PFL - Pretty Fun Language

Welcome to the official documentation of **Pretty Fun Language** (or PFL for short), which is a small imperative programming language developed in Haskell.

Class: 3LEIC11
Group: T11_G04

Student ID	Student Name	Participation
202108691	João Afonso Viveiros	50%
202108701	Tomás Sucena Lopes	50%

Table of Contents

Development

PFL is a hybrid programming language, meaning that it is simultaneously **compiled** and **interpreted**. Its code is converted into a series of intermediate instructions, known as bytecode, that are then executed by PFL's virtual machine. As such, running PFL is a four-phase process:

1. **Lexical analysis -** The source code is divided into **tokens**, which are groups of characters that have a collective meaning. Examples of tokens include variables, keywords, and literals.
2. **Parsing -** The list of tokens is analyzed to form syntax trees, which are the data structures that confer meaning to the ...
3. **Compilation -** The statements and expressions are compiled into PFL's bytecode.
4. **Execution -** The bytecode is interpreted by the virtual machine, thus producing the actual output of the code.

Lexical analysis

The first phase of the compilation process is **lexing**, which consists of reading the code input by the programmer and dividing it into semantic atoms - **tokens**.

Given there are several types of tokens, we created the following data structure to more easily represent them:

```haskell
data Token =
  Var String    -- variables

  -- values
  | I Integer   -- integers
  | B Bool      -- booleans

  -- symbols
  | Semicolon   -- ';'
  | LParen      -- '('
  | RParen      -- ')'

  -- operators
  | Assign      -- ':='
  | Add         -- '+'
  | Mult        -- '*'
  | Sub         -- '-'
  | Le          -- '<='
  | IEqu        -- '=='
  | Not         -- 'not'
  | BEqu        -- '='
  | And         -- 'and'
  | Or          -- 'or'

  -- keywords
  | If          -- 'if'
  | Then        -- 'then'
  | Else        -- 'else'
  | While       -- 'while'
  | Until       -- 'until'
  | Do          -- 'do'

  deriving Show
```

Due to Haskell's pattern matching, lexing the source code is as simple as examining the current character and stepping into the function that creates the adequate token. Below are a few examples:

- If the character is a digit, we are lexing an **integer**. So, we sequentially analyze the next characters until the current character is no longer a digit, at which point we convert the substring into the corresponding integer value.

```haskell
lexDecNumber :: String -> (Token, String)
lexDecNumber s = (Token.I (read number :: Integer), s')
  where
    (number, s') = break (not . isDigit) s
```

```haskell
...

lexer :: String -> [Token]
lexer s@(c:_)
  | isDigit c = token : lexer s'
  where (token, s') = lexDecNumber s
```

- If the character is a lowercase letter, we are either lexing a **variable** or a reserved **keyword** (like *if*, *else*, etc.). To find out which of those it is, we sequentially read the next characters until the current character is neither a letter nor a digit. Then, we compare the substring with the language keywords to see if it is one.

```haskell
lexWord :: String -> (Token, String)
lexWord s = (token, s')
  where
    (word, s') = break (not . isAlphaNum) s
    token = case word of
      "not" -> Token.Not
      "and" -> Token.And
      "or" -> Token.Or
      "if" -> Token.If
      "then" -> Token.Then
      "else" -> Token.Else
      "while" -> Token.While
      "until" -> Token.Until
      "do" -> Token.Do
      "True" -> Token.B True
      "False" -> Token.B False
      _ -> Token.Var word

...

lexer :: String -> [Token]
lexer s@(c:_)
  | isAlpha c = token : lexer s'
  where (token, s') = lexWord s
```

Note: In PFL, variable names can contain digits, but they cannot start with one. For example, *var1* is a valid variable name, whereas *1var* is not.

