Shrimp

an interpreted imperative programming language that wants to believe

Lorenzo Loconte

Introduction

Shrimp is a very simple and didactic imperative programming language designed during the course of *Formal Methods for Computer Science* at Universita' degli Studi di Bari Aldo Moro. **Shrimp** uses an *eager evaluation strategy*. In order to ensure that, the interpreter executes the code using the *call by value* method.

Software Modules

The program is composed by three main components:

- The **parser**
- The **optimizer**
- The **interpreter**

The **parser** takes in input the source code and convert it into an intermediate representation. The intermediate representation have the structure of a n-ary tree having the non-terminals of the grammar as internal nodes and commands, identifiers and constants on the leaves.

The **optimizer** takes in input the intermediate representation given by the parser. The result of the optimizer is an *optimized* intermediate representation. It evaluates the constant expressions (both arithmetic and boolean) that might be present in the source code and replace them with the resulting constants. The optimizer also checks for empty command blocks and useless branch statements and removes (or optimize) them.

The **interpreter** execute the semantics present in an intermediate representation. The basic idea is to use a **state** (or environment) that collects the values of the variables during the execution of the program. The result of the interpretation is the resulting state, that is a set of ground assignments to the variables. The variables can be either of type integer, boolean or array of fixed size of integers.

Functors, Applicatives and Monads

Before introducing the syntax of the programming language, let's introduce three useful classes that are widely used in the construction of an interpreter. These classes are defined on "wrapped" types, such as `Maybe`.

The `Functor` class defines the `fmap` function. The `fmap` function takes in input a simple function `a -> b` and an object of type `a` wrapped inside the functor `f`. The result of the `fmap` function is the application of the function to the "unwrapped" object inside `f`.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The `Applicative` class can be used only if `Functor` is already implemented for that polymorphic data type. It defines a function named `pure` and the `<*>` operator. The `pure` function simply takes in input an object and wraps it into an applicative `f`. The `<*>` operator is more complex. It takes in input a function `a -> b` wrapped in an applicative `f` and an object of type `a` wrapped inside the applicative `f`. The result of the `<*>` operator is the application of the "unwrapped" function to the "unwrapped" object inside `f`.

```
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The last important class is the `Monad` class. The `Monad` class can be used only if `Applicative` is already implemented for that polymorphic data type. It defines a function named `return` and the `bind` operator. The `return` function is very similar to the `pure` function defined for `Applicatives`. The difference is that `return` gives an object wrapped inside a `Monad` while `pure` gives an object wrapped inside an `Applicative`. Often, their implementation is equivalent. The `bind` operator is denoted with ≫= and takes in input an object of type `a` wrapped inside the monad `m` and a function `a -> m b`. The result of the `bind` operator is application of the function to the "unwrapped" object inside `m`.

```
class (Applicative m) => Monad m where
  return :: a -> m a
  return = pure
  (>>=) :: m a -> (a -> m b) -> m b
```

The `Monad` class is very useful because it's possible to *emulate* the behavior of an imperative programming language inside Haskell, a purely functional programming language. That is, the following code snippet ...

```
m1 >>= \a1 ->
  m2 >>= \a2 ->
    ...
      mn >>= \an ->
        f a1 a2 ... an
```

... can be simplified using the `do` construct, as in a imperative programming language.

```
do
  a1 <- m1
  a2 <- m2
  ...
  an <- mn
  f a1 a2 ... an
```

The Language Syntax

The syntax for the **Shrimp** programming language is a context-free grammar. So, it can be denoted using some kind of EBNF (Extended Backus Naur Form) as following:

```
Integer ::= [0-9]+
Identifier ::= [a-zA-Z_]+ \ Keyword
Program ::= Header "shrimp" ";" Block
Header ::= [Variable]*
Block ::= [Command]*

Variable ::= "let" Identifier "as" Type ";"
Type ::= {"int" | "bool" | "array" "[" Integer "]"}
Command ::= {Assignment | Branch | Loop | Skip} ";"

Assignment ::= Identifier
  { "=" ArithmeticExpr
  | "=" BooleanExpr
  | "[" ArithmeticExpr "]" "=" ArithmeticExpr
  }

