Universidade do Minho

Mestrado em Engenharia Informática

Manutenção e Evolução de Software

Teste e Propriedades de Um Processador de Linguagem

Grupo 8:

Catarina Gonçalves (PG50180)

Francisco Toldy (PG50379)

Joana Alves (PG50457)

Junho 2023

Conteúdo

1 Introdução

O presente documento apresenta os resultados do projeto desenvolvido no âmbito da unidade curricular de Manutenção e Evolução de *Software* do perfil de Desenvolvimento, Validação e Manutenção de *Software* do Mestrado em Engenharia Informática.

Este projeto está fundamentalmente dividido em duas partes, tendo sido desenvolvidas de forma sequencial. Assim, apresentamos os objetivos de cada uma:

- **Parte I:** Desenvolvimento de um processador de uma linguagem à la BC do *Linux* e utilização de programação estratégica de forma a otimizar o código fonte e eliminar possíveis *smells*.
- **Parte II:** Desenvolvimento de geradores quer de mutantes quer de casos de teste e a definição de várias propriedades para as várias componentes do processador (aplicação de *Property Based Testing*).

Em suma, este relatório irá apresentar cada componente do projeto final, resultado do trabalho nas duas partes apresentadas.

2 Processador de Linguagem

Como referido na introdução, este trabalho iniciou-se pelo desenvolvimento do processador de linguagem. Desta forma, e para melhor se entender as secções seguintes, apresentamos uma descrição do *parser* desenvolvido, assim como os diversos ficheiros auxiliares e seu propósito.

2.1 Combinadores de Parsing

O primeiro passo desta primeira parte do trabalho prático baseou-se no desenvolvimento de combinadores de *parsing* na linguagem de programação escolhida: **Haskell**. Para tal, ao longo das aulas práticas da unidade curricular, em conjunto com os docentes, construímos todos os combinadores necessários (ficheiro *Parser.hs*) para desenvolver uma gramática de qualquer linguagem.

O primeiro passo foi definir o tipo de dados de um *Parser*, ou seja, que informação caracterizava o mesmo. Assim, construímos o seguinte tipo de dados, onde **r** representa o tipo de dados construído pelo *Parser*. Para além disto, no par resultado (*r*, *String*), **r** representa o *input* já reconhecido e transformado para o tipo de dados correspondente e **String** o resto do *input* que não foi consumido:

```
type Parser r = String -> [(r,String)]
```

De seguida, após a correta definição do tipo de dados de um *Parser*, passamos a construir os combinadores do mesmo. Assim, enumeramos os combinadores mais importantes do ficheiro:

- $<|>$ **:** Este combinador, a qualquer dois *Parsers*, unifica o resultado dos mesmos. É utilizado para numa mesma regra aplicarmos alternativas diferentes, por exemplo:

```
pStatements :: Parser Statement
pStatements =  pAtribuicao
           <|> pWhile
           <|> pFor
```

- $<*>$ **:** Este combinador, a qualquer dois *Parsers*, aplica o primeiro *Parser* ao resultado do segundo. É utilizado para numa regra construirmos a sequência de reconhecimento:

```
pAtribuicao :: Parser Statement
pAtribuicao = ident <*> symbol' '=' <*> pExp
```

- $<\$>$ **:** Este combinador aceita como argumentos uma função e um *Parser*, onde permite aplicar a função ao resultado do *Parser*. É utilizado para numa regra conseguirmos aplicar transformações ao resultado do *Parser*:

```
pReturn :: Parser Statement
pReturn = f <$> token' "return" <*> pMath
    where f a b = Return b
```

Estes combinadores, em conjunto com vários outros auxiliares, permitem o rápido desenvolvimento de uma gramática e manipulação da mesma.

2.2 Características da Linguagem

Tendo os combinadores corretamente construídos, passamos à fase de desenvolvimento e implementação da gramática da linguagem (ficheiro *Gramatica.hs*). Desta forma, o *Parser* desenvolvido apresenta suporte para uma linguagem à la BC do *Linux*, contando com declarações de variáveis e atribuição de valores à mesmas, expressões aritméticas e lógicas, ciclos *for* e *while*, entre outros. Para uma análise mais detalhada, passamos a enumerar em específico cada um destes:

- **Funções:** suporta tanto *function calls* e definição de funções

```
-- function calls          -- definição de função
func1(arg1, arg2);         fun func1(int a, string c){...};
```

