G-assignment in "Implementation of Programming Languages" 2023
A Compiler for the FASTO Language

Schmidt, Victor Alexander, rqc908
Ibsen, Helga Rykov, mcv462
Andersen, Magnus Raabo, dlq140

28. august 2023

Task 1

Part (a)

Lexer & Parser

We start out by accounting for the necessary modifications made to `lexer` when implementing the operators and Boolean literals to FASTO. Boolean literals and logical negation *not* have been added to the `keyword` function:

```
let keyword (s, pos) =
 match s with
   | "not"          -> Parser.NOT pos
   | "and"          -> Parser.AND pos
   | "or"           -> Parser.OR pos
   | "true"         -> Parser.TRUE pos
   | "false"        -> Parser.FALSE pos
```

The corresponding symbols for AND, OR, integer negation as well as SEMICOLON, MULTIPLICATION and DIVISION have been added to the `Token` rule:

```
rule Token = parse
  | '*'    { Parser.TIMES   (getPos lexbuf) }
  | '/'    { Parser.DIVIDE  (getPos lexbuf) }
  | '~'    { Parser.NEGATE  (getPos lexbuf) }
  | "||"   { Parser.OR      (getPos lexbuf) }
  | "&&"   { Parser.AND     (getPos lexbuf) }
  | ';'    { Parser.SEMICOLON (getPos lexbuf)}
```

Since both the relevant keywords and tokens have been defined in `lexer`, the `parser` would now know how to parse them — respectively as keywords and tokens: "NOT", "AND", "OR", etc.

To recognize the tokens above as such, `Parser` requires their definitions. Thus, we have added the following names of token types that have been added to the `lexer` above :

```
%token <Position> TIMES DIVIDE
%token <Position> NOT NEGATE
%token <Position> AND OR
%token <Position> SEMICOLON
%token <Position> TRUE FALSE
```

The next step is to define the associativity and precedence rules for the relevant tokens. We have followed the task instructions and have implemented this part as follows, keeping in mind, though, that FASTO uses the same parser generator tool as F# and that that the token binding, from the strongest to the weakest, goes bottom up:

```
%nonassoc ifprec letprec // lowest precedence
%nonassoc notprec NOT
%left LET IN SEMICOLON TO OF EQ
%left OR
%left AND
%left DEQ LTH BOOL
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc NEGATE        // highest precedence
```

Moreover, we have defined the following rules for expressions (Exp) for FASTO:

```
Exp : TRUE           { Constant (BoolVal (true),
   $1) }
    | FALSE          { Constant (BoolVal (false),
       $1) }
    | Exp TIMES Exp  { Times($1, $3, $2) }
    | Exp DIVIDE Exp { Divide($1, $3, $2)}
    | NOT Exp        { Not   ($2, $1)    }
    | NEGATE Exp     { Negate ($2, $1)   }
    | Exp AND Exp    { And   ($1, $3, $2)}
    | Exp OR Exp     { Or    ($1, $3, $2)}
```

Here, we have used the definitions for type-parameterized datatypes for expressions `"Exp<'T>"` in the `Absyn` module. For example, boolean literals are implemented as type `Constant`:

```
type Exp<'T> =
    Constant  of Value * Position ...

    type Value = BoolVal  of bool
    type Position = int * int
```

Godegen

In this subsection, we will account for the nontrivial implementation of the division operation with respect to divide by zero and boolean operators `&&` and `||` that are short-circuiting. Consider the implementation of `Divide` that returns an error message, if the divisor is zero:

```
1  | Divide (e1, e2, pos) ->
2      let t1 = newReg "divide_L"
3      let t2 = newReg "divide_R"
4      let code1 = compileExp e1 vtable t1
5      let code2 = compileExp e2 vtable t2
6      let zeroDivisorLabel = newLab "zero_divisor"
7      let checkZero = [BEQ (t2, Rzero, zeroDivisorLabel)]
8      let division = [DIV (place, t1, t2)]
9      let zeroDivisorComment = [COMMENT "Division by zero error handling"]
10     let zeroDivisorError = [LABEL zeroDivisorLabel] @ zeroDivisorComment
11     code1 @ code2 @ checkZero @ division @ zeroDivisorError
```

Line 7 checks if the divisor equals zero and jumps to label "zeroDivisorLabel", which is evaluated in `zeroDivisorError` that returns a comment "Division by zero error handling ".

