Imperative interpreter

Interpreters

An interpreter is a program that directly analyse and execute instructions written in a certain programming language. In this project was built a simple interpreter in Haskell. IMPterpreter, as the name suggests, interprets a simple imperative programming language called IMP.

The basic constructs of IMP are:

- **skip**: does nothing
- **assignment** (written as ' := ')**:** evaluate an expression and assign it to a variable
- **if {} else {}**: Performs the conditional selection between two paths
- **while**: Repeat a sequence of instructions until a boolean condition is satisfied
- **foreach**: Loops over all the values of an array without access based on an integer index

Grammar

The grammar of the language interpreted by the provided code is an extension of the original grammar of the IMP language, it adds arrays, records and the foreach loop construct.
The extended grammar is the following:

ARRAYS

```
arrayElement ::= <integer> | <integer>, <arrayElement>
arrayLiteral ::= [ <arrayElement> ]
arrayAccess  ::= <identifier> [ <integer> ]
```

RECORDS

```
recordElement ::= <integer>
                | <identifier>:<integer>, <recordElement>
recordLiteral ::= { <recordElement> }
recordAccess  ::= <identifier>.<identifier>
```

ARITHMETIC EXPRESSIONS

```
aexp        ::= <aterm> + <aexp> | <aterm> - <aexp> | <aterm>
aterm       ::= <afactor> * <aterm>
              | <afactor> / <aterm>
              | <afactor>
afactor     ::= (<aexp>) | <integer> | <identifier>
              | <arrayAccess> | <recordAccess>
```

BOOLEAN EXPRESSIONS

```
bexp        ::= <bterm> OR <bexp> | <bterm>
bterm       ::= <bfactor> AND <bterm> | <bfactor>
bfactor     ::= true | false | !<bfactor> | (bexp)
bcomparison ::= <aexp> = <aexp> | <aexp> ≠ <aexp>
              | <aexp> ≤ <aexp> | <aexp> < <aexp>
              | <aexp> ≥ <aexp> | <aexp> > <aexp>
```

COMMAND EXPRESSIONS

```
program     ::= <command> | <command> <program>
command     ::= <assignment> | <ifThenElse> | <while> | skip;
assignment  ::= <identifier> := <aexp>;
              | <identifier> := <arrayLiteral>
              | <arrayAccess> := <aexp>
              | <identifier> := recordLiteral
              | <recordAccesss> := <aexp>
ifThenElse  ::= if (<bexp>) { <program> }
              | if (<bexp>) {<program>} else {<program>}
while       ::= while (<bexp>) {<program>}
foreach     ::= foreach <identifier> in <identifier> {<program>}
```

For the assignment case we also need to deal with the declaration of array and records and with the setting of a specific value of said structures.

Architecture

The main components of the project are:

- **parser:** parse and evaluate the program string
- **environment:** store variables and structures and manage the access
- **main:** provide interactive text interface and communicate with the previous components to execute the programs

Parser

The parser is a function declared as follows:

```
newtype Parser a = P (Env -> String -> [(Env, a, String)])
```

From this declaration we notice that it uses as input the environment type and the output includes the environment type; this allows to evaluate the program considering the values of the variables stored in the environment given as input, the assignments are the only instructions that modify the environment.

Parser as functor, applicative and monad

```
instance Functor Parser where
  -- fmap :: (a -> b) -> Parser a -> Parser b
  fmap g p = P(
    \env input -> case parse p env input of
        [] -> []
        [(env, v, out)] -> [(env, g v, out)])

instance Applicative Parser where
  -- pure :: a -> Parser a
  pure v = P (\env input -> [(env, v, input)])

  -- <*> :: Parser (a -> b) -> Parser a -> Parser b
  pg <*> px = P(
        \env input -> case parse pg env input of
            [] -> []
            [(env, g, out)] -> parse (fmap g px) env out)

instance Monad Parser where
  -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
  p >>= f = P(
        \env input -> case parse p env input of
            [] -> []
            [(env, v, out)] -> parse (f v) env out)
```

Implementing the parser as instance of the type-classes: Functor, Applicative and Monad allows us to define parsing operations in a way that is type - safe and still pure functional programming, but that resembles to a sequential programming style, that, in this case, is simpler because of the sequential nature of the operation of parsing a string character by character.