- **Statements:**
  - **Ciclos:** *For* e *While*

```
-- ciclos for                    -- ciclos while
for(i=1;i<10;i=i+1){...};        while (a < 2) {...};
```

  - Estrutura **If Then Else**

```
if(i==1){...};
if(i==1){...} else {...};
```

  - **Atribuições:** 'a=1+2'; 'b=ola'
- **Declarações:** 'int a'; 'string b';
- **Expressões Lógicas e Aritméticas**

```
-- operadores: +, -, *, /, <, >, ==, >=, <=, ||, &&, true, false, !
    a + 2 < 1 && 20 / 1 >= 0;
```

No entanto, é de notar que o *Parser* **permite a leitura de várias frases**, ou seja, o resultado final é uma lista de frases lidas. Assim, as frases, mesmo que tenhamos apenas uma, têm de possuir o carácter de separação'**;**' no final. Exemplo:

```
parser "a+1;"
parser "a+1; a<4;"
```

2.2.1 Restrições da Gramática

Apesar do suporte a todas as *features* apresentadas acima, esta gramática apresenta algumas restrições de modo a apenas aceitar *inputs* o mais válidos possível. Assim, passamos a enumerar as restrições presentes na gramática desenvolvida:

- Não é permitido ter declarações de funções dentro de todas as outras estruturas de código, isto é, ciclos *for* e *while*, estruturas *if then else* e outras declarações de funções.
- Como não possui suporte para incrementação de valor de variáveis (p.ex.: a++), é necessário escrever a expressão completa, isto é: a = a+1.

2.3 Tipos de Dados Desenvolvidos

De modo a ser possível gerar uma árvore abstrata de reconhecimento do *input* do utilizador, foi necessário desenvolver, para além da respetiva gramática, os tipos de dados correspondentes para armazenar os valores reconhecidos (ficheiro *Ast.hs*). Assim, criamos dois tipos principais de dados, sendo estes: **Statement** e **Exp**.

O tipo de dados *Statement* engloba todas as operações de alto nível, enquanto que o tipo *Exp* está responsável pelas operações de baixo nível, principalmente expressões aritméticas e lógicas. É de notar que a gramática é uma lista de *Statements*. Para melhor compreensão, apresentamos, por extenso, os tipos de dados criados:

```
data Grammar = Grammar [Statement]

data Statement = Atrib String Exp
               | While Exp [Statement]
               | For [Statement] [Statement]
               | Decl Type String
               | If Exp [Statement] [Statement]
               | Function String [Statement] [Statement]
               | Exp Exp
               | Return Statement


data Exp = Add Exp Exp
         | Minus Exp Exp
         | Mul Exp Exp
         | Div Exp Exp
         | Const Int
         | True
         | False
         | Var String
         | Less Exp Exp
         | More Exp Exp
         | LEqual Exp Exp
         | MEqual Exp Exp
         | Equal Exp Exp
         | NotEqual Exp Exp
         | Or Exp Exp
         | And Exp Exp
         | FunCall String [Exp]
         | Str String
         | Not Exp


data Type = Int
          | Float
          | Double
          | String
          deriving (Show,Data,Eq)
```

2.4 Programação Estratégica

Como referido anteriormente, um dos objetivos desta primeira fase do trabalho foi precisamente a aplicação de programação estratégica para eliminação de *smells* e simplificação de expressões aritméticas (ficheiro *Opt.hs*). Desta forma, recorrendo aos *Zippers* do módulo **Ztrategic**, implementamos uma função **optGrammar** que, dada uma árvore abstrata, percorre a mesma utilizando uma travessia *innermost* do tipo *Type Preserving*, isto é, o tipo de dados de *input* mantém-se no *output*.

Para além disto, desenvolvemos duas funções a serem utilizadas no *Zipper*, tratando cada uma ou de eliminação de *smells* ou de simplificação de expressões aritméticas.

