Grupo

Grupo: **T12_G03**

- Manuel Ramos Leite Carvalho Neto - up202108744
- Matilde Isabel da Silva Simões - up202108782
- Pedro Vidal Marcelino - up202108754

A contribuição de cada elemento do grupo para o trabalho prático é 33.3%.

Introdução

Este relatório apresenta em detalhe a implementação de uma máquina de baixo nível em Haskell, construída para executar instruções específicas. A tarefa proposta requereu a criação de um interpretador capaz de lidar com operações aritméticas, lógicas e controlo de fluxo, oferecendo uma solução robusta e flexível para a execução de programas de baixo nível. Ao longo deste relatório, exploraremos as estratégias adotadas, as decisões fundamentais na definição de tipos de dados e funções, além de discutir a implementação de aspetos cruciais para o funcionamento da máquina virtual.

Parte 1

Definição dos Tipos

Os tipos *Stack* e *State* foram definidos para representar a pilha e o estado da máquina virtual. A pilha é uma lista de valores que podem ser inteiros ou booleanos, enquanto o estado é um *Map* de pares chave-valor cujas chaves são *strings* e os valores podem ser inteiros ou booleanos.

A *Stack* e o *State* foram definidos usando uma declaração **type** porque são apenas novos nomes para tipos já existentes (sinónimos), isto é, a pilha não é mais do que uma lista de valores e o estado é apenas um *Map*. Assim, não se usou **data** porque não havia a necessidade de recorrer a construtores recursivos para estes tipos de dados. Deste modo, o uso de **type** neste contexto não só é mais correto, mas também simplifica a notação e facilita a leitura/compreensão do código.

```haskell
type Stack = [Either Integer Bool]
type State = Map String (Either Integer Bool)
```

A utilização da estrutura de dados *Map* para representar o estado da máquina virtual oferece vantagens significativas, destacando-se pela eficiência na procura de valores associados às variáveis. A capacidade de acesso eficiente, juntamente com a facilidade de inserção e remoção dinâmica de variáveis, torna o *Map* uma escolha apropriada para lidar com o estado durante a execução do programa. Além disso, a ordenação natural das chaves facilita a representação do estado como uma lista de pares chave-valor, ordenados alfabeticamente, como especificado no enunciado. Essa abordagem simplifica o código, contribuindo para uma implementação mais clara e eficaz da máquina de baixo nível.

Definição das Funções

As funções **createEmptyStack** e **createEmptyState** retornam versões vazias de pilha e do estado, respetivamente. Para a pilha, isso significa apenas uma lista vazia, enquanto para o estado usamos a função `empty` do módulo `Data.Map`.

```
createEmptyStack :: Stack
createEmptyStack = []

createEmptyState :: State
createEmptyState = empty
```

As funções **stack2Str** e **state2Str** convertem a pilha e o estado para *strings*.

A função **stack2Str** percorre a pilha e converte cada valor para uma *string*, sendo os valores são separados por vírgulas. Isso facilita a visualização da pilha durante a execução do programa.

```
stack2Str :: Stack -> String
stack2Str [] = ""
stack2Str (h:t)
  | null t = showValue h ++ stack2Str t
  | otherwise = showValue h ++ "," ++ stack2Str t
```

A função **state2Str** converte o estado para uma lista de pares chave-valor, listados por ordem alfabética e separados por vírgulas.

```
state2Str :: State -> String
state2Str state = case toList state of
  [] -> ""
  lst -> intercalate "," $ map (\(x, y) -> x ++ "=" ++ showValue y) lst
```

A função **run** desempenha o papel central na execução dos programas. Esta recebe como entrada a lista de instruções (**Code**), a pilha (**Stack**), e o estado (**State**). O padrão inicial verifica se a lista de instruções está vazia, indicando a conclusão bem-sucedida do programa. Neste caso, a função simplesmente retorna a pilha e o estado atuais.

```
run :: (Code, Stack, State) -> (Code, Stack, State)
run ([], stack, state) = ([], stack, state)
```

Caso haja instruções a serem processadas, o padrão subsequente utiliza a estrutura **case** para identificar o tipo de instrução na cabeça da lista. A partir daí, a lógica específica para cada instrução é executada, manipulando a pilha e o estado conforme necessário. Abaixo está exemplificada a lógica para a instrução **Add**.

```
run :: (Code, Stack, State) -> (Code, Stack, State)
run (h:t, stack, state) = case h of
  Add -> run (t, add stack, state)

add :: Stack -> Stack
add (x:y:t) = case (x, y) of
  (Left x, Left y) -> Left (x + y):t
  _ -> error "Run-time error"
add _ = error "Run-time error"
```

Em resumo, a função **run** coordena a interpretação de cada instrução, garantindo a integridade da pilha e do estado, bem como indicando a falha do programa caso ocorra um erro de execução.

Parte 2

Definição dos Tipos

O princípio exposto anteriormente para decidir entre declarar novos tipos com **type** ou **data** aplicou-se para os tipos abaixo definidos.