- If the character is a symbol, then it could either be an **operator** or a language **symbol** (parenthesis or semicolon).

```haskell
lexer :: String -> [Token]
-- symbols
lexer (';':s) = Token.Semicolon:(lexer s)
lexer ('(':s) = Token.LParen:(lexer s)
lexer (')':s) = Token.RParen:(lexer s)
```

```
-- operators
lexer (':':'=':s) = Token.Assign:(lexer s)
lexer ('<':'=':s) = Token.Le:(lexer s)
lexer ('=':'=':s) = Token.IEqu:(lexer s)
lexer ('+':s) = Token.Add:(lexer s)
lexer ('*':s) = Token.Mult:(lexer s)
lexer ('-':s) = Token.Sub:(lexer s)
lexer ('=':s) = Token.BEqu:(lexer s)
```

The source code of the lexer can be found here.

Parsing

After lexing the source code, the next step is to **parse** the list of tokens. In computer science, parsing refers to building one or more tree structures, usually known as **abstract syntax trees** (or ASTs for short), that reflect the grammar of a language.

The simplest building blocks in PFL are **expressions**, which are units of code that represent values within the programming language. They come in two flavors: **arithmetic**, which evaluate to an integer, and **boolean**, which evaluate to a boolean. We defined their ASTs like so:

```
-- arithmetic expressions
data Aexp =
  I Integer             -- constant
  | Var String          -- variables
  | Add Aexp Aexp       -- addition
  | Mult Aexp Aexp      -- multiplication
  | Sub Aexp Aexp       -- subtraction
  deriving Show

-- boolean expressions
data Bexp =
  B Bool                -- constants
  | Lt Aexp Aexp        -- integer less than
  | Le Aexp Aexp        -- integer less or equal
  | Gt Aexp Aexp        -- integer greater than
  | Ge Aexp Aexp        -- integer greater or equal
  | IEqu Aexp Aexp      -- integer equality
  | Not Bexp            -- negation
  | BEqu Bexp Bexp      -- boolean equality
  | And Bexp Bexp       -- logical AND
  | Or Bexp Bexp        -- logical OR
  deriving Show
```

In addition to expressions, there are also **statements**, which are groups of expressions and/or other statements that have a particular purpose. In PFL, there are three types of statements: **assignments**, **conditional branches** and **loops**. Their ASTs are the following:

```
-- statements
data Stm =
  Assign String Aexp       -- integer assignment
  | If Bexp [Stm] [Stm]    -- if
  | While Bexp [Stm]       -- loops
  deriving Show
```

Given the recursive nature of our ASTs, we implemented a Recursive Descent parser. We researched the topic in a book named "Crafting Interpreters", by Robert Nystrom, and took inspiration from the 6th chapter, wherein Nystorm explains in great detail how to design a top-down parser.

Succintly, our parser attempts to parse the expressions with the highest precedence and works its way down from there. In practice, this means that expressions are parsed in the following order:

- **Arithmetic -** literals (integers and variables) > unary operators > multiplications > sums = subtractions
- **Boolean -** literals ('True' and 'False') > integer comparisons > unary operators ('not') > logical conjunction ('and') > logical disjuntion ('or')

For instance, take an arithmetic expression. The steps the parser takes to analyze it are listed below:

1. The parser starts by executing `parseAexp`, which is a function that parses arithmetic expressions.

```
-- Parse an arithmetic expression.
parseAexp :: [Token] -> (Maybe Aexp, [Token])
parseAexp tokens = parseTerm (Nothing, tokens)
```

2. `parseAexp` calls `parseTerm`, which is used to parse sums and subtractions.

```
-- Parse terms.
parseTerm :: (Maybe Aexp, [Token]) -> (Maybe Aexp, [Token])

-- addition
parseTerm (Just lhs, Token.Add:tokens) =
    case parseFactor (Nothing, tokens) of
      (Just rhs, tokens') -> parseTerm (Just (AST.Add lhs rhs), tokens')
      _ -> error "Parse error - expected an arithmetic expression after '+'"

-- subtraction
parseTerm (Just lhs, Token.Sub:tokens) =
    case parseFactor (Nothing, tokens) of
      (Just rhs, tokens') -> parseTerm (Just (AST.Sub lhs rhs), tokens')
      _ -> error "Parse error - expected an arithmetic expression after '-'"

parseTerm (Nothing, tokens) =
  case parseFactor (Nothing, tokens) of
    (Nothing, tokens') -> (Nothing, tokens')
    (exp, tokens') -> parseTerm (exp, tokens')
```

```haskell
parseTerm (exp, tokens) = (exp, tokens)
```