2.4.1 optExp

A função **optExp** trata todas as simplificações das expressões aritméticas e lógicas. Estas simplificações aritméticas lidam maioritariamente com a eliminação de valores redundantes nas adições e subtrações (tal como a sua simplificação), e com a propriedade do elemento neutro na multiplicação e divisão. Já as simplificações lógicas tratam otimizações para os operadores lógicos de conjunção e disjunção com valores booleanos e também as comparações de valores.

```
optExp :: Exp -> Maybe Exp

-- expressões aritméticas
optExp (Add (Const 0) e) = Just e
optExp (Add e (Const 0)) = Just e
optExp (Add (Const a) (Const b)) = Just (Const (a+b))

optExp (Minus (Const 0) (Const e)) = Just (Const (-e))
optExp (Minus (Const 0) e) = Just e
optExp (Minus e (Const 0)) = Just e
optExp (Minus (Const a) (Const b)) = Just (Const (a-b))

optExp (Mul (Const 0) e) = Just (Const 0)
optExp (Mul e (Const 0)) = Just (Const 0)
optExp (Mul (Const 1) e) = Just e
optExp (Mul e (Const 1)) = Just e

optExp (Div (Const 0) e) = Just (Const 0)
optExp (Div e (Const 1)) = Just e


-- comparação de valores
optExp (Less (Const c1) (Const c2)) = if (c1 < c2) then Just True else Just False
optExp (More (Const c1) (Const c2)) = if (c1 < c2) then Just True else Just False
optExp (LEqual (Const c1) (Const c2)) = if (c1 < c2) then Just True else Just False
optExp (MEqual (Const c1) (Const c2)) = if (c1 < c2) then Just True else Just False
optExp (Equal (Const c1) (Const c2)) = if (c1 == c2) then Just True else Just False
optExp (NotEqual (Const c1) (Const c2)) = if (c1 /= c2) then Just True else Just False


-- expressões lógicas
optExp (Or True exp) = Just True
optExp (Or exp True) = Just True
optExp (And False exp) = Just False
optExp (And exp False) = Just False
```

```haskell
-- operador de negação
optExp (Not (True)) = Just False
optExp (Not (False)) = Just True
optExp (Not (Less e1 e2)) = Just (MEqual e1 e2)
optExp (Not (More e1 e2)) = Just (LEqual e1 e2)
optExp (Not (LEqual e1 e2)) = Just (More e1 e2)
optExp (Not (MEqual e1 e2)) = Just (Less e1 e2)
optExp (Not (Equal e1 e2)) = Just (NotEqual e1 e2)
optExp (Not (NotEqual e1 e2)) = Just (Equal e1 e2)

optExp _ = Nothing
```

2.4.2 optSmellsStat

Por último, a função *optSmellsStat* trata da eliminação de potenciais *smells* no tipo de dados *Statement*, tendo, no momento, apenas um caso referente ao *statement* IF, em que o valor de retorno dos dois caminhos possíveis é equivalente ao resultado da expressão:

```haskell
optSmellsStat :: Statement -> Maybe Statement
optSmellsStat (If e [Return (Exp True)] [Return (Exp False)]) = Just (Return (Exp e))
optSmellsStat _ = Nothing
```

2.4.3 Travessia da Árvore

Explicitadas as funções para otimização do código, apresentamos agora um *snippet* de código com a função que aplica o *Zipper* na árvore abstrata:

```haskell
optGrammar :: Grammar -> Grammar
optGrammar l =
    let arvZipper    = toZipper l
        Just listaNova = applyTP (innermost step) arvZipper
            where step = failTP `adhocTP` optExp `adhocTP` optSmellsStat `adhocTP` optSmellsExp
    in fromZipper listaNova
```

Tal como referido anteriormente, esta função aplica uma travessia *innermost*, isto é, aplicação repetida de uma transformação até atingir um ponto fixo, e, para tal, foi necessário combinar a sua implementação com a função **failTP** que se trata de uma transformação pré-definida que falha em todos os casos. Desta forma, quer na função de otimização de expressões de aritméticas quer na de eliminação de *smells*, no último caso de aplicação da mesma, que se refere a todos os casos não interessantes e que portanto não fizeram *pattern matching* nas opções acima, é necessário devolver *Nothing*.

2.5 Unparser

Para além do *parser*, desenvolvemos também o *unparser*, isto é, recebe uma árvores abstrata resultante da aplicação do *parser* e transforma de volta no *input* inserido. Esta transformação foi implementada de forma recursiva, ou seja, percorre todos os nodos presentes na árvore e, para cada um, vai aplicando o *unparsing* respetivo, concatenando tudo numa *string* final.