Branch ::= "if" "(" BooleanExpr ")" "then" Block ["else" Block] "end"
Loop ::= "while" "(" BooleanExpr ")" "do" Block "end"
Skip ::= "skip"

ArithmeticExpr ::= ArithmeticTerm
                   [{"+" | "-"} ArithmeticTerm]*
ArithmeticTerm ::= ArithmeticFactor
                   [{"*" | "/" | "%"} ArithmeticFactor]*
ArithmeticFactor ::=
    Integer
  | Identifier
  | Identifier "[" ArithmeticExpr "]"
  | "-" ArithmeticExpr
  | "(" ArithmeticExpr ")"

BooleanExpr ::= ArithmeticTerm ["or" ArithmeticTerm]*
BooleanTerm ::= ArithmeticFactor ["and" ArithmeticFactor]*
BooleanFactor ::=
    "true"
  | "false"
  | Identifier
  | "not" BooleanExpr
  | ArithmetciExpr "eq" ArithmeticExpr
  | ArithmetciExpr "neq" ArithmeticExpr
  | ArithmetciExpr "lt" ArithmeticExpr
  | ArithmetciExpr "gt" ArithmeticExpr
  | ArithmetciExpr "leq" ArithmeticExpr
  | ArithmetciExpr "geq" ArithmeticExpr
  | "(" BooleanExpr ")"
```

Some of the non-terminals of this context-free grammar are reported directly in Haskell. That is, I defined an *abstract syntax tree* that also represents the intermediate representation of a program. This intermediate representation will be the result of the parser. Moreover, the presence of an intermediate representation permits us to apply optimizations *at prior* respect to the interpretation step. The following code snippet contains the definition of the *abstarct syntax tree*.

```haskell
type Program = (Header, Block)
type Block = [Command]
type Header = [Variable]

data ArithmeticExpr
  = Constant Int
  | IntegerVar String
  | ArrayVar String ArithmeticExpr
  | Add ArithmeticExpr ArithmeticExpr
  | Sub ArithmeticExpr ArithmeticExpr
  | Mul ArithmeticExpr ArithmeticExpr
  | Div ArithmeticExpr ArithmeticExpr
  | Mod ArithmeticExpr ArithmeticExpr
  | Neg ArithmeticExpr
  deriving (Eq, Show)

data BooleanExpr
  = Truth Bool
  | BooleanVar String
  | Not BooleanExpr
  | Or BooleanExpr BooleanExpr
  | And BooleanExpr BooleanExpr
  | Equal ArithmeticExpr ArithmeticExpr
  | NotEqual ArithmeticExpr ArithmeticExpr
  | Less ArithmeticExpr ArithmeticExpr
  | Greater ArithmeticExpr ArithmeticExpr
  | LessEqual ArithmeticExpr ArithmeticExpr
  | GreaterEqual ArithmeticExpr ArithmeticExpr
  deriving (Eq, Show)

data Command
  = Skip
  | ArithmeticAssignment String ArithmeticExpr
  | BooleanAssignment String BooleanExpr
  | ArrayAssignment String ArithmeticExpr ArithmeticExpr
  | Branch BooleanExpr Block Block
  | Loop BooleanExpr Block
  deriving (Eq, Show)

data Variable
  = IntegerDecl String
  | BooleanDecl String
  | ArrayDecl String Int
  deriving (Eq, Show)
```

The Parser

The parser can be viewed as a function from a string to a list of pairs of values and strings (Graham Hutton).

```haskell
newtype Parser a = Parser {unwrap :: String -> [(a, String)]}
```

Note that the parser have a special function called `unwrap` that takes the function out from the parser. The parser is implemented in a "monadic" way. That is, I implemented the following interfaces: *monad plus* and *monad alternative* as suggested in *Monadic*

Parsing in Haskell (Hutton & Meijer). In order to implement the monad interface I also implemented the *functor*, *applicative* and *monad* interfaces.

```
instance Functor Parser where
  fmap f p = Parser (\cs ->
    [(f a, cs') | (a, cs') <- unwrap p cs])
```

The functor implementation for the parser implements the `fmap` function, that is the application of a function on a wrapped parser.