3. `parseTerm` runs `parseFactor`, which is a function that parses multiplications.

```haskell
-- Parse factors.
parseFactor :: (Maybe Aexp, [Token]) -> (Maybe Aexp, [Token])

-- multiplication
parseFactor (Just lhs, Token.Mult:tokens) =
  case parseUnaryA (Nothing, tokens) of
    (Just rhs, tokens') -> parseFactor (Just (AST.Mult lhs rhs), tokens')
    _ -> error "Parse error - expected an arithmetic expression after '*'"

parseFactor (Nothing, tokens) =
  case parseUnaryA (Nothing, tokens) of
    (Nothing, tokens') -> (Nothing, tokens')
    (exp, tokens') -> parseFactor (exp, tokens')
parseFactor (exp, tokens) = (exp, tokens)
```

4. `parseFactor` invokes `parseUnary`, which, as the name implies, is used to parse unary expressions.

```haskell
-- Parse unary arithmetic expressions.
parseUnaryA :: (Maybe Aexp, [Token]) -> (Maybe Aexp, [Token])
parseUnaryA (_, Token.Sub:tokens) =
  case parsePrimaryA tokens of
    (Just exp, tokens') -> parseUnaryA (Just (AST.Sub (AST.I 0) exp), tokens')
    _ -> error "Parse error - expected an arithmetic expression after '-'"
parseUnaryA (Nothing, tokens) = parsePrimaryA tokens
parseUnaryA (exp, tokens) = (exp, tokens)
```

5. `parseUnary` calls `parsePrimary` to obtain the highest priority expression.

```haskell
-- Parse primary arithmetic expressions.
parsePrimaryA :: [Token] -> (Maybe Aexp, [Token])
parsePrimaryA ( (Token.I i):tokens ) = (Just (AST.I i), tokens)
parsePrimaryA ( (Token.Var var):tokens ) = (Just (AST.Var var), tokens)
parsePrimaryA ( Token.LParen:tokens ) =
  case parseAexp tokens of
    (exp, Token.RParen:tokens') -> (exp, tokens')
    _ -> error "Parse error - expected a closing ')'."
parsePrimaryA tokens = (Nothing, tokens)
```

6. Finally, the parser descends the parse tree by going backwards in the chain of function calls.

As such, this approach eliminates any conflict that could arise from parsing expressions with different precedences.

Parsing statements is similar to parsing expressions, except we do not have to be concerned with precedence. In fact, it is analogous to expressing the syntax of the statement in code. As an explanatory example, take the parsing of assignment statements:

```
-- Parse a statement or a group of statements delimited by parenthesis.
parseStm :: ([Stm], [Token]) -> ([Stm], [Token])

-- assignment
parseStm ([], (Token.Var var):(Token.Assign):tokens) =
  case parseAexp tokens of
    (Just exp, Token.Semicolon:tokens') -> ( [AST.Assign var exp], tokens')
    (_, Token.Semicolon:tokens') -> error "Parse error - expected an arithmetic
expression"
    _ -> error "Parse error - expected a semicolon after an assignment"
```

In accordance with the syntax of an assignment, the parser looks for the name of a variable followed by the assignment operator. Provided it finds them, it attempts to parse the arithmetic expression that comes after the operator. If it succeeds, that means the statement was correctly formed, thus a suitable AST is generated, otherwise an error is thrown.

Compilation

Upon obtaining the ASTs, the next step is **compiling** them into proper instructions, which will then be executed in the final phase.

Instead of targetting specific hardware i.e. compiling to architecture dependent instructions, we opted to compile to intermediate instructions - **bytecode** - which are then interpreted by PFL's **virtual machine**. This approach ensures PFL is a portable language, albeit at the cost of performance.

