A Compiler for The FASTO Language

Group Project for the Compiler Course

Block 2, Winter 2012/13

Contents

1 Introduction

This is the description of the project for the Compilers Course (Oversættere), in block 2 of 2012-2013. The project should be solved in groups of (up to) three students. The task is available from Monday, Nov.19, 2012 (after the lecture), and solutions must be handed in by Friday, Dec.21, 2012 (2:00pm), by uploading code and report via the course website at Absalon. Use the group submission function in Absalon, and indicate the names of all group members in the report. There is no standard format, please create one yourselves.

This project will be assessed as passed or failed, without a mark. Approval of this task is a prerequisite for participation in the final exam (in addition to approval of at least four of the five weekly assignments), and *cannot be resubmitted.*

2 Project / Task Description

2.1 Overview

The task is to implement a compiler for the FASTO language[1], described in detail in Section 3. In summary, FASTO is a simple strongly-typed first-order functional language which supports arrays by special array constructors and combinators (`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 to produce a working implementation of the entire language. The partial implementation, as well as a number of input programs and expected output for tests, can be found in the archive *Fasto.zip*. The archive contains four subfolders:

`SRC/`**:** contains the implementation of the compiler.

`DATA/`**:** 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 task you will need to modify the following files in `SRC` folder:

`Lexer.lex`**:** Lexical definition of FASTO's tokens, to be used with `mosmllex`.

`Parser.grm`**:** The implementation of FASTO's grammar, to be used with `mosmlyac`. The parse result is the abstract syntax tree of the input program.

`Fasto.sml`**:** The abstract syntax of FASTO, and functions to format and print entities of a FASTO program.

`Interpret.sml`**:** An interpreter for FASTO.

`Type.sml`**:** Type-checker for FASTO.

`Optimization.sml` High-level optimizations for FASTO.

`Compiler.sml`**:** 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.

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

There are two more modules in the compiler: a register allocator (`RegAlloc.sml`), and functions and types for MIPS instructions (`Mips.sml`). Most likely it will not be necessary to modify these files.

The main program `FastoC.sml` 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, and is added by the type checker when unknown.) To run the interpreter on a file located at `DATA/file.fo`, use option "-i", `$BIN/FastoC -i DATA/file` (omit the file extension `.fo`). To compile the file with optimizations, use option "-o". Otherwise use no option, i.e., `$BIN/FastoC DATA/file`.

The compiler modules are described in detail in Appendix 4.3.

Please use the Moscow ML [1] compiler, `mosmlc`, to compile the above modules, including the unchanged ones. For lexical and syntactical analysis, please use Moscow ML's tools `mosmllex` and `mosmlyac`. (We have tested our implementation and will evaluate yours with these tools.) Files `SRC/compile.sh` and `SRC/compile.bat` contain commands to rebuild all modules and the compiler under Linux and Windows, respectively. Some of the modules might generate compiler warnings, which can be safely ignored, but please be aware of any new error messages or warnings introduced by your changes to the code, i.e., check that the newly introduced warnings are indeed harmless and not bugs in your code. The bundle also contains makefiles, but you are not required to understand or use them.

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. There is a link to Mars from the course Absalon page, also see the short description in `DOC/mips-module.pdf`.

2.2 Features to Implement

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

- multiplication, division, boolean operators (AND, OR, NOT), boolean literals;
- bounds-checking for array indexing;
- using operators as functions in array combinators (explained in 3.2.4);
- one of the array combinators `zipWith` or `scan`, explained in 3.2.4, and either a `length` function or a *polymorphic* `write` function;
- an optimization pass to eliminate unused functions.

The project tasks are described in detail in Section 4, after the language description. 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.

Submitting Your Solution

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

1. A `zip` or `tar.gz` archive containing the full implementation of the compiler and tests. Please use the same directory structure as *Fasto.zip* (source code in `SRC`, tests in `DATA`). Do not submit any binaries, but make sure that the script `compile.[bat|sh]` still works and produces your compiler as `BIN/FastoC`.

