Segundo Trabalho Prático de PFL 2023

Âmbito

O presente documento diz respeito ao Segundo Trabalho Prático da Unidade Curricular de Programação Funcional e em Lógica, da Licenciatura em Engenharia Informática e Computação, no ano letivo 2023/24.

O trabalho representa a aplicação prática de conhecimentos adquiridos no âmbito do estudo do paradigma de programação funcional, através da linguagem Haskell.

Grupo e Distribuição de Trabalho

Grupo **T05_G01**:

- Pedro de Almeida Lima (up202108806@up.pt) - contributo de 50%
- Pedro Simão Januário Vieira (up202108768@up.pt) - contributo de 50%

Problema

O problema proposto para abordagem no descrito trabalho centra-se numa pequena linguagem de programação imperativa.

Num primeiro momento, foi implementado um *assembler* que, dadas instruções em código análogo ao Assembly para as operações oferecidas pela pequena linguagem, é capaz de simular a execução do código, recorrendo a estruturas de dados que fornecem abstrações para uma pilha (*stack*), um pequeno armazenamento (*state*) e a própria sequência de instruções.

Já numa segunda parte, desenvolveu-se um *parser* e um compilador responsáveis por converter código-fonte da pequena linguagem de programação para o referido código-máquina a ser interpretado pela solução desenvolvida para a primeira parte.

Detalhes da Implementação

Primeira parte — *Assembler*

Tendo em conta que o propósito do *assembler* é o de processar instruções em *assembly*, é fundamental uma estrutura de dados que represente as ditas instruções. Tal é assegurado por `data Inst`, como fornecido no modelo para o código do trabalho. Um conjunto de instruções é representado por `type Code = [Inst]`.

Dada a necessidade de suporte para variáveis com nome e conteúdo, recorreu-se a `type Var = (VarName, VarValue)`, que, para representar uma variável, consiste num par composto pelo nome (`type VarName = String`) e pelo conteúdo (`data VarValue`) dessa variável. Esta última estrutura distingue o conteúdo entre inteiros e booleanos.

Para responder à necessidade de ter uma pilha (*stack*) auxiliar à execução do código, bem como algo que cumpra o propósito de memória/armazenamento para os programas (*state*), foram criadas, respetivamente, `type Stack = [VarValue]` e `type State = [Var]`. Recordando que `Var` engloba o nome e o conteúdo de uma variável, o tipo apenas foi usado no âmbito do estado, dado que não é esperado nem desejável que a pilha contenha tal informação.

Foram, ainda, definidas funções que permitem visualizar os conteúdos da pilha e da memória de uma forma mais inteligível (`stack2Str :: Stack -> String` e `state2Str :: State -> String`, respetivamente), bem como funções que inicializam uma instância de cada uma das duas estruturas (`createEmptyStack :: Stack` e `createEmptyState :: State`, respetivamente).

Implementou-se funções de manipulação da pilha (`top` (obter o elemento no topo), `pop` (remover o elemento no topo) e `push` (colocar um elemento no topo da pilha)).

Concretizámos as funções que efetivamente cumprem o papel das instruções do código *assembly*, conforme definidas em `data Inst`, fazendo os cálculos necessários e manipulando a pilha e o estado. Para cada instrução, foi criada uma função com o mesmo nome, à exceção da instrução de `Loop`, uma vez que pode ser reescrita recursivamente como uma instrução `Branch` que encapsula o `Loop`.

Para o descrito efeito, algumas funções principais recorrem a funções auxiliares, de maneira a separar as preocupações e facilitar a leitura (e escrita) do código, permitindo que uma função que desempenhe o papel de uma instrução possa ser lida como uma abstração de mais alto nível tendo em conta a operação que se realiza, "escondendo" em funções auxiliares os meandros da operação.

Tomemos como exemplo as funções afetas à instrução de adição (`Add`):

```
addValues :: VarValue -> VarValue -> VarValue
addValues (IntegerValue a) (IntegerValue b) = IntegerValue (a + b)
addValues _ _ = error "Run-time error"

add :: Stack -> Stack
add stack = push result (pop (pop stack))
            where result = top stack `addValues` top (pop stack)
```

No caso, podemos constatar que o trabalho de verificação de verificação da viabilidade da operação (dado que, concretamente, apenas podemos adicionar dois inteiros e não um inteiro e um booleano, ou dois booleanos) e de efetuar a adição em si é deixado para `addValues`, ficando a função primária, `add`, com a responsabilidade de manipular a pilha recorrendo ao resultado da operação.