The instructions supported by PFL's virtual machine are as follows:

```
data Inst =
  Push Integer          -- push an integer value to the stack
   | Add                -- add two integers and push the result to the stack
   | Mult               -- multiply two integers and push the result to the stack
   | Sub                -- subtract two integers and push the result to the stack
   | Tru                -- push 'True' to the stack
   | Fals               -- push 'False' to the stack
   | Equ                -- test if two values are equal and push the result to the
stack
   | Le                 -- test if an integer is less or equal than another and
push the result to the stack
   | And                -- perform a logical AND between two booleans and push the
result to the stack
   | Neg                -- negate a boolean and push the result to the stack
   | Fetch String       -- fetch the value of a variable and push it to the stack
   | Store String         -- pop the value on top of the stack and store it in a
variable
   | Noop                 -- do nothing
   | Branch Code Code     -- conditional branch
   | Loop Code Code       -- loop
  deriving Show
```

Given there are arithmetic and boolean expressions, we designed `compA` and `compB` for compiling them, respectively. Despite being different functions, they follow the same principles: compile the base cases and, for the remaining expressions, recursively compile their arguments.

For arithmetic expressions, the base cases are integers and variables. As such, compiling any other arithmetic expression, which represent binary operations, becomes a matter of recursively compiling its operands. This behaviour is illustrated below:

```
-- Compiles an arithmetic expression
compA :: Aexp -> Code
compA (AST.I i) = [Inst.Push i]
compA (AST.Var s) = [Inst.Fetch s]
compA (AST.Add lhs rhs) = (compA rhs) ++ (compA lhs) ++ [Inst.Add]
compA (AST.Mult lhs rhs) = (compA rhs) ++ (compA lhs) ++ [Inst.Mult]
compA (AST.Sub lhs rhs) = (compA rhs) ++ (compA lhs) ++ [Inst.Sub]
```

The same is applicable in boolean expressions, except the base cases are the boolean literals 'True' and 'False', as can be seen below:

```
compB (AST.B True) = [Inst.Tru]
compB (AST.B False) = [Inst.Fals]
compB (AST.Le lhs rhs) = (compA rhs) ++ (compA lhs) ++ [Inst.Le]
compB (AST.Not exp) = (compB exp) ++ [Inst.Neg]
...
```

Using the aforementioned expression compilation functions, we created the driver function of the compilation function - `compile`. It converts a list of statements into a program by successively compiling each statement into its intermediate representation until there are no more left.

```
-- Compiles a program i.e. a list of statements.
compile :: Program -> Code
compile [] = []
compile ( (AST.Assign var exp):xs ) = (compA exp) ++ [Inst.Store var] ++ (compile
xs)
compile ( (AST.If cond c1 []):xs ) = (compB cond) ++ [Inst.Branch (compile c1)
[Inst.Noop] ] ++ (compile xs)
compile ( (AST.If cond c1 c2):xs ) = (compB cond) ++ [Inst.Branch (compile c1)
(compile c2) ] ++ (compile xs)
```

```
compile ( (AST.While c1 c2):xs ) = [Inst.Loop (compB c1) (compile c2) ] ++
(compile xs)
```

Execution

The final phase of the compilation process is the **execution** of the bytecode. This process is carried out by PFL's **virtual machine**, which can be identified by the tuple $(code, stack, state)$.

- **Code** is a list containing the bytecode instructions to be executed.
- **Stack** is the virtual machine's stack.
- **State** is a map that represents the program's internal state i.e. its variables and respective values.

Values

In order for a virtual machine to function, it is necessary to specify what kind of **values** it must encompass. Since PFL only supports integers and booleans, we defined the following data structure to represent the language's values:

```
data Value =
  I Integer | B Bool
```

Additionally, we defined several operations between values, so that we could manipulate them and filter out incorrect usage. All operations follow the same basic structure: if the two values are the correct type, perform it, else return nothing. As an example, consider the sum of two values:

```
-- operator overloading
(+) :: Value -> Value -> Maybe Value
(+) (I lhs) (I rhs) = Just (I (lhs Prelude.+ rhs) )
(+) _ _ = Nothing
```

Stack

PFL's virtual machine is stack-based, meaning that all primary interactions involve moving short-lived temporary values to and from a **stack**. For convenience, we defined this data structure as a list of values.

```
type Stack = [Value]
```