```
instance Applicative Parser where
  pure a = Parser (\cs -> [(a, cs)])
  p <*> q = Parser (\cs -> concat
    [unwrap (fmap a q) cs' | (a, cs') <- unwrap p cs])
```

Moreover, the applicative implementation for the parser introduces both the function `pure` and the operator `<*>`. The `pure` function takes a simple value and wraps it into a parser. The `<*>` operator takes in input a function wrapped in a parser and another parser. The result is the application of the wrapped function onto the parser.

```
instance Monad Parser where
  return a = pure a
  p >>= f = Parser (\cs -> concat
    [unwrap (f a) cs' | (a, cs') <- unwrap p cs])
```

The `bind` operator ≫= takes as input a parser and a function that returns a parser. The result of this operator is the application of the function to the "unwrapped" parser. The application of multiple `bind` nested operations can be abbreviated using the `do` construct in Haskell.

Moreover, two other *custom* interfaces are implemented, needed for making the parser: `MonadPlus` and `MonadAlternative`. The `plus` operator defined on parsers concatenates the result of each one. Moreover, the `MonadAlternative` give us an operator `<|>` useful for combining parsers in a mutually exclusive way. The classes definition can be found in the following code snippet.

```
class (Monad m) => MonadPlus m where
  zero :: m a
  plus :: m a -> m a -> m a

class (MonadPlus m) => MonadAlternative m where
  (<|>) :: m a -> m a -> m a
  many :: m a -> m [a]
  many m = some m <|> return []
  some :: m a -> m [a]
  some m = liftA2 (:) m (many m)
  chain :: m a -> m (a -> a -> a) -> m a
  chain p o = do a <- p; rest a
    where
      rest a = (do f <- o; a' <- p; rest (f a a')) <|> return a
```

Note that the function `liftA2` is the composition of `fmap` and the applicative operator `<*>`. The functions `many` and `some` are called *combinators* and are used to define the concept of repeated parsing. The `many` function refers to zero to any number of applications of a parser while the `some` function refers to at least one to any number of applications of a parser. The implementation for the parser of the `bind` operator and the other functions described before is the following. The `chain` function is another combinator that it's used for left-associative recursion on parsers, as described by Hutton & Meijer.

```haskell
instance MonadPlus Parser where
  zero = Parser (const [])
  p `plus` q = Parser (\cs -> unwrap p cs ++ unwrap q cs)

instance MonadAlternative Parser where
(<|>) p q = Parser (\cs ->
  case unwrap (p `plus` q) cs of
    [] -> []
    (x : _) -> [x])
```

The `zero` function defines what is an empty parser. Note that in this context an empty parser is a *failed* parser, that is a parser that results from a syntactical error. The `plus` function concatenates the results of two individual parsers. Using the monadic definition of parser, it permits us to easily build a parser for each structure of the programming language, without having to handle the eventual parsing errors individually. For example, it's possible to define a parser for symbols, identifiers and integers as in the following code snippet. Moreover, since the parser implements the `MonadAlternative` interface described above, the combination of multiple parsers is straightforward and easily *parallelizable* by the compiler.

```haskell
item :: Parser Char
item = Parser (\case "" -> []; (c : cs) -> [(c, cs)])

satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do c <- item; if p c then return c else zero

token :: Parser a -> Parser a
token p = do space; v <- p; space; return v

space :: Parser String
space = many $ satisfy isSpace

identifier :: Parser String
identifier = do
  s <- token $ some $ satisfy isLetter
  if s `elem` keywords
    then zero
    else return s

constant :: Parser Int
constant = read <$> token (some $ satisfy isDigit)

char :: Char -> Parser Char
char c = satisfy (c ==)
```

That is, I firstly defined a `item` function that reads a character from the string. Then I defined a function called `satisfy` that apply a constraint to the character read by the parser. This function returns an `zero` parser if the constraint is not satisfied. In the end, using both the combinators `many` and `some`, I built parsers for identifiers, constants and also spaces that are the main blocks of the context-free grammar already defined. In the same way, it's possible to define a parser for other constructs, such as keywords and symbols. It's important to note that the identifier parser fails if a keyword is found. This is a very fundamental aspect for a programming language. So, it's not possible use variables identified by keywords.