2. a written *report* describing and evaluating your work and the main design decisions you took. The report should not exceed 16 pages of text, and must start by a cover page which states all group members by name.

Files are delivered via the course website. Use the group submission feature of Absalon, i.e., submit only one copy per group.

Contents of the Report The report shall include justification of the changes that were made in every compiler stage, i.e., lexer, parser, interpreter, type checker, machine-code generator.

When evaluating your work, the main focus will be to verify that the 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 code in the content of your report, but you *must* include the parts that were either added, i.e., new code, or substantially changed. Add them as Figures, or code listings in Annexes or the like. If you refer to implementation code, this must also be included in the report document.

The report shall describe whether the compilation and execution of your input/test (FASTO) programs results in the correct/expected behavior. If it does not, it should explain how and why it deviates from the expected behavior. In addition, (i) it must be assessed to what extent the delivered test programs cover all programming patterns, and (ii) if the implementation deviates from the expected behavior then the test program(s) should illustrate the shortcoming to its largest extent.

Known shortcomings in type checking and compilation must be described, and, whenever possible, you need to make suggestions on how these might be corrected. It is largely up to you to decide what you think is important to include in the report, as long as the explicit requirements of this section are met.

The report should be kept below 16 pages, and it should include all important information: all major design decisions should be presented and justified, and all deficiencies must be described. Ideally, we should not need to read your source code. You might be penalized if your report includes too many irrelevant details.

2.3 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.

2.4 MosML-Lex and MosML-yacc

Documentation related to these tools is available in Moscow ML Owners 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, and also provides an independent example of using Moscow ML's lex and yacc tools (`Lex+Parse.zip`), in Danish.

3 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.

3.1 Lexical and Syntactical Details

A context-free grammar of FASTO is given in Fig. 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`).

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

(... continued on the right)

Exp	→	*Exp* + *Exp*
Exp	→	*Exp* - *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	→	ID [*Exp*]
Exp	→	`iota` (*Exp*)
Exp	→	`replicate` (*Exp* , *Exp*)
Exp	→	`map` (ID , *Exp*)
Exp	→	`reduce` (ID , *Exp* , *Exp*)
Exps	→	*Exp* , *Exps*
Exps	→	*Exp*

Figure 1: Syntax of the FASTO Language.

- A character literal (**CHARLIT**) is surrounded by single quotes ('). A character literal 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.

- 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 +.

- Any sequence of characters starting with // and ending at the end of the respective line is a comment and will be ignored by the parser.

- Whitespace is irrelevant for FASTO, and no lexical atoms contain whitespace.

3.2 Semantics

FASTO implements a small functional language; unless otherwise indicated, the language semantics is similar to that of SML. With the exception of IO read and write operations this subset is pure, e.g., no destructive updates.

3.2.1 FASTO Basics

As can be seen in Fig. 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 with/calls 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`), and so on.

3.2.2 FASTO Built-In Functions

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

Function `read` and `write`, which read and write to standard input and output, are the only FASTO constructs having side effects (IO); the language is otherwise *purely functional*. Since `read` and `write` are polymorphic (and their types not expressible in FASTO), the parser does not treat calls to them as function calls, but instead represents them by special `Read` and `Write` nodes in the abstract syntax.

`read` receives a type parameter that indicates the type of the value to be read: `read(int)` returns `int`, `read(char)` and `read(bool)` return `char` and `bool`, respectively. These are all the valid uses of `read`.

`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!")`.

3.2.3 (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., `let a = 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, type `[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 result is undefined (but you will fix this!) 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' }`.
  Both arrays of characters and string literals are stored with an extra null `0` character at the end, which allows the IO-system calls of MIPS to use them.

- `iota` and `replicate` are array constructors (AC): `iota(N)`$\equiv$`{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)`$\equiv$`{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]]]`.

3.2.4 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 first parameter an arbitrary function f, which accepts one argument, and applies f to each element of the outer dimension of the array a, its second parameter, thereby creating a new array.