To abstract the stack manipulation details from the virtual machine, we created the functions below:

```
-- Creates a new empty stack.
createEmptyStack :: Stack
createEmptyStack = []
```

```haskell
-- Pushes a value onto the stack.
push :: Value -> Stack -> Stack
push el [] = el:[]
push el stack = el:stack
```

To test our virtual machine, we also implemented a function which converts the stack into a string.

```haskell
import Data.List (intercalate)

-- Prints the values on the stack.
stack2Str :: Stack -> String
stack2Str stack = intercalate "," (map show stack)
```

Note: As will become clear in due time, we did not require a function for popping a value off the stack.

State

The virtual machine's **state** is a binary search tree containing key-value pairs of variables and their current values. It is used to access and modify the program's variables.

For simplicity, we opted to model this data structure using Haskell's standard binary search tree implementation.

```haskell
import qualified Data.Map as Map (Map, empty, insert, lookup, mapAccumWithKey)

type State = Map.Map String Value
```

Thanks to this decision, creating the state manipulation functions was very simple, as we only had to call pre-established functions from the `Data.Map` library.

```haskell
-- Creates an empty machine state.
createEmptyState :: State
createEmptyState = Map.empty

-- Inserts a variable and its value in the machine's state.
-- If the variable is already present, its value is replaced with the new one.
push :: String -> Value -> State -> State
push key value state = Map.insert key value state

-- Looks up the value of a variable in the machine's state.
find :: String -> State -> Maybe Value
find key state = Map.lookup key state
```

Similarly to the stack, we defined a function which returns a string representation of the state. To iterate through the key-value pairs, we relied on the `Map.mapAccumwithKey` function.

```
-- Prints the values on the machine's state.
state2Str :: State -> String
state2Str state
  | acc == "" = ""
  | otherwise = init acc -- return everything except the last comma
  where
    (acc, _) = Map.mapAccumWithKey printVar "" state
    printVar acc key value = (acc ++ key ++ "=" ++ show value ++ ",", Nothing)
```

Executing the Bytecode

We created a single function for executing the bytecode instructions, fittingly named `run`. It is pretty straightforward, so, for brevity, we won't detail how each individual instruction is processed. However, they all fall into one of the following categories:

- **Push a literal to the stack**

For instructions like `Push Integer`, `Tru` and `Fals`, we simply push the adequate value to the top of the stack.

Ex:

```
-- Push an integer to the stack.
run ( (Push i):xs, stack, state) =
  run (xs, Stack.push (I i) stack, state)
```

- **Manipulate a variable**

PFL boasts two variable manipulation operations: `Fetch x`, which fetches the value of a variable and pushes it to the top of the stack, and `Store x`, which pushes the value on top of the stack and assigns it to a variable. Both functions appropriately verify if the variable exists.

```
-- Fetches the current value of a variable and pushes it to the top of the stack.
run ( (Fetch s):xs, stack, state ) =
  case State.find s state of
    Just value -> run ( xs, Stack.push value stack, state )
    Nothing -> error "Run-time error"

-- Fetches the current value of a variable and pushes it to the top of the stack.
run ( (Store s):xs, value:stack, state ) =
  run ( xs, stack, State.push s value state )
run ( (Store s):_, [], _ ) = error "Run-time error"
```

- **Binary operations**

Since there are a lot of operations that require popping two values from the top of the stack and performing a binary operation with them, we designed the following reusable function:

```haskell
-- Reusable function that pops two values from the stack and performs a binary operation with them.
-- The result of the operation is then pushed to the top of the stack.
applyBinaryOp :: (Value -> Value -> Maybe Value) -> Stack -> Stack
applyBinaryOp op ( lhs:rhs:stack ) =
  case op lhs rhs of
    Nothing -> error "Run-time error"
    Just value -> Stack.push value stack
applyBinaryOp _ _ = error "Run-time error"
```

Then, we simply call it, passing the corresponding operator as an argument.

