A Compiler for the FASTO Language

Contents

1 Project / Task Description

1.1 Overview

The task is to complete an optimizing compiler for the FASTO language[1], 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.

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

`TypeChecker.sml`	A type-checker for FASTO.

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`[2] 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 a separate Compiler Design document.

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, 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`.

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

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 `iota`, `map`, and `reduce`;

Both project task are described in detail in Section 3, after the language description.

1.4 `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 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.
- 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 `\`.

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*

(... continued on the right)

Exp → *Exp* * *Exp*
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 → `map` (*FunArg* , *Exp*)
Exp → `reduce` (*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.

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` $\rightarrow$ `char` and `ord` : `char` $\rightarrow$ `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` and `reduce`.

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` array constructor (ACs): `iota(N)`$\equiv$`{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]`. The implementation of `iota` throughout the compiler is part of task 2.

2.2.4 Map-Reduce 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` throughout the compiler is part of task 2. 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.

$$\begin{array}{lll} \texttt{map} & (f,\ \{\ a_1,\ldots,a_n\ \}) & \equiv\ \{\ f(a_1),\ldots,f(a_n)\ \} \\ \texttt{reduce} & (f,\ e,\ \{\ a_1,\ a_2,\ a_3\ \}) & \equiv\ f(f(f(e,\ a_1),\ a_2),\ a_3) \end{array}$$

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.

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

[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.

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 Map and Reduce With Lambda Expressions (Anonymous Functions)

So far, we have presented how `map` and `reduce` 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.

In this sense FASTO allows lambda expressions, a.k.a., anonymous functions, to be passed as arguments to `map` and `reduce`. (This feature is already implemented for you.) Anonymous functions have the following syntax:

FunArg → `fn` *Type* `(` `)` `=>` *Exp*
FunArg → `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`.

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

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

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 Standard ML's, and its type in a Standard ML-like notation is $(a \to b) * [a] \to [b]$ 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
```

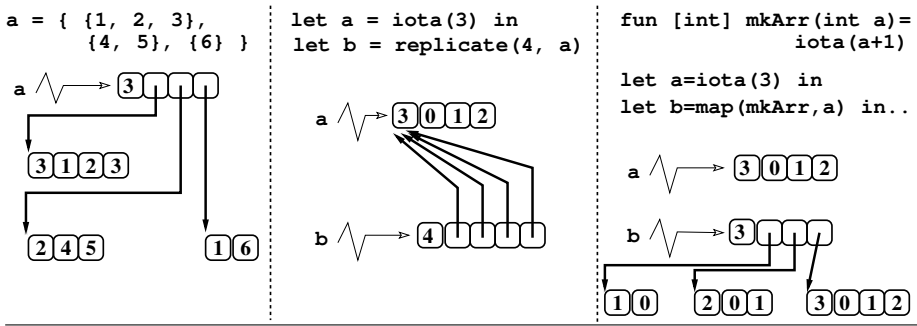

Figure 5: Array Layout.

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.)

2.2.7 Array Layout Used in MIPS Code Generation

Figure 5 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 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 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.

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

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 `iota`, `map` and `reduce`

This task is about implementing `iota`, `map` and `reduce`. Recall that (i) `iota(N)` builds the array `[0,...,N-1]` of size `N`, (ii) `iota` has type `int -> [int]`, and (iii) the parameter `N` must the greater or equal to zero (since it is the size of the resulted array).

Recall that `map` and `reduce` 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, and write MIPS code based on it (replacing variable names with symbolic registers and loops with conditional jumps). Please follow the instructions/hints for this task in the separately-provided document and in the source code.

3. 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 efficiently. 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. For copy/constant propagation this correspond to implemented the cases when the expression is a (i) variable, (ii) array index, and (iii) `let`-binding.

For constant folding, implement at least the cases when the expression is a multiplication ($e_1 \times e_2$) and logical and ($e_1 \&\& e2$), 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 a separate document provides more information about how the optimizer works and hints/details about what copy/constant propagation and constant folding are, and how are they to be implemented.

(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.

3.2 Testing your Solution, Input (FASTO) Programs

As a starting point, some input programs can be found in folder `tests`. Note that these programs assume that you have already implemented `map`, `reduce`, `iota`, etc. Among them:

- `fib.fo` computes the n^{th} Fibonacci number.
- `iota.fo` uses the array constructors.
- `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.

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 `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[3] or Cygwin[4]. Of these, Cygwin is more heavyweight, but also more likely to work.

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.

[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.

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

[4] http://cygwin.com/