Similarly, `zipWith` receives as first parameter an arbitrary function g, which accepts two arguments, and as second and third parameter two arrays. `zipWith`

`map`	$(f,\ \{\ a_1,\ldots,a_n\}\ )$	$\equiv$	$\{f(a_1),\ldots,f(a_n)\}$
`zipWith`	$(g,\{a_1,\ldots,a_n\},\{b_1,\ldots,b_m\}\ )$	$\equiv$	$\{g(a_1,b_1),\ldots,g(a_k,b_k)\}$ with $k=\min(n,m)$
`reduce`	$(f,\ e,\ \{a_1,\ldots,a_n\}\ )$	$\equiv$	$f(\ldots(f(f(e,a_1),\ a_2),\ldots,\ a_{n-1}),a_n)$
`scan`	$(f,\ e,\ \{a_1,\ldots,a_n\}\ )$	$\equiv$	$\{\ e,\ f(\ e,\ a_1\ ),\ f(\ f(\ e,\ a_1),\ a_2\ ),\ldots$ $f(\ldots(f(f(e,a_1),\ a_2),\ldots,\ a_{n-1}),a_n)\}$

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

applies g to the elements at the same indexes in the two input arrays. This creates a result array whose size is the minimum of the two input array sizes. `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. Note that the dimension of the `reduce` result is one dimension smaller than that of the input array – for instance, the sum across all elements of a $1D$ array is a number. Finally, `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, $f(e,a_1)$, $f(f(e,a_1),a_2)$, $\ldots$. 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.

```
                                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 Fig. 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 `b` are then summed up using `reduce`. Next, `map` is called again with built-in function `chr` to convert array `a` 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 which 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, `map(writeInt,a)` prints an array of integers `a`, where `fun int writeInt(a) = write(a)`. The shortcoming is that `map(writeInt,a)` will create a duplicate of `a`, because every call to `map` creates a new array.

3.2.5 More About Second-Order Array Combinators SOAC

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 the well-known one in Standard ML, 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, a)`, 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 `a` must have type `[[int]]`, a 2*D* array of integers, and `map(f, a)` will return `[[char]]`, a 2*D* array of characters. In contrast, if `g` takes a single `bool` to an `int`, the type of `map(g, a)` will be `[int]` and the type of `a` 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]
zipWith : (a * b → c) * [a] * [b] → [c]
reduce  : (a * a → a) * a * [a] → a
scan    : (a * a → a) * a * [a] → [a]
```

Function types like these cannot be expressed in FASTO, so we cannot write the argument type of `map` 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 -> 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,

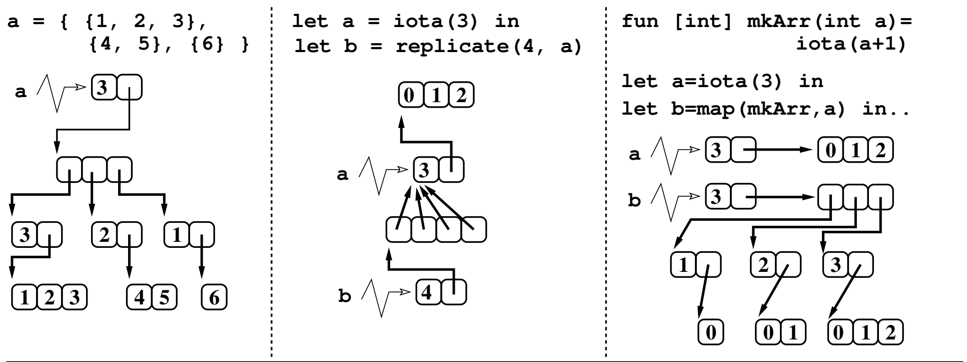

```
a = { {1, 2, 3},
      {4, 5}, {6} }
```
```
let a = iota(3) in
let b = replicate(4, a)
```
```
fun [int] mkArr(int a)=
           iota(a+1)
let a=iota(3) in
let b=map(mkArr,a) in..
```

Figure 4: Array Layout.