We have run an additional test on `Divide` in order to test whether it returns as expected. For that purpose we have added additional files to the `tests` folder: `div.fo`, `div.in` and `div.out`.

```
div.fo:
fun int div(int x, int y) =
    x / y

fun int testDiv(int i) =
    if i == 0 then
        0
    else
        let n1 = read(int) in
        let n2 = read(int) in
        let result = write(div(n1,n2)) in
```

```
            testDiv(i-1)

fun int main() =
    let result = testDiv(5) in
    0
```

The test succeeds when run in the interpreter mode and returns the error message, when the divisor is zero: "Interpreter error: Error: Division by zero at line 2, column 7".

However, when run in the compiler mode, the test fails. Apparently, this is due to the fact that there's only one output file that catches the error in the interpreter. Yet, the error is not catched for the compiler (or rather, it catches a different error in the compiler than the interpreter error!). Even if we add a separate output file for the compiler, it will fail when run in the interpreter mode. Yet, we believe that the `Divide` is implemented according to the given requirements.

In a similar manner, we have added an additional test for `Times` to the `tests` folder: `mul.fo`, `mul.in` and `mul.out`. , which returns as expected. The same goes for the test `multilet.fo`.

Codegen continued

In order to implement the boolean operators `&&` and `||` as short-circuiting, it was essential to ensure that the relevant right-hand or left-hand operands are being evaluated in the correct order. Consider the example below, that illustrates the implementation of `Or`:

```
| Or (e1, e2, pos) ->
      let t1 = newReg "or_temp_1"
      let t2 = newReg "or_temp_2"
      let r1 = newReg "or_one"
      let code1 = compileExp e1 vtable t1
      let code2 = compileExp e2 vtable t2
      let trueLabel = newLab "or_true"
      let endLabel = newLab "or_end"
      code1 @
        [ LI (r1, 1) // Load 1 into r1
        ; BEQ (t1, r1, trueLabel) ]// If e1 is true, jump to trueLabel
        @ code2 @
```

```
13            [ BEQ (t2, r1, trueLabel)  // If e2 is true, jump to trueLabel
14            ; LI (place, 0)            // If both e1 and e2 are false, set result to false
15            ; J endLabel               // Jump to endLabel
16            ; LABEL (trueLabel)        // Label for true case
17            ; LI (place, 1)            // Set the result to true
18            ; LABEL (endLabel)         // Label for end of function
19            ]
```

In the code above, line 12 is only executed and `code2` is evaluated only if the result of `BEQ` is not 1 (false). Have `code2` been evaluated prior to line 11, that would have led to the tests involving this operator fail.

The `And` operator have been implemented in a similar manner, except that `code2` is evaluated only if the result of `BEQ` is 1 (true) (cf. the file for the detail).

Finally, the tests for the boolean oprators, logical and integer negation (`negate`, `short_circuit`) also return as expected.

Part (b)

In order to allow a single `let-in` to declare multiple variables, where the individual declarations are separated by semicolons and are equivalent to a series of nested single-declaration `lets`, we have added to the `parser` the following two rules for declarations (Decs):

```
Decs : ID EQ Exp SEMICOLON Decs
         { Let (Dec (fst $1, $3, $2), $5, $2) }
       | ID EQ Exp IN Exp %prec letprec
         { Let (Dec (fst $1, $3, $2), $5, $2) }
;
```

A declaration can be an identifier, an equals sign, an expression, a semicolon, and more declarations, or an identifier, an equals sign, an expression, an IN keyword, and another expression. The actions create let-nodes in the syntax tree. All tests involving let-in expressions with semicolons (fx `multilet.fo`) return as expected.

Task 2

This task asks us to implement the array combinators `replicate`, `filter` and `scan`.

