaced by copy propagation in the same way ...

- for the purpose of copy/constant-propagation, the other nodes just apply `copyConstPropFoldExp` recursively on the node's subexpressions so that all program (sub)expressions are optimized.

For the implementation of constant folding you will have to implement the language constructs that you have introduced in Task 1: multiplication, division, logical and/or, logical and integer negation. Constant folding performs algebraic simplification of scalar expressions. For demonstration we look at the provided code for addition:

```
fun copyConstPropFoldExp vtable e =
    case e of ...
      | Plus (e1, e2, pos) =>
        let val e1' = copyConstPropFoldExp vtable e1
            val e2' = copyConstPropFoldExp vtable e2
        in case (e1', e2') of
               (Constant (IntVal x, _), Constant (IntVal y, _)) => Constant (IntVal (x+y), pos)
             | (Constant (IntVal 0, _), _)                     => e2'
             | (_, Constant (IntVal 0, _))                     => e1'
             | _                                               => Plus (e1', e2', pos)
        end
```

The code above calls recursively `copyConstPropFoldExp` on subexpressions `e1` and `e2` to perform propagation and algebraic simplification, resulting in optimized subexpressions `e1'` and `e2'`. Further simplifications are possible:

- if both `e1'` and `e2'` are constant values than we just create a new value (expression) which is the sum of the two,

- otherwise if `e1'` (`e2'`) is constant `0` then the result is `e2'` (`e1'`), because `0+e2' == e2'`. If `e2'` happens to be a variable than further copy propagation might be enabled.

- otherwise a new (optimized) expression is created `Plus (e1', e2', pos)`.

In a similar way your task is to implement `*, /, &&, ||, ∼, not`. For example, **For a multiplication expression**: `e1′ * e2′`, where `e1′` and `e2′` are the recursively optimized subexpression, further simplifications can be done in several cases:

- if `e1′` and `e2′` are constant values then the result will be the multiplication value,
- if `e1′` (`e2′`) is value `1` then the result is `e2′` (`e1′`) because `1 * e2′ = e2′`,
- if `e1′` (`e2′`) is value `0` then the result is constant value `0` because `0 * e2′ = 0`,
- otherwise the optimised result is `Times (e1′, e2′, pos)`.
- A similar rationale can be applied to `/, &&, ||, ∼, not`.

A.5.4 DeadBindingRemoval

The purpose of this optimization is to remove variable bindings that are not used. This is conceptually quite simple, but complicated by the fact that we cannot remove bindings whose expressions perform IO, as this would change the semantics of the program.

```
fun int f(int x, int y) =
  (x+2) * (y-2)

fun int main() =
  let a = read(int) in
  let b =
    let x = a in
    let y = 2 in
    0
  in
  b
```

$$\xrightarrow[\textit{Remove dead vars}]{}$$

```
fun int f(int x, int y) =
  (x+2) * (y-2)

fun int main() =
  let a = read(int) in
  let b = 0 in
  b
```

While dead binding removable is rarely useful on directly programmer-written code, many optimizations produce dead variables, and it is easier to remove them in a separate pass, than to build removal into the other optimizations.

A.5.5 DeadFunctionRemoval

The final optimization is to remove dead functions. This is done by computing the call graph of the program, and removing all functions that are not called (directly or indirectly) by `main`.

```
fun int f(int x, int y) =
  (x+2) * (y-2)

fun int main() =
  let a = read(int) in
  let b = 0 in
  b
```

$$\xrightarrow[\textit{Remove dead funs}]{}$$

```
fun int main() =
  let a = read(int) in
  let b = 0 in
  b
```

A.5.6 The pipeline as a whole

The optimization passes mentioned above can be executed in any order and any number of times. Indeed, for non-trivial programs, it is usually necessary to run copy-propagation/constant-folding and dead-binding-removal several times in order to reach a fixed point where no further optimization takes place. In the FASTO compiler, the `-o` option runs through the pipeline just once, but you can experiment with the `-p` option to see the effects of the various stages for yourself.

A.6 Register Allocator

The register allocator is implemented in `RegAlloc.sml`. *The project will not require changes to the register-allocator implementation.*

A.7 MIPS Code Generation

Code Generation for the MIPS architecture is implemented in file `CodeGen.sml`. The entry point is `compile : Fasto.KnownTypes.Prog → Mips.mips list`, which takes as input the abstract-syntax representation of the input program and generates a list of MIPS instructions. The type constructors for MIPS instructions are implemented in file `Mips.sml`, i.e. `datatype Mips`, together with the functionality to print them, i.e. `ppMips` and `ppMipsList`.

Function `compile` generates MIPS code for all functions declared in the program (via `compileFun`), and combines the resulting code with static code that initializes constants and executes the `main` function, also adding standard IO routines (`put/getint` and `put/getstring`) and code for exception handling (`_IllegalArrSizeError_`). Documentation on the MIPS instruction set and the mechanism via which functions are called is provided in file `mips-module.pdf`.