Por último, temos a função `run :: (Code, Stack, State) -> (Code, Stack, State)`, que garante o fluxo de processamento das instruções desejadas. É recursiva; para uma dada instrução, após invocar a função correspondente, chama-se a si própria com as instruções restantes, bem como a pilha e o estado resultantes da operação efetuada.

Segunda parte — Compilador e *Parser*

Compilador

Tal como pretendido no enunciado, definimos 3 estruturas fundamentais no âmbito da compilação dos programas: `data Aexp`, para representar expressões aritméticas, i.e. que produzem um inteiro como resultado, `data Bexp`, para representar expressões booleanas, i.e. que produzem um booleano como resultado, e `data Stm`, para representar aquilo que, numa linguagem de programação imperativa, não é expressão: *statements*, que podem conter expressões e outros *statements*. Definimos, ainda, `type Program = [Stm]`, que estabelece que um programa, no âmbito da segunda parte do trabalho, é uma sequência de

statements, dado que uma expressão, por si só, não tem efeito num programa escrito numa linguagem imperativa, i.e. apenas tem valor se enquadrada num *statement* que defina o que efetivamente se faz com o seu resultado.

As estruturas descritas servem como base comum entre o compilador e o *parser*, dado que o *parsing* da pequena linguagem produz instâncias delas.

Implementámos 3 funções basilares da compilação:

```
compA :: Aexp -> Code
compA (AddExp a1 a2) = compA a2 ++ compA a1 ++ [Add]
compA (MultExp a1 a2) = compA a2 ++ compA a1 ++ [Mult]
compA (SubExp a1 a2) = compA a2 ++ compA a1 ++ [Sub]
compA (Var x) = [Fetch x]
compA (Num n) = [Push n]

compB :: Bexp -> Code
compB (AndExp b1 b2) = compB b2 ++ compB b1 ++ [And]
compB (LeExp a1 a2) = compA a2 ++ compA a1 ++ [Le]
compB (AEquExp a1 a2) = compA a2 ++ compA a1 ++ [Equ]
compB (BEquExp b1 b2) = compB b2 ++ compB b1 ++ [Equ]
compB (NegExp b) = compB b ++ [Neg]
compB (Bool True) = [Tru]
compB (Bool False) = [Fals]

compStm :: Stm -> Code
compStm (Assign name a) = compA a ++ [Store name]
compStm (While b stmList) = [Loop (compB b) (compStm stmList)]
compStm (IfThenElse b thenStmList elseStmList) = compB b ++ [Branch (compStm
thenStmList) (compStm elseStmList)]
compStm (SequenceOfStatements stmList) = concatMap compStm stmList
compStm NoopStm = [Noop]
```

São responsáveis por transformar expressões aritméticas, booleanas e *statements*, respetivamente, em código *assembly* tal como definido na primeira parte do trabalho, assegurando a correta ordem deste código. Como pode ser observado, todas elas recorrem à recursividade, uma vez que a natureza tanto das expressões como dos *statements* pressupõe imbricação.

Por último, tal como especificado no enunciado, temos `compile :: [Stm] -> Code` que, através de uma chamada a `concatMap`, do prelúdio-padrão, obtém o código *assembly* correspondente a cada *statement* e concatena-o numa só lista de instruções *assembly* (recordando, da primeira parte, que `type Code = [Inst]`).

Parser

O componente de *parsing* da pequena linguagem de programação imperativa foi desenvolvido recorrendo à biblioteca Parsec.