Ex:

```haskell
-- Add two integers.
run ( Add:xs, stack, state ) =
  run ( xs, applyBinaryOp (Value.+) stack, state )
```

Note: To preserve the code readability, we opted to use Haskell's pattern mathing in `applyBinaryOp` to detect whether the stack had enough values for the operands. In our opinion, the alternative, which would be defining a function that popped the stack and returned nothing in case of failure, would clutter the code due to the amount of verifications it would require.

- **Unary operations**

Even though the 'not' operator is the only unary operator in PFL, we still created a reusable function for applying them. This would simplify extending the language with new unary operators in the future.

```haskell
-- Reusable function that pops a value from the stack and performs a unary operation with it.
-- The result of the operation is then pushed to the top of the stack.
applyUnaryOp :: (Value -> Maybe Value) -> Stack -> Stack
applyUnaryOp op ( x:stack ) =
  case op x of
    Nothing -> error "Run-time error"
    Just value -> Stack.push value stack
applyUnaryOp _ _ = error "Run-time error"
```

Once again, to use it, we must pass the operator as an argument.

Ex:

```haskell
-- Negates a bool.
run ( Neg:xs, stack, state ) =
  run (xs, applyUnaryOp (Value.not) stack, state)
```

- **Conditional branches**

PFL's conditional branch instruction, `Branch c1 c2`, verifies the top of the stack in search of a boolean value, issuing an error when it does not find one. Depending on whether the value is 'True' of 'False', the first (`c1`) or second (`c2`) list of instructions are executed, respectively.

```
-- A conditional branch.
run ( (Branch c1 c2):xs, (B True):stack, state ) =
  run ( c1 ++ xs, stack, state )
run ( (Branch c1 c2):xs, (B False):stack, state ) =
  run ( c2 ++ xs, stack, state )
run ( (Branch _ _):_, _, _ ) = error "Run-time error"
```

- **Loops**

The loop instruction, `Loop cond code`, functions like a standard programming loop: the condition is evaluated and, if it is 'True', the code inside the loop is executed, otherwise the virtual machine leaves the loop and executes the remaining instructions.

```
-- A loop.
run ( (Loop cond code):xs, stack, state ) =
  case run ( cond, stack, state) of
    ( [], (B True):stack', state' ) -> run ( (code ++ [Loop cond code] ++ xs), 
stack', state' )
    ( [], (B False):stack', state' ) -> run ( xs, stack', state' )
    (_, _, _) -> error "Run-time error"
```

Extra Features

In addition to the required language features that were specified in the project's description, we also developed the following new features to enrich PFL:

- Single-line and multiline comments.

```
// this is a comment
/* This is a multiline comment.
It can span multiple lines. */

x := /* Hello, World! */ 10;
```

- Variable names that contain keywords as substrings.

```
whileNot := 10; // this is a valid name
```

- *if* statements without an *else* clause.

```
x := 0;
if True then
    x := 5;
// x = 5
```

- Binary, octal and hexadecimal numbers.

```
x := 0b1111; // 15
y := 0o17; // 15
z := 0xF; // 15
```

- The remaining integer comparison operators - '!=', '<', '>' and '>='.

```
x := 21;
if x > 10 and x != 13 then
    x := 10;
// x = 10
```

- Logical disjunction i.e. the 'or' operator.

```
x := 0;
if False or True then
    x := 5;
// x = 5
```

- *until* loops, which are loops that are executed until the condition evaluates to 'True'.

```
x := 0; y := 0;
until x == 10
    (x := x + 1;
    y := y + 2;)
// x = 10, y = 20
```

Conclusions

Overall, creating a programming language from scratch, even if it was a tiny one, was a joy. In fact, watching our program unfold from a simple virtual machine to a fully functional compiler was undeniably a rewarding experience.

Moreover, Haskell proved to be an exceptional tool for this project. The functional programming paradigm let us solve some issues in the compilation pipeline, in particular within the parser, that would have been quite troublesome in imperative languages.

Bibliography

- "Crafting Interpreters", by Robert Nystrom: https://craftinginterpreters.com/
- Haskell documentation: https://www.haskell.org/documentation/