```haskell
keyword :: String -> Parser String
keyword cs = token $ word cs

word :: String -> Parser String
word [c] = do char c; return [c]
word (c : cs) = do char c; word cs; return (c : cs)

symbol :: Char -> Parser Char
symbol c = token $ char c
```

In order to build parsers for both arithmetic and boolean expression the `<|>` operator between parsers is used. So, I defined the parser for arithmetic expressions using other sub-parsers that are used in order to maintain the operators precedence. Moreover, I used the `chain` combinator in order to build the *abstract syntax tree* by the left-associative operators of the grammar.

```haskell
arithmeticExpr :: Parser ArithmeticExpr
arithmeticExpr = chain arithmeticTerm op
  where
    op =
      do symbol '+'; return Add
        <|> do symbol '-'; return Sub

arithmeticTerm :: Parser ArithmeticExpr
arithmeticTerm = chain arithmeticFactor op
  where
    op =
      do symbol '*'; return Mul
        <|> do symbol '/'; return Div
        <|> do symbol '%'; return Mod

arithmeticFactor :: Parser ArithmeticExpr
arithmeticFactor =
  do Constant <$> constant
    <|> do
      d <- identifier
      do
        symbol '['
        k <- arithmeticExpr
        symbol ']'
        return (ArrayVar d k)
        <|> return (IntegerVar d)
    <|> do symbol '-'; Neg <$> arithmeticExpr
    <|> do symbol '('; a <- arithmeticExpr; symbol ')'; return a
```

The `chain` combinator is used in both `arithmeticExpr` and `arithmeticTerm` in order to permits to concatenate multiple addition/subtraction operators and multiple multiplication/division/modulus operators. If we didn't used the `chain` combinator but simple plain tail recursion we would have that the order of operations would be from right to left instead of left to right. The parser for boolean expressions is similar to the parser for arithmetic expressions and so it's omitted in this documentation.

In the end we have a parser for every command described in the grammar. It's important to notice that the grammar also allows for `if-then` statements (i.e. without the `else` command block). A parser that works on both `if-then-else` and `if-then` statements is implemented by combining the two individual parsers using the `<|>` operator.

Moreover, using the many combinator and the <|> operator, the parsers for both multiple commands and a single command are straightforward.

```
command :: Parser Command
command = do
  c <- assignment <|> branch <|> loop <|> skip
  symbol ';'
  return c

assignment :: Parser Command
assignment = do
  d <- identifier
  do
    symbol '='
    ArithmeticAssignment d <$> arithmeticExpr
    <|> do
      symbol '='
      BooleanAssignment d <$> booleanExpr
    <|> do
      symbol '['
      k <- arithmeticExpr
      symbol ']'
      symbol '='
      ArrayAssignment d k <$> arithmeticExpr

branch :: Parser Command
branch = do
  keyword "if"
  symbol '('
  b <- booleanExpr
  symbol ')'
  keyword "then"
  c1 <- block
  do
    keyword "else"
    c2 <- block
    keyword "end"
    return (Branch b c1 c2)
    <|> do
      keyword "end"
      return (Branch b c1 [Skip])

loop :: Parser Command
loop = do
  keyword "while"
  symbol '('
  b <- booleanExpr
  symbol ')'
  keyword "do"
  c <- block
  keyword "end"
  return (Loop b c)

skip :: Parser Command
skip = do
  keyword "skip"
  return Skip
```

Another parser that is needed for the grammar, is the parser for the variable declarations. That is, in this programming language we can declare variables only on the top of the program, and before the `shrimp` keyword. The variables can be integers, boolean or array of integers. For the declaration of array of integers, the size of the array must be fixed and a constant integer value.

```
variable :: Parser Variable
variable = do
  keyword "let"
  d <- identifier
  keyword "as"
  v <- var
  symbol ';'
  return (v d)
  where
    var =
      do keyword "int"; return IntegerDecl
        <|> do keyword "bool"; return BooleanDecl
        <|> do
          keyword "array"
          symbol '['
          n <- constant
          symbol ']'
          return (`ArrayDecl` n)
```

The parser for the entire program is defined as in the following code snippet. That is, the parser for a block of commands is defined using the `many` combinator on the parser for a `command`. In the same way, the parser for a header is defined using the `many` combinator on the parser for a `variable`. The parser for the entire parser is defined using both the parser for the header and the command block. Note that the keyword `shrimp` separates these parts of the program.