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.)

3.2.6 Array Layout Used in MIPS Code Generation

Figure 4 illustrates the array representation used in the MIPS32 code generator on several code examples. In essence, a FASTO array, named `arr`, is implemented by a two-word header, where the first word holds the *length* of the (outer dimension of the) array, named N, and the second word holds the address of the array's *content*. The content is a contiguous *word-aligned* chunk of memory big enough to store all array elements. If `arr :[bool]` then its size is $(N+3)/4$ words, which holds at least N bytes. Arrays of type `[char]` have an extra `'0'` element at the end (for printing), making the content size $(N+4)/4$ words (which holds at least $N+1$ bytes). If `arr :[int]`, the content size is N words, as the size of integer is one word. The same applies if an array is multidimensional: its content holds pointers (one word each) to arrays of lower dimensions. Note that several elements of an array may point to the same (lower-dimensional) array, as shown in Fig. 4.

4 Project Tasks

4.1 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.

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.

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

As usual, multiplication and division bind stronger than addition and subtraction. Likewise, the `&` operator binds stronger than the `|` operator, and both 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)`

2. Bounds Checking for the Array Index Operation. Currently, the behaviour of the implementation is undefined when indexing an array outside its bounds, e.g., `a[∼1]`. Extend the code generation for array indexing in `compileExp` to check the index value and terminate the program execution with an error message if the value is not within 0 and the array size minus one. Place your implementation in function `check_bounds`, which is currently empty. The implementation already checks that the size argument of `iota` and `replicate` is greater than zero. Implement the index-out-of-bounds error in the same way. In particular, the error message should contain the position of the failing index operation in the source code.

3. Built-In Operators/Functions Used Directly in SOAC: This task is to allow unary operators (∼ and `not`) to be passed directly as the function parameter of `map`. For this the FASTO's grammar needs to be extended such that the first argument of `map` can be the keyword `op` followed by an operator symbol. Similar changes should be made for the `reduce` function (allowing `op` followed by a binary operator, i.e., `+, *, &, |`, as the first parameter). For instance the following code should become legal in FASTO:

```
fun int main() =
  let x = iota(10) in
  reduce(op +, 0, x)
// 0+0+1+..+9=45
```

```
fun [int] main() =
    let x = iota(10) in
    map(op ~, x)
// {0, -1, -2, .., -9}
```

The implementation strategy is your choice. For example a valid approach would be to design a source-to-source translation that constructs a new abstract-syntax representation (ABSYN) in which (i) a uniquely named function is created for each such operator, i.e., `fun int plus(int a, int b) = a + b`, and in which (ii) calls such as `reduce(op +,..)` are replaced with `reduce(plus,..)`.

4. Implement *either* (i) `zipWith` and `write` *or* (ii) `scan` and `length`. 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 `zipWith` or `scan`, as described in Sections 3.2.4 and 3.2.5. The operations need to be added to all compiler phases. Make sure you understand how `map` and `reduce` are handled, then the implementation of `zipWith` and `scan` should be straightforward. Type-checking for `zipWith` is (a bit) more difficult then for `scan`, and they both require roughly the same implementation effort in the code generation phase.

The second part of the task is to implement one of the (polymorphic) built-in functions `length` and `write`, which will now operate on arrays of arbitrary types. Function `length : [a] → int` returns the length of an arbitrary array, i.e., the number of elements in the array's outer dimension. The machine code for `length` should be "short and straightforward", while type checking requires slightly more work. Function `write` currently works only on basic types and on arrays of characters (and string literals), but you will extend the implementation to print arbitrary arrays. The type-checker needs to be relaxed to allow this more general use of `write` (easy), and the interpreter and code generator need to be extended to traverse and print the array elements. Printing multidimensional arrays will require recursive calls to code generation. With both extensions, the code below should be legal:
`fun int main() = write( map(length, { "abc", "de", "f" } ) )`
and the output could possibly be: `{ 3 2 1 }`.