Lexer

For the lexer to be able to gather tokens corresponding to the keywords of the array combinators, we add the following tokens to the keyword-special section of the lexer:

```
let keyword (s, pos) =
match s with
...
| "filter"        -> Parser.FILTER pos
| "replicate"     -> Parser.REPLICATE pos
| "scan"          -> Parser.SCAN pos
...
```

Parser

Next, we add the following productions for `Exp` to the parser:

```
Exp : ...
    | FILTER LPAR FunArg COMMA Exp RPAR
        { Filter ($3, $5, (), $1) }
    | REPLICATE LPAR Exp COMMA Exp RPAR
        { Replicate ($3, $5, (), $1) }
    | SCAN LPAR FunArg COMMA Exp COMMA Exp RPAR
        { Scan ($3, $5, $7, (), $1) }
```

to allow the parser to parse the array combinators. We also add the array combinator tokens to the parser, so that they can be identified. The position for which we define these tokens determine the precedence of the array combinators. We define them on the same line as the already implemented array combinators `reduce` and `map` to give them the same precedence. As such we have the following tokens defined in the parser in the order stated below:

```
%token <int * Position> NUM
%token <char * Position> CHARLIT
%token <string * Position> ID STRINGLIT
%token <Position> IF THEN ELSE LET IN EOF
```

```
%token <Position> INT CHAR BOOL
%token <Position> PLUS MINUS LESS
%token <Position> TIMES DIVIDE
%token <Position> NOT NEGATE
%token <Position> AND OR
%token <Position> DEQ LTH EQ MAP REDUCE IOTA ARROW
%token <Position> AND OR
%token <Position> DEQ LTH EQ MAP REDUCE FILTER SCAN REPLICATE IOTA ARROW
%token <Position> FUN FN COMMA SEMICOLON READ WRITE
%token <Position> LPAR RPAR LBRACKET RBRACKET LCURLY RCURLY
%token <Position> TRUE FALSE
```

Type Checker

For typechecking we add the array combinators in the `checkExp` function of the type checker as individual cases that are pattern-matched with `exp`. We have that `filter` has the type $(a \to \text{bool}) * [a] \to [a]$, i.e. takes a function returning a boolean as first argument and an array as second argument and returns an array of elements with the same type as the elements in the input array. We therefore first check that the second argument is an array and next that the first argument is a function returning a boolean. Finally we check that the type in the input array corresponds to the type of the input argument of the function. This is done in the order explained by the following code:

```
| Filter (func, arr_exp, _, pos) ->
    let (arr_type, arr_dec) = checkExp ftab vtab arr_exp
    let elem_type =
        match arr_type with
            | Array t -> t
            | _ -> reportTypeWrongKind "third argument of
                filter" "array" arr_type pos
    let (f', f_res_type, f_arg_type) =
        match checkFunArg ftab vtab pos func with
        | (f, res, [a1]) ->
            if res <> Bool then
                reportTypeWrongKind "function is filter
                    does not return" "bool" res pos
                (f, res, a1)
        | (_, res, args) ->
            reportArityWrong "operation in filter" 1 (
                args,res) pos
```

```
if elem_type <> f_arg_type then
    reportTypesDifferent "operation and array-element
        types in filter"
                         f_arg_type elem_type pos
let return_type = checkFun ftab func
    if return_type <> elem_type then
    reportTypesDifferent "return and array-element
       types in filter"
                         return_type elem_type pos
(Array elem_type, Filter (f', arr_dec, elem_type, pos
   ))
```

We have that **replicate** has the type int $* \, a \to [a]$, i.e. takes an integer as first argument and a type as first argument an returns an array of that type. To ensure this we check that the first argument is an integer. This is done in the order explained by the following code:

```
| Replicate (e_exp1, e_exp2, _, pos) ->
    let (t1, e1) = checkExp ftab vtab e_exp1
    let (t2, e2) = checkExp ftab vtab e_exp2
    if t1 <> Int then
        reportTypeWrong "argument of replicate" Int t1
            pos
    (Array t2, Replicate (e1, e2, t2, pos))
```