O tipo **Aexp** representa expressões aritméticas. Este tipo é definido por um conjunto de construtores de dados para: números inteiros (Num), variáveis (Var), adição (AddA), subtração (SubA) e multiplicação (MultA).

```
data Aexp = Num Integer | Var String | AddA Aexp Aexp | SubA Aexp Aexp | MultA
Aexp Aexp deriving Show
```

O tipo **Bexp** representa expressões booleanas. Este tipo é definido por um conjunto de construtores de dados para: valores booleanos (Bool), igualdade de expressões aritméticas (EqA), comparação "menor ou igual" entre expressões aritméticas (LeA), igualdade de expressões booleanas (EqB), conjunção de expressões booleanas (AndB) e negação de expressões booleanas (NegB).

```
data Bexp = Bool Bool | EqA Aexp Aexp | LeA Aexp Aexp | EqB Bexp Bexp | AndB Bexp
Bexp | NegB Bexp deriving Show
```

O tipo **Stm** representa instruções imperativas. Este tipo é definido por um conjunto de construtores de dados para: atribuição de variável (Assign), instruções condicionais (If) e ciclos (While).

```
data Stm = Assign String Aexp | If Bexp [Stm] [Stm] | While Bexp [Stm] deriving
Show
```

O tipo **Program** representa um programa. Este tipo é definido como uma lista de instruções **Stm**.

```
type Program = [Stm]
```

O tipo **Token** representa uma palavra (nome de variável ou palavra reservada) ou operador do código. Este tipo é definido por um conjunto de construtores de dados para: `:=` (AssignTok), `if` (IfTok), `then` (ThenTok), `else` (ElseTok), `while` (WhileTok), `do` (DoTok), `;` (SepTok), `(` (OpenTok), `)` (CloseTok), um número (NumTok), uma variável (VarTok), `+` (PlusTok), `-` (MinusTok), `*` (MultTok), `not` (NotTok), `and` (AndTok), `<=` (LessEqTok), `==` (EqTok), `=` (EqBoolTok), `True` (TrueTok) e `False` (FalseTok).

```
data Token = AssignTok | IfTok | ThenTok | ElseTok | WhileTok | DoTok | SepTok |
OpenTok | CloseTok
           | NumTok Integer | VarTok String | PlusTok | MinusTok | MultTok |
NotTok | AndTok
           | LessEqTok | EqTok | EqBoolTok | TrueTok | FalseTok deriving (Show,
Eq)
```

Definição das Funções

A função **compA** compila expressões aritméticas para código da máquina virtual. Utiliza recursão para percorrer a expressão, gerando o código correspondente para números, variáveis e operadores aritméticos.

```
compA :: Aexp -> Code
compA expA = case expA of
  Num num -> [Push num]
  Var var -> [Fetch var]
  AddA left right -> compA right ++ compA left ++ [Add]
  SubA left right -> compA right ++ compA left ++ [Sub]
  MultA left right -> compA right ++ compA left ++ [Mult]
```

A função **compB** compila expressões booleanas para código da máquina virtual. Similar a compA, utiliza recursão para percorrer a expressão e gera o código correspondente para valores booleanos, operadores de comparação e operadores lógicos.

```
compB :: Bexp -> Code
compB expB = case expB of
  Bool bool -> if bool then [Tru] else [Fals]
  EqA left right -> compA right ++ compA left ++ [Equ]
  LeA left right -> compA right ++ compA left ++ [Le]
  EqB left right -> compB right ++ compB left ++ [Equ]
  AndB left right -> compB right ++ compB left ++ [And]
  NegB exp -> compB exp ++ [Neg]
```

A função **compile** compila programas para código da máquina virtual. Utiliza recursão para percorrer a lista de instruções e gera o código correspondente para cada tipo de instrução. É, assim, considerada fundamental para a tradução efetiva de programas imperativos em código executável pela máquina virtual definida.

```
compile :: Program -> Code
compile [] = []
compile (h:t) = case h of
  Assign var expA -> compA expA ++ [Store var] ++ compile t
  If expB s1 s2 -> compB expB ++ [Branch (compile s1) (compile s2)] ++ compile t
  While expB s -> Loop (compB expB) (compile s) : compile t
```

Este compilador suporta atribuições, estruturas condicionais e ciclos, gerando código apropriado para cada caso.

Lexer

O **lexer** é responsável por transformar uma *string* numa lista de tokens, em que cada token representa um componente léxico da linguagem. Atua, assim, como a primeira fase do compilador, identificando e classificando elementos individuais, como operadores e números. A necessidade do *lexer* decorre da complexidade inerente à interpretação de código, que exigem uma representação estruturada e simplificada para um processamento eficiente.