Implement either (i) `zipWith` *and* `write` *or (ii)* `scan` *and* `length`.

5. Dead-Function Elimination. Dead-Code Elimination is an optimization to remove code which does not affect the program results. This shrinks the program size and avoids the execution of irrelevant operations.

Implement a source-to-source translation of the abstract syntax tree which removes all unused functions from the program, i.e., functions that are defined but are never called during execution. Do this by extending the function `live_funs` in the file `Optimization.sml`. The function `live_funs` has the signature

```
Fasto.Exp * string list * (string * Fasto.FunDec) list -> string list
```

The arguments are (i) a FASTO expression, `exp`, (ii) a list of already known to be reachable function names, `alive`, and (iii) a function symbol table, `ftable`.

The return value is a list of *all* function names that are called during the execution of `exp`, including the ones already `alive` before. It is not necessary to consider whether program logic prevents a function call from being made. The result list must not contain duplicate names.

The idea is to build the result recursively by accumulating the contributions of all child expressions of `exp`. If `exp` involves a function name, `fname`, e.g. by being `Apply`, `Map` or `Reduce`, this function also needs to be investigated. If `fname` is not already in `alive`, the function is added and its body is searched for other functions that could be called from `fname`.

4.2 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 `DATA`. Among them:

- `fib.fo` computes the nth Fibonacci number.
- `testBuiltIn.fo` tests the built-in functions.
- `ArrayLitWrite.fo` tests the write operation on bi-dimensional arrays of basic types `bool`, `char`, and `int`.
- `SimpleIota.fo` and `SimpleReduce.fo` test programs with `iota` and `reduce`.
- `SimpleRead.fo` and `testRead.fo` test the read function.
- `InlineMap.fo` and `OptSimple.fo` test the optimizations.
- `testIOMSSP.fo` implements the non-trivial algorithm for solving the "maximal segment sum" problem.
- `error-01-type.fo` tests a simple type error (compilation will fail), and `runtime-error-00-iota-zero.fo` compiles, but fails at runtime.

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 `testg.sh` script in the `SRC` directory. Standing in the `SRC` directory, run

```
$ ./testg.sh
```

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

```
$ MARS=/path/to/Mars_4_2.jar ./testg.sh
```

where `/path/to/Mars_4_2.jar` is the path to Mars.

On Windows, you may have luck running `testg.sh` with MSYS[2] or Cygwin[3]. Of these, Cygwin is more heavyweight, but also more likely to work.

4.3 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 *and* 4 require changes to all compiler phases. Try to implement each part of them immediately after the corresponding lecture. It is even possible to *start task* 1 *immediately* after you learn about parsing, because the rest of the code can be "pattern matched" from similar, already implemented, code.

Task 2 can be completed immediately after the intermediate/machine-code generation lecture. It should not take more than two hours. *Tasks* 3 *and* 5 can be completed immediately after building the abstract-syntax tree (ABSYN) of the input program, i.e., after parsing, as they require (only) an ABSYN-level translation.

References

[1] Moscow ML – a light-weight implementation of Standard ML (SML). http://www.itu.dk/people/sestoft/mosml.html, 2008.

[2] MARS, a MIPS Assembler and Runtime Simulator, version 4.2. http://courses.missouristate.edu/kenvollmar/mars/, 2002-2011. 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.

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

[3] http://cygwin.com/

Appendix

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.

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 3.1, 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 variant `insert` of `bind` raises an exception when an old entry would be hidden by a new one.

Abstract Syntax Representation

The abstract-syntax representation is defined in file `Fasto.sml`: the entire program has type `Fasto.Prog`, which is a list of function declarations. A function declaration has type `Fasto.FunDec`. Expressions and types are build by the constructors of types `Fasto.Exp` and `Fasto.Type`, respectively, and so on.