Finally we have that **scan** has the type $(a * a \to a) * a * [a] \to [a]$. We first check that the third argument is an array and next we check that that the first argument is a function and that it takes two arguments of the same type. Then we check that the function returns a type that is identical to the type of the function arguments. Finally we check that the third argument array has the same type as the arguments of the first argument function and that the second argument is of the same type as the arguments for the first argument function. This is done in the order explained by the following code:

```
| Scan (f, n_exp, arr_exp, _, pos) ->
    let (n_type  , n_dec  ) = checkExp ftab vtab n_exp
    let (arr_type, arr_dec) = checkExp ftab vtab arr_exp
    let test_arr_type =
        match arr_type with
          | Array t -> t
          | _       -> reportTypeWrongKind "third
            argument of Scan" "array" arr_type pos
    let (f', f_argres_type) =
        match checkFunArg ftab vtab pos f with
```

```
    | (f', res, [a1; a2]) ->
      if a1 <> a2 then
          reportTypesDifferent "argument types
             of operation in scan"
                               a1 a2 pos
      if res <> a1 then
          reportTypesDifferent "argument and
             return type of operation in scan
             "
                               a1 res pos
          (f', res)
    | (_, res, args) ->
        reportArityWrong "operation in scan" 2
          (args,res) pos
if test_arr_type <> f_argres_type then
    reportTypesDifferent "operation and array type
       in scan"
                         f_argres_type
                            test_arr_type pos
if n_type <> f_argres_type then
    reportTypesDifferent "operation and array-
       element types in scan"
                         f_argres_type n_type pos
(Array test_arr_type, Scan (f', n_dec, arr_dec,
   test_arr_type, pos))
```

Code Generation

The `filter` function is performed by iterating over the input array, applying the function to each element, and storing the filtered elements in a new array. This is done by the RISC-V code seen in appendix A. We have implemented it so that it performs the following in the stated order:

1. Initialize registers for intermediate values.
2. Evaluate expressions into their respective registers.
3. Retrieve the size of the input array.
4. Initialize necessary registers for filtering.
5. Define a loop with a header, body, and footer.

6. In the loop header, compare the iterator with the size of the input array and load the current element.
7. Apply the function f to the current element.
8. In the loop, check the result of the function and store the element if non-zero.
9. Update registers at the end of each loop iteration.
10. Concatenate all instruction sequences to form the final RISC-V code.

The `replicate` function replicates a value `a_exp` `n_exp` times into a new array. This is implemented by the RISC-V code seen in appendix A. We have implemented it so that it performs the following in the stated order:

1. Initialize registers for intermediate values.
2. Evaluate expressions into their respective registers.
3. Evaluate the size of the input/output array (`n_exp`) and store it in `size_reg`.
4. Define a label (`safe_lab`) and instructions to check if the size is non-negative. If the size is negative, it raises a runtime error with a specific message.
5. Initialize necessary registers for the replication process.
6. Define a loop with a header, body, and footer.
7. In the loop header, compare the iterator (`i_reg`) with the size of the input/output array (`size_reg`).
8. In the loop body, store the value of `a_exp` into the new array at the current address (`addr_reg`) using the Store instruction. Then increment the address by the size of the element.
9. Update the iterator at the end of each loop iteration and jump back to the loop beginning.
10. Concatenate all instruction sequences together in the required order to form the RISC-V code.

The `scan` function iterates over an input array, applying a given binary operator to the accumulator and each element of the array, and storing the intermediate results in a result array. This is implemented by the RISC-V code seen in appendix A. We have implemented it so that it performs the following in the stated order:

1. Initialize registers for intermediate values, including `arr_reg` for the address of the array, `size_reg` for the size of the input array, `i_reg` as a loop counter, `tmp_reg` for temporary purposes, `addr_reg` for the current result element address, and `acc_reg` for the accumulator.

2. Define labels for the loop beginning and end.

3. Evaluate expressions into their respective registers.

4. Retrieve the size of the input array using a load word instruction (LW) and store it in `size_reg`.

5. Allocate memory for the result array using the dynalloc function.

6. Compile the initial value expression `acc_exp` into RISC-V instructions and store the result in `acc_reg`.

7. Set `arr_reg` to the address of the first element of the input array and set `i_reg` to 0 for the loop.

8. Define the loop code, including a comparison between `i_reg` and `size_reg` to determine the loop termination condition.

9. Load the current element from the input array into `tmp_reg` using the Load instruction.

10. Apply the binary operator binop to the accumulator `acc_reg` and `tmp_reg` using the applyFunArg function and store the result back in `acc_reg`.

11. Store the result of the operator in the result array at the current address `addr_reg`.

12. Update the `addr_reg` to point to the next result element.

13. Concatenate all instruction sequences together in the required order to form the RISC-V code.

Task 3

Task 3 asked us to implement the missing parts of `CopyConstPropFold.fs` and `DeadBindingRemova.fs`. In the following sections, we will touch on the non-trivial implementations of each.