Alternative parser

```
instance Alternative Parser where
  -- empty :: Parser a
  empty = P (\env input -> [])

  -- (<|>) :: Parser a -> Parser a -> Parser a
  p <|> q = P(
        \env input -> case parse p env input of
            [] -> parse q env input
            [(env, v, out)] -> [(env, v, out)])
```

This construct allows defining alternative approaches for the parsing of programming language structures, for example arithmetic expression terms can be multiplications or integer divisions of terms.

Token parsing

To correctly parse the language we need to start from parsing a single char, so the first Parser is: `item .` It consumes the first character of the string and fails with empty strings.

```
item :: Parser Char
item = P(
    \env input -> case input of
        [] -> []
        (x : xs) -> [(env, x, xs)])
```

The other basic primitive is `sat`, it succeeds if the given char satisfies the predicate `p` .

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do
    -- item :: Parser Char
    -- x <- item is a function from char to Parser Char
    x <- item
    if p x then return x else empty -- Parser Char
```

Now is possible to define the following parsers:

- digit: parses a digit
- space: parses a single space or newline
- char x: parses a specific char x

- ident: based on letter, lower, upper and alphanum
- token p: parses a specific value p and ignores spaces
- symbol s: parses a specific symbol token s

Parsing of records and arrays

All the parsers are based on the listed primitives, as an important example we provide the parsing of the declaration of arrays or records, that are based on the syntax where assignment and declaration contextually happen using a literal string.

Examples

- array_example := [1, 5, -6]
- record_example := { first: 5, second: 7 }

```
arrayLiteral :: Parser [(Int, Int)]
arrayLiteral =
    do
        symbol "["
        es <- arrayElements
        symbol "]"
        return (zip [0..] es)

arrayElements :: Parser [Int]
arrayElements =
    do
        x <- integer
        symbol ","
        xs <- arrayElements
        return (x:xs)
    <|> do
        x <- integer
        return [x]
```

```
recordLiteral :: Parser [(String, Int)]
recordLiteral =
    do
        symbol "{"
        xs <- recordElements
        symbol "}"
        return xs
recordElements :: Parser [(String, Int)]
recordElements =
```

```
    do
        f <- identifier
        symbol ":"
        v <- integer
        symbol ","
        xs <- recordElements
        return ((f, v):xs)
    <|> do
        f <- identifier
        symbol ":"
        v <- integer
        return [(f, v)]
```

Parsing of control structures

As an example of parsing a control structure we provide the `foreach` construct.

It is an additional construct that allows to iterate over an entire array without using integer indexes.

Example

```
x := [1, 5, -3, 0];
y := 0;
foreach xi in x { y := y + xi; }
Result: y := 3
```

Parser code

```
foreach :: Parser String
foreach = do
    w <- consumeForEach
    repeatLoop w
    -- parse the string until the identifier of the collection
    symbol "foreach"
    element <- identifier
    symbol "in"
    i <- identifier



    -- ignore the rest
    symbol "{"
```

```
    parseProgram
    symbol "}"
    -- store the value of the collection to iterate
    collection <- readVariable ("_" ++ i)
    -- reconstruct the code
    repeatLoop w
    -- iterate on the collection
    foreachSteps collection
    -- remove the element variable from the environment
    updateEnvRemove element
    -- empty the code after last iteration
    consumeForEach
    return ""

foreachSteps :: Maybe VarType -> Parser String
foreachSteps Nothing = return ""
foreachSteps (Just EmptyArray) = return ""
foreachSteps (Just (ArrayElement index arrayValue xs)) = do
    w <- consumeForEach
    repeatLoop w
    -- get the identifier of the iterator variable
    symbol "foreach"
    element <- identifier
    -- ignore the collection part since we already have it as a
parameter
    symbol "in"
    identifier
    -- add the iterator variable to the environment
    updateEnv Variable { name = element, value = Just (IntType
arrayValue) }
-- execute the loop body
    symbol "{"
    program
    symbol "}"
    -- repeat the loop code
    repeatLoop w
    -- iterate
    foreachSteps xs
```

Environment

The Environment is represented as a list of variables, the variables represents the integer values or the structures (arrays and records) that can be defined in the program.
The type of the variable value is declared as follows