Note that several type constructors of `Fasto.Exp` contain `Fasto.Type` nodes, which are initialized to `Fasto.UNKNOWN`, for instance the indexing constructor is declared as `Index of string * Exp * Type * 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 (`pp_fun`), a type (`pp_type`), or an expression (`pp_exp`). 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.

Interpreter

The Interpreter is implemented in file `Interpret.sml`, and its signature is given in file `Interpret.sig`. 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 `Fasto.Exp`.

Type Checking

The type checker is implemented in file `Type.sml`, and its signature is given in file `Type.sig`. The entry point is `checkProgram : Fasto.Prog -> Fasto.Prog`, which verifies that all functions in the program have the declared type.

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 `Fasto.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.Exp` constructors are changed from `UNKNWON` to the correct types (for instance the type of the array for array-indexing `a[i]`).

High-Level Optimizations

Several code transformation at the abstract-syntax-representation level are implemented in file `Optimization.sml`. The entry point in the optimization module is function `opt_pgm`, which first renames all variable names in all functions to (new) unique names (via the use of `unique_rename`), and then calls function `opt_fixpoint`, which applies a set of code transformations to a fix point, i.e., until no further changes are possible.

- *Let Normalization* attempts to bring `let` expressions to a normal form, e.g., `let x = (let a = iota(n) in map(f,a)) in reduce(plus,0,x)`

will be transformed via function `let_norm_exp` to
`let a=iota(n) in let x=map(f,a) in let t=reduce(plus,0,x) in t.`

- *Inlining* performs aggressive function inlining: at every stage, if a function does not call any other function directly (calls via SOAC `map` and `reduce` are not considered), its body is inlined at every call site. The inlining process is then repeated until no further inlining is possible (fix point), because inlining a function may create new opportunities for inlining other functions. The implementation's entry point is function `inline_driver`.

- *Map Fusion* replaces nested calls to `map` by function composition, using the invariant `map(g, map(f, a))` ≡ `map(g o f, a)`. The expression on the right-hand side does not materialize the intermediate array `map(f,a)` in memory, thus memory consumption is reduced considerably. The implementation is careful to only remove arrays which are not used elsewhere, e.g., with the code `let b = map(f, a) in let c = map(g, b) in` *exp*, the two `map`s are fused to `let c = map(g o f, a)` only if `b` is not used inside *exp* any more. One could replace several occurrences of `b` by `map(f,a)`, but if `f` is expensive to compute, the program might run slower because `map(f,a` is computed several times. The implementation's entry point is `mapfusion_driver`.

Note that you are not required to understand the details pertaining to the implementation of these transformations. We provide them just to demonstrate examples of high-level optimizations.

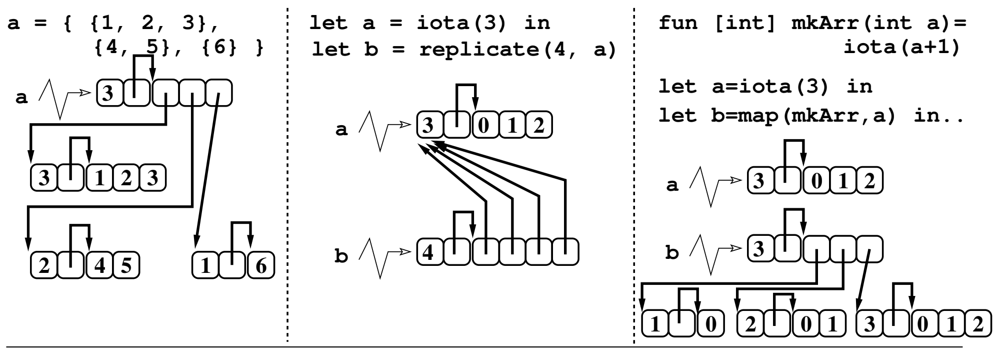
Register Allocator

The register allocator is implemented in `RegAlloc.sml` and its associated signature can be found in `RegAlloc.sig`. *The project will not require changes to the register-allocator implementation.*

MIPS Code Generation

Code Generation for the MIPS architecture is implemented in file `Compiler.sml` and its signature is given in file `Compiler.sig`. The entry point is `compile : Fasto.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., `pp_mips` and `pp_mips_list`.