```
program :: Parser Program
program = do
  h <- header
  keyword "shrimp"
  symbol ';'
  b <- block
  return (h, b)

block :: Parser Block
block = many command

header :: Parser Header
header = many variable

parse :: String -> Result (Program, String)
parse cs = case unwrap program cs of
  [] -> Error EmptyProgram
  [(p, cs)] -> Ok (p, cs)
```

The Optimizer

The *optimization* process is an intermediate step between the parsing and the interpretation of the program itself. Currently, the main optimization step is related to the execution of constant values. That is, if an expression in a loop is defined only on constant values, it's better to optimize the computation of that expression by replacing it with its constant result. This procedure is done before the interpretation of the program. The implementation of the *optimization* step in Haskell is straightforward, due to simple recursion functions.

For example, consider the following arithmetic expression, expressed in intermediate representation, that we wish to optimize. If we apply the optimization step to this arithmetic expression, we obtain the equivalent but more efficient arithmetic expression. A very similar optimization process is also implemented on boolean expressions.

```
let expr = Div (
    (Mul (Identifier "x") (Sub (Constant 5) (Constant 3)))
    (Add (Constant 9) (Constant 1)))
```

```
let expr' = Div (Mul (Identifier "x") (Constant 2)) (Constant 10)
```

The *optimization* process also includes a basic optimization on commands such as `skip`, `if-then-else` and `while-do` statements. First of all, all the `skip` commands are removed from the intermediate representation. Moreover, if the condition of a `if-then-else` command is always *true* then the entire statement is replaced with the first block of commands. In a similar way, if the condition is always *false* then the entire statement is replaced with the second block of commands.

```
let command = Branch (Equal (Sub (Constant 1) (Constant 1)) 0)
              [Assignment "x" (Constant 1)]
              [Assignment "x" (Constant 2)]
```

```
let command' = [Assignment "x" (Constant 1)]
```

Furthermore, this approach is also used for `while-do` commands. That is, if the condition of a `while-do` command is always *false* then the entire statement is completely removed.

```
let command = Loop (Equal (Sub (Constant 1) (Constant 1)) 1)
              [Assignment "x" (Add (IntegerVar "x") (Constant 1))]
```

```
let command' = Skip
```

However, if the condition is always *true*, an exception named *Infinite Loop* is raised. In other words, the optimizer is capable of detecting trivial infinite loops and prevents the interpretation of such programs. For example, the following loop program's intermediate representation is not interpreted, because the *Infinite Loop* exception is raised *before* the interpretation step.

```
let command = Loop (Equal (Sub (Constant 1) (Constant 1)) 0)
              [Assignment "x" (Add (IntegerVar "x") (Constant 1))]
```

These kinds of code optimizations are very basic and don't consider the *expected state* of the program at a certain point, in order to apply more advanced kinds of optimizations (such as expressions simplification or variables pruning).

The Interpreter

Finally, the obtained intermediate representation is interpreted by the interpreter. The interpreter scan the intermediate representation using *depth-first traversal* on the n-ary tree that compose the intermediate representation itself. Before introducing the execution of the interpreter on an intermediate representation, some useful data types are defined in order to handle errors.

```haskell
data Exception
  = EmptyProgram
  | InfiniteLoop
  | DivisionByZero
  | UndeclaredVariable String
  | MultipleVariable String
  | TypeMismatch String
  | OutOfBound String Int
  | InvalidSize String

instance Show Exception where
  show EmptyProgram = "Empty Program"
  show InfiniteLoop = "Infinite Loop"
  show DivisionByZero = "Division By Zero"
  show (UndeclaredVariable d) = "Undeclared Variable" ++ ": " ++ d
  show (MultipleVariable d) = "Multiple Variable" ++ ": " ++ d
  show (TypeMismatch d) = "Type Mismatch" ++ ": " ++ d
  show (OutOfBound d i) = "Out Of Bound" ++ ": " ++ d ++ " at " ++ show i
  show (InvalidSize d) = "Invalid Size" ++ ": " ++ d

data Result a = Ok a | Error Exception

instance Functor Result where
  fmap f (Ok v) = Ok (f v)
  fmap _ (Error e) = Error e

instance Applicative Result where
  pure v = Ok v
  (<*>) (Ok f) (Ok v) = Ok (f v)
  (<*>) (Error e) _ = Error e
  (<*>) _ (Error e) = Error e