No âmbito do Parsec, um *parser* para uma linguagem é, na verdade, um conjunto de *parsers* direcionados a subconjuntos da linguagem (no caso: expressões aritméticas e booleanas e seus termos, diferentes *statements*

e comentários). Assim, essencialmente, foram implementados:

```
-- Expressões aritméticas
aExpParser :: Parser Aexp

-- Termos das expressões aritméticas (inteiros, nomes de variáveis ou outras
expressões aritméticas)
aTerm :: Parser Aexp
intParser :: Parser Integer
varNameParser :: Parser String


-- Expressões booleanas
bExpParser :: Parser Bexp

-- Termos das expressões booleanas (comparações entre expressões aritméticas ou
constantes de valores de verdade)
arithmeticComparisonParser :: Text.Parsec.Prim.ParsecT   String ()
Data.Functor.Identity.Identity Bexp


-- Statements
statementParser :: Parser Stm

-- Diferentes tipos de statements
ifParser :: Parser Stm
whileParser :: Parser Stm
noStatementParser :: Parser Stm
sequenceOfStatementsParser :: Parser Stm
assignParser :: Parser Stm


-- Comentários
commentParser :: Parser String
```

É de notar que cada um dos sub-*parsers* pode, ainda, recorrer a (sub-)sub-*parsers*, o que explica, por exemplo, a existência de um `statementParser` distinto de todos os *parsers* para os diferentes tipos de *statements*.

Por último, há um aspeto a salientar, que se prende com o *parsing* de expressões:

Utilizando o Parsec, uma maneira genérica de definir um *parser* para expressões é fazer com que o mesmo invoque a definição dos operadores (aritméticos ou booleanos) e um sub-*parser* para os termos de uma expressão (por termo, entenda-se operando da expressão, que pode ser, também, uma nova expressão). Tal pode ser verificado, por exemplo, naquilo que diz respeito ao *parsing* de expressões aritméticas:

```
-- Parser para expressões aritméticas
aExpParser :: Parser Aexp
aExpParser = buildExpressionParser aOperators aTerm

-- Sub-parser para termos de uma expressão aritmética
aTerm :: Parser Aexp
aTerm = lexeme term
  where
    term = try (parens (lexeme aExpParser))
      <|> try (Num <$> lexeme intParser)
      <|> try (Var <$> lexeme varNameParser)
    parens = between (stringWithSpaces "(") (stringWithSpaces ")")

-- Operadores aritméticos
aOperators :: [[Operator String () Data.Functor.Identity.Identity Aexp]]
aOperators = [[Infix (MultExp <$ charWithSpaces '*') AssocLeft],
              [Infix (AddExp <$ charWithSpaces '+') AssocLeft, Infix (SubExp <$
charWithSpaces '-') AssocLeft]]
```

No sub-*parser* `aTerm`, a ordem dos "try" garante precendência de termos entre parênteses. Já em `aOperators`, o resultado retornado é uma lista de listas em que cada sublista agrupa operadores com prioridade igual, sendo que os operadores da primeira sublista têm maior prioridade do que os da segunda, e assim por diante. No caso, o operador de multiplicação ('*') tem prioridade máxima, ao passo que o de adição ('+') e de subtração têm menor (e igual entre um e outro).

Uma limitação deste método é que apenas permite que os termos de uma expressão aritmética sejam aritméticos, ou que termos de uma expressão booleana sejam booleanos.

Ora, para o segundo caso, sabemos que nem sempre tal se verifica na nossa pequena linguagem. Por exemplo, na expressão `2 <= 5`, `2` e `5` são termos aritméticos, mas o resultado da expressão é um booleano, pelo que se está perante uma expressão booleana.

Analogamente à situação anterior, temos:

```
-- Parser para expressões booleanas
bExpParser :: Parser Bexp
bExpParser = buildExpressionParser bOperators bTerm

-- Sub-parser para termos de uma expressão booleana
bTerm :: Parser Bexp
bTerm = lexeme term
  where
    term = try (parens (lexeme bExpParser))
      <|> try (lexeme arithmeticComparisonParser)
      <|> try (Bool False <$ lexeme (stringWithSpaces "False"))
      <|> try (Bool True <$ lexeme (stringWithSpaces "True"))
    parens = between (stringWithSpaces "(") (stringWithSpaces ")")

-- Operadores para booleanos
bOperators :: [[Operator String () Data.Functor.Identity.Identity Bexp]]
bOperators = [[Prefix (NegExp <$ stringWithSpaces "not")],
              [Infix (BEquExp <$ charWithSpaces '=') AssocNone],
              [Infix (AndExp <$ stringWithSpaces "and") AssocLeft]]
```