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_`).



```
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 5: Array Layout.

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`.

MIPS Code Generation: Array Layout

We implement an array of integers of length `n` via the following representation: We allocate a header of 2 (four-byte) words, and a content area of `n` words. The header holds holds the length of the array, `n`, and a pointer to the content area (the content will often follow right behind the header, but this is not required in the design). The content area of `n` words holds the values of the array. 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 need to be *word-aligned*, to avoid runtime failures caused by load-word instructions. It follows that for an array of booleans we allocate `(n+3)/4` words of content space (rounding up to a full word). Again, two words are allocated for the array header (length and content pointer), and the content space is at least `n` bytes holding the array's elements. 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 allocation as an array of integers, however the array elements are now (word-sized) pointers to other arrays.

Figure 5 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 an array header holding value `3` and a pointer to the (immediately following) content space of 3 words. The three 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`. Note that the inner arrays have a similar representation.

The example in the middle of Figure 5 replicates array `a = iota(3) = {0, 1, 2}` four times. The result array `b` is thus an array of size `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.

MIPS Code Generation Case Study: Iota and Map

This section describes the MIPS translation for the array-producing functions `iota` and `map`. We start with the high-level intuition, and then we present the structure of the code that implements the corresponding MIPS translation.

```
let x = iota(n) in ...              x = dynalloc((n+2)*4);
                                    x[0] = n; x[1] = &(x[2]);
                                    for(i=0, i<x[0]; i++)
                                        x[i+2] = i;
```

The code snippet above suggests a translation of `x = iota(n)` to the `C`-like code on the right-hand side: `dynalloc` allocates space for `n+2` words, then the first word is set to `n`, the resulting-array size, and the second word points to the content of the array, i.e., the third word. The `for` loop fills the content of the array with the first `n` natural numbers (starting from 0), corresponding to `iota`'s semantics.

```
fun int plus2(b) = b + 2            x = dynalloc((a[0]+2)*4);
fun int main() =                    x[0] = a[0]; x[1] = &(x[2]);
  let x = map(plus2, a)             for(i=0, i<x[0]; i++) {
  in ..                                 x[i+2] = plus2(a[i+2]);
```

`map` is treated similarly, the only differences being that (i) the size of the result array, `x`, is found by inspecting the size of the input array, `a`, and (ii) the elements of `x` are computed by applying/calling function `plus2` on the corresponding elements of `a`. Obviously, parts of the code are the same as for `iota`, and the implementation will use helper functions to generate those parts.

```
fun compileExp e vtable place =
  case e of
  ........
  | Fasto.Iota (n, pos) =>
      let val sz_reg  = "_size_reg_"^newName()
          val code_sz = compileExp n vtable sz_reg
          fun funel(i,r) = [Mips.MOVE(r,i)]
      in  code_sz @ dynalloc( sz_reg, place, Fasto.Int(pos) ) @
          compileDoLoop( 4, sz_reg, place, funel, pos )
      end
```

The code above implements the MIPS code generation for `iota`: First, a new symbolic register is created, `sz_reg`, and the recursive call to `compileExp` generates the code that will compute `iota`'s argument, `n`, where the result of evaluating `n` will be "placed" in `sz_reg`.

Second, we define function `funel` that generates the code that will compute the value to be stored in the $\texttt{i}^{th}$ element of the result array ($\texttt{(i+2)}^{th}$ with our array layout). The arguments of `funel` are the symbolic register that stores the loop counter `i` and the result register `r`. The implementation simply moves the value of the loop counter to the result register, since for `iota`, the $\texttt{i}^{th}$ element is exactly the loop counter `i`, i.e., `a[i] = i;`.

Third, (i) the code that computes `n`, i.e., `code_sz`, is merged with (ii) the code generated by helper function `dynalloc`, which allocates memory for the result array and fills in the array's header, i.e., the first two words, and with (iii) the code that implements the loop. The latter code is computed via helper (and second-order) function `compileDoLoop` that receives as argument (and uses) function `funel` in order to generate the loop structure `for(i=0, i<x[0]; i++) x[i] = i;`.

Finally, the code that implements MIPS-code generation for `map`, given below, exhibits a similar structure: First, `compileExp` is recursively called to generate the code that stores the input-array address in symbolic register `arr_reg`.