instance Monad Result where
  (>>=) (Ok v) f = f v
  (>>=) (Error e) _ = Error e

exception :: Exception -> a
exception e = errorWithoutStackTrace $ show e
```

As one can see, the `Result` data type is a polymorphic type that can be either a `Ok` or a `Error`. The `Error` type also encapsulate an exception, one of the listed above. Moreover functor, applicative and monad interfaces are implemented in order to apply the needed operators directly on intermediate results. The functor, applicative and monad interfaces implementation is as in the `Maybe` data type. In the end, the `exception` function is useful in order to print the error and stop the program without printing the stack trace.

In order to interpret the intermediate representation, two sub-interpreters are implemented. The first interpreter loads all the variables in the variables declaration section of the program. The result of the variables loading is a consistent state, viewed as a dictionary with key as the identifier of a variable and value the effective value of the variable at a certain point of the program. The value of a variable can be either an integer, a boolean value or a vector of integers.

```
initialize :: State -> Header -> State
initialize s [] = s
initialize s ((IntegerDecl d) : hs) =
  case search d s of
    Just _ -> exception (MultipleVariable d)
    Nothing -> initialize s' hs
      where
        s' = insert d (IntegerValue 0) s
initialize s ((BooleanDecl d) : hs) =
  case search d s of
    Just _ -> exception (MultipleVariable d)
    Nothing -> initialize s' hs
      where
        s' = insert d (BooleanValue False) s
initialize s ((ArrayDecl d n) : hs) =
  case search d s of
    Just _ -> exception (MultipleVariable d)
    Nothing ->
      if n > 0
        then initialize s' hs
        else exception (InvalidSize d)
      where
        s' = insert d (ArrayValue (zeroArray n)) s
```

The second interpreter executes the commands specified in the commands section of the command (after the `shrimp` keyword). So, it interprets every command specified in the grammar, that are `skip`, various kinds of `assignments`, `if-then-else` and `if-then` statements and the `while-do` statement.

```
execute :: State -> Block -> State
execute s [] = s
execute s (Skip : cs) = execute s cs
execute s ((ArithmeticAssignment d a) : cs) =
  case search d s of
    Just (IntegerValue _) ->
      case evalArithmetic s a of
        Ok v -> execute s' cs
          where
            s' = insert d (IntegerValue v) s
        Error e -> exception e
    Just (BooleanValue _) -> exception (TypeMismatch d)
    Just (ArrayValue _) -> exception (TypeMismatch d)
    Nothing -> exception (UndeclaredVariable d)
execute s ((BooleanAssignment d b) : cs) =
  case search d s of
    Just (BooleanValue _) ->
      case evalBoolean s b of
        Ok t -> execute s' cs
          where
            s' = insert d (BooleanValue t) s
        Error e -> exception e
    Just (IntegerValue _) -> exception (TypeMismatch d)
    Just (ArrayValue _) -> exception (TypeMismatch d)
    Nothing -> exception (UndeclaredVariable d)
```

```
execute s ((ArrayAssignment d k a) : cs) =
  case search d s of
    Just (ArrayValue vs) ->
      case (evalArithmetic s a, evalArithmetic s k) of
        (Ok v, Ok i) -> execute s' cs
          where
            s' = insert d (ArrayValue vs') s
            vs' = case writeArray i v vs of
              Just vs' -> vs'
              Nothing -> exception (OutOfBound d i)
        (Error e, _) -> exception e
        (_, Error e) -> exception e
    Just (IntegerValue _) -> exception (TypeMismatch d)
    Just (BooleanValue _) -> exception (TypeMismatch d)
    Nothing -> exception (UndeclaredVariable d)
execute s ((Branch b cs' cs'') : cs) =
  case evalBoolean s b of
    Ok True -> execute s (cs' ++ cs)
    Ok False -> execute s (cs'' ++ cs)
    Error e -> exception e
execute s (c@(Loop b cs') : cs) =
execute s (Branch b (cs' ++ [c]) [Skip] : cs)
```

The implementation of the evaluation of arithmetic expressions make *heavily* use of the methods exposed by the applicative interface. Moreover, the `liftA2` function is used in order to make the evaluation functions implementation more compact and readable. It's important to notice that arithmetic expressions evaluation includes reading from variables and accessing to arrays by an index that is the evaluation of another arithmetic expression. Moreover, the current implementation includes the evaluation of the `minus` unary operator. In this implementation, I also used the functions `safeDiv`, `safeMod` and `seqM2`. In other words, the `seqM2` is a custom function that implements binary sequencing, similarly to the `bind` operator.