Code generation for an arbitrary FASTO expression is achieved via function
`compileExp: Fasto.Exp*(string*string)list*string -> Mips.mips list`
Its first argument is the expression to generate code for, the second argument is the symbol table for variables, which associates a variable name with the name of the symbolic register that holds the variable's value, and the third argument is the name of the symbolic register where the result of the expression is to be placed. The implementation of `compileExp` is done by case analysis on the type constructors of `Fasto.Exp`. *The* FASTO *extensions to implement in the tasks will mainly require extensions to* `compileExp`.

A.8 MIPS Code Generation: Array Layout

We implement an array of integers of length `n` via the following representation: We allocate `n+1` words, in which the first word holds the length of the array, and the remaining `n` words hold the array's elements, i.e., the content of the array. (An integer is stored into a 4-byte word).

The array heap is implemented as reserved data space in the compiled program pointed at by a heap pointer. Allocation of memory simply means to increment the heap pointer by the corresponding amount, the previous value points to the allocated memory.

An array of booleans or characters is implemented similarly, the only difference is that each array element requires only one `byte` rather than one word (4 bytes). On the other hand, memory allocation in the heap needs to be *word-aligned*, to avoid runtime failures caused by load-word instructions. It follows that for an array of booleans we allocate `1 + ((n+3)/4)` words: the first word stores the array's length and the remaining `(n+3)/4` words are the content space (rounding up to a full word). It follows that the content space is at least `n` bytes, each holding one array element. An array of characters is handled similarly, but an extra `0` character is added at the end, making its length logically `n+1` bytes. Therefore, `(n+4)/4` words are allocated for the content, and we write `0` as the last character. This is done in order to allow for printing the array as a string using the `putstring` system call. Finally, a multi-dimensional array requires the same

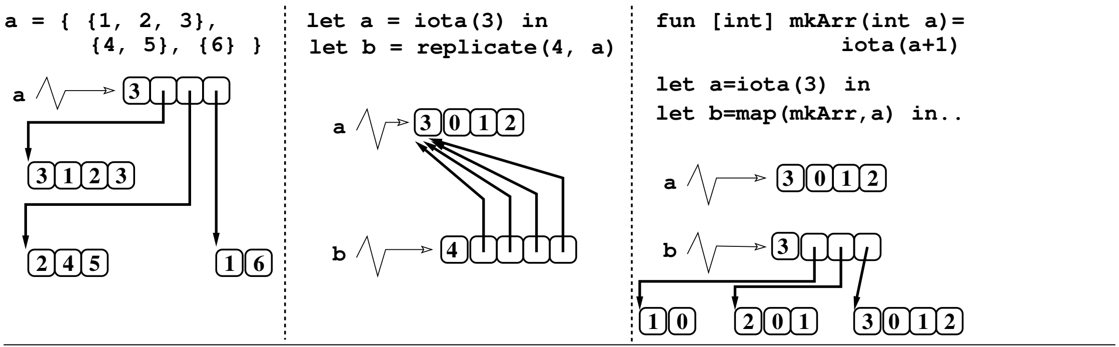


Figure 7: Array Layout.

allocation as an array of integers, however the array elements are now (word-sized) pointers to other arrays.

Figure 7 depicts and demonstrates the array layout on three code examples. The leftmost example shows a bi-dimensional array of literals: the representation of `a` is a pointer to `4`-allocated words: the first word stores the length of the (outer dimension of the) array, i.e., `3` and the other three words, denoting the array elements are pointers to the inner arrays: The first inner array has length `3` and elements `1,2,3`, the second has length `2` and elements `4, 5`, and the third one has length `1` and content `6`.

The example in the middle of Figure 7 replicates array `a = iota(3) = {0, 1, 2}` four times. The result array `b` is thus an array of length `4`, i.e. the first word in its representation holds value `4`, and the content of the array consists of four identical pointers pointing to array `a`. Finally the rightmost example uses `map` to create a bi-dimensional array, `b`, of three rows, in which the first, second and third rows have one, two and three elements, respectively. The array structure is similar to the one of the array `a` in the leftmost example.

A.9 FASTO Compiler Driver

The main driver of the (whole) compiler resides in file `src/FastoC.sml`, and the executable will be generated in `bin/fasto` (or `bin/fasto.exe` if you're on Windows). To interpret an input program located in file `data/filename.foo`, run

```
$ bin/fasto -i data/filename
```

To compile-to/generate MIPS code while applying the high-level optimizations, run

```
$ bin/fasto -o data/filename
```

To compile-to/generate MIPS code without applying the high-level optimizations, run

```
$ bin/fasto -c data/filename
```

This will generate file `data/filename.asm` that you can run with the MARS simulator, i.e.

```
$ java -jar Mars4_5.jar data/filename.asm
```

On both Linux, MacOS and Windows, the whole compiler, including the lexer and parser, can be built via

```
$ make
```

Once compiled, run

```
$ bin/fasto
```

with no arguments to get more help on which flags you can pass to the compiler.