```
fun compileExp e vtable place =
  case e of
  ........
  | Fasto.Map  (fid, arr, eltp, rtp, pos) =>
      let val arr_reg = "_arr_reg_"  ^newName()
          val inp_addr= "_arr_i_reg_"^newName()
          val sz_reg  = "_size_reg_" ^newName()
          val arr_code  = compileExp arr vtable arr_reg

          fun funel(i, r)=if ( getElSize eltp = 1 )
                          then Mips.LB(r, inp_addr, "0")
                            :: ApplyRegs(fid, [r], r, pos)
                            @ [Mips.ADDI(inp_addr,inp_addr,"1")]
                          else Mips.LW(r, inp_addr, "0")
                            :: ApplyRegs(fid, [r], r, pos)
                            @ [Mips.ADDI(inp_addr,inp_addr,"4")]
      (*** we use sz_reg to hold the input/output-array size***)
      in arr_code @ [ Mips.LW(sz_reg, arr_reg, "0")] @
         dynalloc(sz_reg, place, rtp) @
         [Mips.LW(inp_addr, arr_reg, "4")] @
         compileDoLoop(getElSize rtp, sz_reg, place, funel, pos)
      end
```

Second, function `funel` is defined: (i) `inp_arr` implements an iterator of the input array, (ii) the current element is stored in symbolic register `r`, (iii) the helper function `ApplyRegs` generates the code that calls/applies `map`'s input function `fid` on `r` and stores the result back into `r`, and (iv) finally, `inp_addr` iterator is incremented to point to the next array element. Note that the size of the array's element type, `el_tp`, is used to discriminate between the boolean/character and integer cases. With the former, the load byte (`LB`) instruction is used (instead of load word, `LW`), and `inp_addr` is incremented with one byte (rather than four bytes).

Finally, (i) the code that computes the input-array address, `arr_code`, is merged with (ii) the code that loads the input-array size in register `sz_reg`, i.e., `[ Mips.LW(sz_reg, arr_reg, "0")]`, with (iii) the result of `dynalloc`, with (iv) code to initialize the input-array iterator, i.e., `[Mips.LW(inp_addr, arr_reg, "4")]`, and with the code that generates the loop, i.e., `compileDoLoop(getElSize rtp, sz_reg, place, funel, pos)`.

FASTO Compiler Driver

The main driver of the (whole) compiler resides in file `SRC/FastoC.sml`, and the executable will be generated in `BIN/FastoC`. To interpret an input program located in file `DATA/filename.foo`, run

```
$ BIN/FastoC -i DATA/filename
```

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

```
$ BIN/FastoC -o DATA/filename
```

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

```
$ BIN/FastoC DATA/filename
```

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

```
$ java-jar Mars\_4\_2.jar DATA/filename.asm
```

Instructions related to Mars simulator for MIPS can be found in document `mips-module.pdf`.

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

```
$ make
```

or

```
$ source compile.sh
```

On Windows, use `compile.bat`, instead.

Note, these tools do not remove the generated files `Fasto.u*`, `Lexer.u*`, or `Parser.u*`. These files should be present when using the following tool.

For debugging purposes you can use the program `SeeSyntax.sml`. If you run it in the interactive system, i.e.,

```
$ mosml SeeSyntax.sml
```

and you call function `showsyntax` with an input-program filename as an argument

```
> showsyntax("../DATA/fib.fo");
```

you will see the SML data structure of the abstract syntax representation.