No excerto de código seguinte, exemplifica-se a ação do *lexer* para o caso dos parênteses.

```
lexer :: String -> [Token]
lexer [] = []
lexer (c:t) = case c of
  '(' -> OpenTok : lexer t
  ')' -> CloseTok : lexer t
```

Ao dividir a *string* em unidades menores, os tokens, o *lexer* facilita o trabalho subsequente do *parser*. Cada token representa uma peça fundamental da linguagem de programação, permitindo que o compilador identifique e compreenda a estrutura e a semântica do código.

Parse

O **parse** constitui uma fase fundamental do processo de compilação, seguindo o *lexer*. Enquanto o *lexer* converte o código-fonte em tokens, o parser utiliza esses tokens para construir uma representação hierárquica conhecida como árvore sintática abstrata (AST). Essa árvore reflete a estrutura gramatical do programa, o que facilita a sua análise e construção.

Assim, o código do *parser* é composto por diversas funções auxiliares, cada uma delas dedicada a analisar e processar tipos específicos de instruções ou expressões da linguagem, retornando uma estrutura de dados que representa a instrução e contém a lista de tokens restantes.

Funções de *Statements* (`parseAssign`, `parseIfThenElse`, `parseWhile`): cada uma destas funções é especializada em analisar uma instrução específica da linguagem (atribuição, estrutura condicional `if-then-else` e ciclo `while`).

Funções de Expressões Aritméticas (`parseAexp`, `parseMultIntPar`, `parseIntPar`): responsáveis por analisar expressões aritméticas, lidando com adição, subtração, multiplicação e operações com números e

variáveis, respeitando a prioridade e a associatividade dos operadores.

Funções de Expressões Booleanas (`parseBexp`, `parseEqBNegEqALeBoolPar`, `parseNegEqALeBoolPar`, `parseEqALeBoolPar`, `parseLeBoolPar`, `parseBoolPar`): dedicadas à análise de expressões booleanas, incluindo operações de igualdade, negação e comparações.

A prioridade de operações é um princípio fundamental na análise sintática, garantindo que as expressões são avaliadas corretamente, respeitando a precedência e a associatividade dos operadores. Neste contexto, essa hierarquia é mantida cuidadosamente para as operações aritméticas e booleanas.

No caso das operações aritméticas, como adição, subtração, e multiplicação, a hierarquia é preservada na função **parseAexp**. Esta função chama **parseMultIntPar** que, por sua vez, chama **parseIntPar**. Dessa forma, a análise é realizada de acordo com a prioridade dessas operações, garantindo que a multiplicação é tratada antes da adição e da subtração.

No que diz respeito às operações booleanas, a hierarquia é respeitada na função **parseBexp**. Esta função chama **parseEqBNegEqALeBoolPar** (lida com igualdade entre expressões booleanas) que, por sua vez, chama **parseNegEqALeBoolPar** (lida com a negação) que, por sua vez, chama **parseEqALeBoolPar** (lida com igualdade entre expressões aritméticas) que, por sua vez, chama **parseLeBoolPar** (lida com a operação menor ou igual) e, por fim, chama **parseBoolPar** (lida com booleanos e parênteses). A hierarquia é mantida ao chamar as funções de forma sequencial, garantindo a avaliação correta das operações booleanas.

O código apresenta uma implementação cuidadosa e precisa no tratamento de parênteses, vital para assegurar a correta análise sintática das expressões. Nas funções **parseIntPar** e **parseBoolPar**, responsáveis por analisar expressões inteiras e booleanas, respetivamente, a presença de parênteses é estrategicamente considerada. Com a presença de um parêntese de abertura (**OpenTok**), as funções invocam **parseAexp** e **parseBexp**, respetivamente, para analisar a expressão contida dentro dos parênteses. De seguida, é verificada a presença do parêntese de fecho correspondente (**CloseTok**). Esta abordagem garante que as expressões contidas nos parênteses são avaliadas primeiro, seguindo a ordem estabelecida pela prioridade de operações aritméticas e booleanas.

Da mesma forma, na função **getInstructions**, que divide a lista de tokens em partes correspondentes a instruções específicas, o código lida adequadamente com parênteses. Ao encontrar um parêntese de abertura, a função chama-se recursivamente a si mesma, garantindo que as instruções dentro dos parênteses são processadas corretamente.

Funções Principais (`parse`, `parseAux`): utilizam as funções auxiliares mencionadas acima para realizarem a análise sintática geral da lista de tokens e iteram sobre as instruções, construindo a AST do programa à medida que ele avança.

Conclusão

A implementação de uma máquina de baixo nível em Haskell, apresentada neste relatório, representa um esforço abrangente para executar programas de baixo nível eficientemente. Ao abordar a definição de tipos, funções e estruturas, procuramos oferecer uma solução flexível e robusta, elaborando tipos de dados, estratégias de execução eficientes e a capacidade de compilar programas imperativos. A integração de conceitos fundamentais, como o *lexer* e o *parser*, permite a tradução da linguagem virtual para código capaz de ser executado pela máquina desenvolvida.