```
data VarType = IntType (Maybe Int)
  | EmptyArray | ArrayElement Int (Maybe Int) (Maybe VarType)
  | EmptyRecord | RecordElement String (Maybe Int) (Maybe VarType)
```

The data type `VarType` describes how a value of a variable should be constructed.

For simple integer variables we define the constructor `IntType`,
while for the structures it is based on a recursive definition.

The recursive structure emulates the behavior of a list because it involves a constructor that represents an Empty list and a recursive term that allows adding more elements to the pseudo - list.

We used this strategy instead of a plain haskell list because in this way we have a " heterogeneous collection ". It means that in the same type we can collect scalar values, arrays, singleton arrays (e.g. [3]) or records.

An array element is composed of two integer values, the first one is the value index, that is useful in retrieving and modifying values, while the second one is the integer value.

The same approach is implemented for the records, but in this case the indexes are of type String because the access is associative (by field name).

All the integer values (except indexes) are wrapped in a Maybe type in order to enable the structures also to the ' `Nothing` ' value.

The `VarType` is wrapped in the `Variable` type that associates to each variable a name.

The value of a variable is wrapped in a `Maybe` type too; this is useful in the search of a variable, in particular ' `Nothing` ' is returned in the error case (not finding the variable) and in the recursion base case.

```
data Variable = Variable {name :: String, value :: Maybe VarType}
```

The access to the environment is handled by the functions getValue and setValue respectively for reading and writing of variables.

```
getValue :: Maybe VarType -> String -> Maybe Int

getValue Nothing _ = Nothing

getValue (Just (IntType n)) _ = n

getValue (Just EmptyArray) _ = Nothing
getValue ( Just (ArrayElement index value xs)) searchIndex
  | index == read searchIndex = value
  | otherwise = getValue xs searchIndex

getValue (Just EmptyRecord) _ = Nothing
getValue ( Just(RecordElement field value xs)) searchField
  | field == searchField = value
  | otherwise = getValue xs searchField
```

```
setValue :: Maybe VarType -> String -> Maybe Int -> Maybe VarType

setValue (Just (IntType n)) _ newValue = Just (IntType newValue)

setValue (Just EmptyArray) _ _ = Just EmptyArray
setValue (Just (ArrayElement index value xs)) searchIndex newValue
  | index == read searchIndex = Just (ArrayElement index newValue
xs)
  | otherwise = Just (ArrayElement index value (setValue xs
searchIndex newValue))

setValue (Just EmptyRecord) _ _ = Just EmptyRecord
setValue (Just (RecordElement field value xs))
         searchField newValue
  | field == searchField = Just (RecordElement field newValue xs)
  | otherwise = Just (RecordElement field value
    (setValue xs searchField newValue))
```

Examples

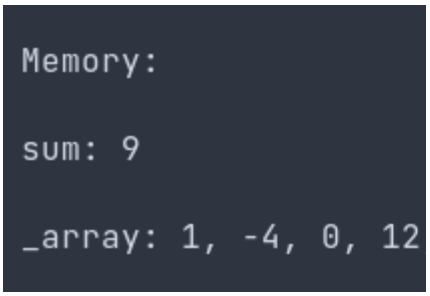
- **Array sum:** Compute the sum of the components of an array

```
sum := 0

array := [1, -4, 0, 12]

foreach element in array { sum := sum + element; }
```

Result



- **Complex numbers sum:** Compute the sum of two complex numbers represented by records with two fields: real and im

```
first := { real: 5, im: 8 };

second := { real: 4, im: 1};

sum := { real: 0, im: 0 };

sum.real := first.real + second.real;

sum.im := first.im + second.im;
```

Result