`CopyConstPropFold.fs`

Copy / constant propagation was to be implemented on:

1. variables,
2. array indexing, and
3. let-expressions, where the expression associated with the name could be a:
4. variable,
5. constant, or
6. another let expression.

of these, only the let-expressions within let-expressions has a non-trivial solution.

Let-expressions within let-expressions have been implemented, not as hinted at, but rather as the expression in the given context. The comment hinted at implementing the recursive solution for:

`let y = (let x = e1 in e2) in e3`
which would be `let x = e1 in let y = e2 in e3`

however we did not go this route. Instead we implemented the given case (the first of the previous lines), where `let y = e in body` where `e = (let x = e1 in e2)`. The following steps have been implemented (the code is too wide to fit neatly in the report, please check the code for yourselves):

1. Evaluate `let x = e1 in e2` recursively (we have to pretend this is no inner let-expressions for the recursion to evaluate the expressions).
2. Bind `y` to `e2`, in other words `let y = e2` in the vtable.

3. Evaluate the `body` expression using the new vtable (where `y = e2`, but `x` $\neq$ `e1`).

4. Return the newly optimized let-expression, where `e1, e2, e3` all have been optimized.

... A quick elaboration as to why we did not go with the hinted-at solution, is due to that *that* implementation failed the test `inlining_map.fo`, whereas our current direct implementation, passed this test and the other required tests.

Quick note on shadowing: Our solution does not account for potential shadowing. This results in that the `inlining_shadowing.fo` test fails. Shadowing was not required to work for us to pass the group project, so we decided our time were better spent elsewhere (e.g. this report).

`DeadBindingRemoval.fs`

The removal of dead bindings needed to be implemented when a dead variable was assigned a:

1. variable,

2. array index, or

3. a let-expression.

again only the let-expression has a non-trivial solution (some might even consider it trivial, but we will go over it either way).

Checking if a let-expression is dead. To check whether or not `let x = e in body` is a dead expression, we have to find out if `x` is used in `body` or if `e` contains IO. To split the process into steps, this is what we did:

1. Evaluate the expression `e`.

2. Evaluate the `body`.

3. If `e` contains IO or `body`'s DRB contains `x` then the let-binding is required, as `x` is necessary to fulfill some purpose and therefore can not be removed. Otherwise, the `body` can replace the whole let-expression, as `x` is considered a dead binding.