```
evalArithmetic :: State -> ArithmeticExpr -> Result Int
evalArithmetic _ (Constant v) = Ok v
evalArithmetic s (IntegerVar d) =
  case search d s of
    Just (IntegerValue v) -> Ok v
    Just (BooleanValue _) -> exception (TypeMismatch d)
    Just (ArrayValue _) -> exception (TypeMismatch d)
    Nothing -> exception (UndeclaredVariable d)
evalArithmetic s (ArrayVar d k) =
  case search d s of
    Just (ArrayValue vs) ->
      case evalArithmetic s k of
        Ok i ->
          case readArray i vs of
            Just v -> Ok v
            Nothing -> Error (OutOfBound d i)
        Error e -> exception e
    Just (IntegerValue _) -> exception (TypeMismatch d)
    Just (BooleanValue _) -> exception (TypeMismatch d)
    Nothing -> exception (UndeclaredVariable d)
evalArithmetic s (Add a1 a2) = liftA2 (+) v1 v2
  where
    v1 = evalArithmetic s a1
    v2 = evalArithmetic s a2
```

```haskell
evalArithmetic s (Sub a1 a2) = liftA2 (-) v1 v2
  where
    v1 = evalArithmetic s a1
    v2 = evalArithmetic s a2
evalArithmetic s (Mul a1 a2) = liftA2 (*) v1 v2
  where
    v1 = evalArithmetic s a1
    v2 = evalArithmetic s a2
evalArithmetic s (Div a1 a2) = seqM2 safeDiv v1 v2
  where
    v1 = evalArithmetic s a1
    v2 = evalArithmetic s a2
evalArithmetic s (Mod a1 a2) = seqM2 safeMod v1 v2
  where
    v1 = evalArithmetic s a1
    v2 = evalArithmetic s a2
evalArithmetic s (Neg a) = negate <$> v
  where
    v = evalArithmetic s a
```

The implementations of `saveDiv`, `safeMod`, and `seqM2` functions are listed in the following code snippet. Note that `saveDiv` and `safeMod` are special functions that handles divisions by zero at runtime.

```haskell
safeDiv :: Int -> Int -> Result Int
safeDiv _ 0 = Error DivisionByZero
safeDiv u v = Ok (div u v)

safeMod :: Int -> Int -> Result Int
safeMod _ 0 = Error DivisionByZero
safeMod u v = Ok (mod u v)

seqM2 :: (Monad m) => (a -> b -> m c) -> m a -> m b -> m c
seqM2 f x y = join $ liftA2 f x y

join :: (Monad m) => m (m a) -> m a
join m = m >>= id
```

The implementation of the evaluation of boolean expressions is very similar to the one for arithmetic expressions, so it's omitted.

Conclusion

The monadic implementation of the parser give us a very simple way of concatenating and combining multiple parsers of sub-grammars. Also, it's very easy to extend the grammar of the language in order to include other commands or statements. Moreover, the use of an intermediate representation permits us to apply post-processing and optimizations and make the interpretation itself straightforward and more efficient.

Future works may include the introduction of other types for variables. Another extension of this work consists of adding useful information about the parsing errors, i.e. missing tokens and relevant row and column locations in the source code where the error occurred. However, other improvements can be done in the optimization step. That is, one can implement more "aggressive" optimizations of the intermediate representation based on the expected state of memory of the program during its execution.

Running an example

First of all, navigate to the project directory. Then, open `ghci` and load the needed modules as following.

```
:load Main.hs Shrimp.hs
```

Finally, run the main program and insert the path of a source file in the `examples` directory (for example `primes.shr`).

```
*Main> main
The Shrimp Interpreter
Insert the path of the source file:
examples/primes.shr
Memory state:
    "n": int = 49
    "i": int = 15
    "j": int = 0
    "stop": bool = False
    "primes": array[15] = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```