Como se pode verificar, falta a definição dos operadores de comparação entre termos aritméticos. Recordando a limitação exposta, é necessária uma outra maneira de definir um (sub-)*parser* para expressões em que o tipo dos termos não corresponde ao tipo do resultado:

```
arithmeticComparisonParser :: Text.Parsec.Prim.ParsecT   String ()
Data.Functor.Identity.Identity Bexp
arithmeticComparisonParser =
  do a1 <- aExpParser
     op <- arithmeticComparisonOperators
     a2 <- aExpParser
     return (op a1 a2)

arithmeticComparisonOperators :: Text.Parsec.Prim.ParsecT   String ()
Data.Functor.Identity.Identity (Aexp -> Aexp -> Bexp)
arithmeticComparisonOperators = (stringWithSpaces "<=" >> return LeExp)
                              <|> (stringWithSpaces "==" >> return AEquExp)
```

Surge, então, o `arithmeticComparisonParser`, que pode ser visto a ser invocado em `bTerm`. Recordando, precisamente, `bTerm`, a prioridade que os operadores de comparação entre termos aritméticos devem ter sobre os operadores entre termos booleanos é garantida pela ordem dos "try" nessa função.

Funcionalidades adicionais

Salientamos duas funcionalidades implementadas, adicionais àquilo que é exigido pelo enunciado:

- Capacidade para *parsing* de comentários (iniciados por `//`);
- Possibilidade de *parsing* (e, consequentemente, compilação e execução) de programas escritos em ficheiros de texto separados, i.e. sem ter de colar o código-fonte no GHCi. No diretório `src`, encontram-se dois ficheiros com código-fonte de exemplo (não necessariamente, naturalmente, com a extensão `.pfl` para a nossa pequena linguagem), que podem ser executados invocando `testParserFile`.

Exemplos de utilização

Seguem-se exemplos de utilização da solução implementada, dados pela chamada a fazer no GHCi e pelo respetivo retorno, um par com os conteúdos da pilha e da memória do programa executado.

Assembler

- `testAssembler [Fals,Push 3,Tru,Store "var",Store "a", Store "someVar"]` => `("","a=3,someVar=False,var=True")`
- `testAssembler [Push 10,Store "i",Push 1,Store "fact",Loop [Push 1,Fetch "i",Equ,Neg] [Fetch "i",Fetch "fact",Mult,Store "fact",Push 1,Fetch "i",Sub,Store "i"]]` => `("","fact=3628800,i=1")`
- `testAssembler [Push 1,Push 2,And]` => Obtém-se a mensagem de erro "Run-time error", dado que é impossível `1 and 2`.

Compilador e *Parser*

- `testParser "if (not True and 2 <= 5 = 3 == 4) then x :=1; else y := 2;"` => `("","y=2")`
- `testParser "i := 10; fact := 1; while (not(i == 1)) do (fact := fact * i; i := i - 1;);"` => `("","fact=3628800,i=1")`
- `testParserFile "codigo_exemplo2.pfl"` => `("","aux1=21,aux2=34,decimo_numero_fibonacci=34,i=9,temp=21")`
- `testParser "x:=5"` => Obtém-se uma mensagem de erro que contém:
  "unexpected end of input
  expecting digit, white space, operator or ";"".

Conclusões

Foi implementada com sucesso uma solução para o problema apresentado no âmbito do trabalho.

A sua realização permitiu aplicar, na prática, os conhecimentos adquiridos ao longo das aulas teórias e teorico-práticas da UC, além de ter constituído um primeiro projeto no paradigma da programação funcional. Foi uma oportunidade para aprimorar o domínio da linguagem Haskell.

Além disso, em particular, pudemos ficar a conhecer a biblioteca Parsec para Haskell.

Bibliografia

- *Parsing a simple imperative language — HaskellWiki*
    - https://wiki.haskell.org/Parsing_a_simple_imperative_language
    - Consultada pela última vez a 01/01/2024
- *Intro to Parsing with Parsec in Haskell*
    - https://jakewheat.github.io/intro_to_parsing/
    - Consultada pela última vez a 01/01/2024

Grupo T05_G01, 02/01/2024