A RISC-V code for Task 2

```
Filter (f, arr_exp, tp, pos) ->
      let size_reg = newReg "size" (* size of input array
          *)
      let arr_reg  = newReg "arr"  (* address of input
         array *)
      let inc_reg  = newReg "inc"  (* incrementer /
         counter *)
      let addr_reg = newReg "addr" (* address of output
         array *)
      let i_reg    = newReg "i"    (* iterator *)
      let tmp_reg  = newReg "tmp"  (* temporary register
         *)
      let res_reg  = newReg "res"  (* holds current input
          element *)
      let elem_size = getElemSize tp

      let arr_code = compileExp arr_exp vtable arr_reg

      let get_size = [ LW (size_reg, arr_reg, 0) ] (*
         size of input array *)

      let init_regs = [ ADDI (addr_reg, place, 4)
                      ; MV (i_reg, Rzero)
                      ; MV (inc_reg, Rzero)
                      ; ADDI (arr_reg, arr_reg, 4)
                      ]

      let loop_beg = newLab "loop_beg"
      let fil_false = newLab "fil_false"
      let loop_end = newLab "loop_end"

      let loop_header = [ LABEL (loop_beg)
                        ; BGE (i_reg, size_reg, loop_end)
                        ; Load elem_size (res_reg,
                            arr_reg, 0)
                        ]

      let apply_code =
          applyFunArg (f, [res_reg], vtable, tmp_reg, pos
             )
```

```
      let loop_filter = [ BEQ (tmp_reg, Rzero, fil_false)
                        ; SW (res_reg, addr_reg, 0)
                        ; ADDI (addr_reg, addr_reg,
                            elemSizeToInt elem_size)
                        ; ADDI (inc_reg, inc_reg, 1)
                        ]

      let loop_footer = [ LABEL (fil_false)
                        ; ADDI (arr_reg, arr_reg,
                            elemSizeToInt elem_size)
                        ; ADDI (i_reg, i_reg, 1)
                        ; J loop_beg
                        ; LABEL (loop_end)
                        ; SW (inc_reg, place, 0) (*
                            update the size of the result
                             array *)
                        ]

      arr_code
        @ get_size
        @ dynalloc (size_reg, place, tp)
        @ init_regs
        @ loop_header
        @ apply_code
        @ loop_filter
        @ loop_footer

  Replicate (n_exp, a_exp, tp, (pos1, _)) ->
      let size_reg = newReg "size" (* size of input/
          output array *)
      let a_reg    = newReg "a"    (* value of expr a *)
      let addr_reg = newReg "addr" (* address of element
          in new array *)
      let i_reg    = newReg "i"

      (* Evaluate expressions into their respective
          registers *)
      let get_replicates = compileExp n_exp vtable
          size_reg
      let eval_a_exp = compileExp a_exp vtable a_reg
      let elem_size = getElemSize tp
```

```
let safe_lab = newLab "safe"
let checksize = [ BGE (size_reg, Rzero, safe_lab)
                ; LI (Ra0, pos1)
                ; LA (Ra1, "m.BadSize")
                ; J "p.RuntimeError"
                ; LABEL (safe_lab)
                ]

let init_regs = [ ADDI (addr_reg, place, 4)
                ; MV (i_reg, Rzero)
                ]

let loop_beg = newLab "loop_beg"
let loop_end = newLab "loop_end"
let loop_header = [ LABEL (loop_beg)
                  ; BGE (i_reg, size_reg, loop_end)
                  ]

let loop_replicate = [ Store elem_size (a_reg,
   addr_reg, 0)
                     ; ADDI (addr_reg, addr_reg,
                         elemSizeToInt elem_size)
                     ]

let loop_footer =
        [ ADDI (i_reg, i_reg, 1)
        ; J loop_beg
        ; LABEL loop_end
        ]

get_replicates
 @ eval_a_exp
 @ checksize
 @ dynalloc (size_reg, place, tp)
 @ init_regs
 @ loop_header
 @ loop_replicate
 @ loop_footer

Scan (binop, acc_exp, arr_exp, tp, pos) ->
      let arr_reg  = newReg "arr"   (* address of array
         *)
      let size_reg = newReg "size"  (* size of input
```

```
    array *)
let i_reg     = newReg "ind_var"    (* loop counter
    *)
let tmp_reg   = newReg "tmp"    (* several purposes
    *)
let addr_reg = newReg "addr"   (* current result
    element adress *)
let acc_reg   = newReg "acc"    (* accumulator *)
let loop_beg = newLab "loop_beg"
let loop_end = newLab "loop_end"

let arr_code = compileExp arr_exp vtable arr_reg
let get_size = [ LW(size_reg, arr_reg, 0) ]

(* Allocate memory *)
let alloc =
      dynalloc (size_reg, place, tp)

(* Compile initial value into accumulator (will be
    updated below) *)
let acc_code = compileExp acc_exp vtable acc_reg

(* Set arr_reg to address of first element instead.
    *)
(* Set i_reg to 0. While i < size_reg, loop. *)
let loop_code =
     [ ADDI (arr_reg, arr_reg, 4)
     ; ADDI (addr_reg, place, 4)
     ; MV (i_reg, Rzero)
     ; LABEL (loop_beg)
     ; BGE (i_reg, size_reg, loop_end)
     ]
(* Load arr[i] into tmp_reg *)
let elem_size = getElemSize tp
let load_code =
     [ Load elem_size (tmp_reg, arr_reg, 0)
     ; ADDI (arr_reg, arr_reg, elemSizeToInt
         elem_size)
     ]
(* place[i] := binop(accumulator, tmp_reg) *)
let apply_code =
     applyFunArg(binop, [acc_reg; tmp_reg], vtable,
         acc_reg, pos)
```

```
let update_regs =
    [ Store elem_size (acc_reg, addr_reg, 0)
    ; ADDI (addr_reg, addr_reg, elemSizeToInt
        elem_size)
    ]

arr_code @ get_size @ alloc @ acc_code @ loop_code
   @ load_code @ apply_code @ update_regs @
    [ ADDI(i_reg, i_reg, 1)
    ; J loop_beg
    ; LABEL loop_end
    ]
```