Para uma melhor compreensão, apresentamos um *snippet* do código correspondente ao *unparsing* do tipo de dados *Statement*:

```
unpStatement :: Statement -> String
unpStatement (Atrib s exp) = s ++ " = " ++ unpExp exp
unpStatement (While exp stats) = "while (" ++ (unpExp exp) ++") {" ++ (unpStatements stats) ++"}"
unpStatement (For listArgs stats) = "for (" ++ (unpListaSemicolonExp listArgs) ++
                                    ") {" ++ (unpStatements stats) ++"}"
unpStatement (Decl t s) = unpType t ++ s
unpStatement (If exp stats []) = "if (" ++ (unpExp exp) ++ ") {" ++ (unpStatements stats) ++ "}"
unpStatement (If exp stat1 stat2) = "if (" ++ (unpExp exp) ++ ") {" ++ (unpStatements stat1) ++
                                    "} else {" ++ (unpStatements stat2) ++ "}"
unpStatement (Function s stat1 stat2) = "function " ++ s ++ " (" ++ (unpArgs stat1) ++
                                        ") {" ++ (unpStatements stat2) ++ "}"
unpStatement (Exp exp) = unpExp exp
unpStatement (Return s) = "return " ++ unpStatement s
```

3 Automated Test Case Generation

Para conseguir testar corretamente as funções de parser, unparser, otimizações e mutantes é necessário ter um exemplo de cada tipo de árvore que se poderia utilizar para testar, no entanto, com a infinidade de opções, seria melhor escolha a criação de um gerador para árvores do tipo desejado. Assim, foi criado um gerador que devolverá exemplos da gramática que é criada através do Parser.

Para conseguir criar este gerador, foi necessário criar para cada tipo de dados o seu correspondente gerador, isto é, existe um gerador para ciclos While, ciclos For, condições If then Else, e assim continuadamente por todos os tipos até conseguirmos criar uma lista de Statments, que consiste na Gramática desejada.

Exemplificando, para um gerador de declarações existe o seguinte código:

```
genDecl :: StateT Vars Gen Statement
genDecl = do t <- genType
             var <- genUniqueString
             return(Decl t var)
```

No entanto, visto que existe a restrição de que variáveis só podem ser utilizadas se foram declaradas e de declaração não repetidas ao mesmo nível, implicou a utilização do monad StateT do haskell que permitia guardar as variáveis utilizadas ao longo da geração de novos elementos da árvore.

Para tal, foi necessário alterar as assinaturas de todos os geradores entre outras funções que tiveram de ser alvo de pequenas alterações para acomodar esta alteração:

- **genUniqueString:w** Foi efetivamente criado para acomodar a segunda restrição, permitindo gerar variáveis que já não se encontrem declaradas, para o uso futuro no gerador de Declarações.

  ```
  genUniqueString :: StateT Vars Gen String
  genUniqueString = do varString <- genString
                       state <- get
                        if varString `elem` state
                       then genUniqueString
                       else do modify (varString :)
                               return varString
  ```

  Como podemos verificar, esta função é baseada na recursividade até encontrar uma variável que ainda não esteja declarada.

- **genExistingVar:** Este gerador também foi adicionado para permitir a utilização de variáveis já declaradas, obtendo apenas uma variável das que estão no estado.

  ```
  genExistingVar :: StateT Vars Gen String
  genExistingVar = do state <- get
                      lift $ elements state
  ```

- **genStatement:** Este gerador já existia, no entanto foi modificado para permitir que o gerador apenas avance para estritamente geradores de atribuições ou lógica com variáveis, se efetivamente existirem variáveis guardadas em estado (isto é, declaradas previamente).

```
genStatement :: StateT Vars Gen Statement
genStatement = do state <- get
                  if Prelude.null state
                    then genStatementNoAtrib
                    else genStatementFull
```

- **genFactor:** Semelhantemente ao gerador anterior, o *genFactor* terá de ter também em consideração se existe variáveis declaradas antes de poder gerar algum elemento que utilize alguma variável.

```
genFactor ::  StateT Vars Gen Exp
genFactor = do state <- get
               if Prelude.null state
                 then genFactorNoAtrib
                 else genFactorFull
```

Assim, perante este gerador, conseguimos obter exemplos da gramática extensos e com as restrições que foram requisitadas. No entanto, foi optado por deixar a geração de exemplos de tamanhos intermédios para permitir uma leitura mínimamente possível.

4 Mutation Testing

O segundo objetivo da última fase deste trabalho era a criação de um gerador de Mutantes. Este objetivo deveria ser cumprido recorrendo às noções de Geradores e programação estratégica previamente utilizadas. Para conseguir cumprir esta proposta, o grupo seguiu o seguinte raciocínio:

4.1 Catálogo de Mutações

O grupo começou por identificar uma série de possíveis conjuntos de mutações que pretendia que fossem aplicados às árvores criadas. Esses conjuntos de mutações são os seguintes:

- Troca de operadores aritméticos
- Troca de operadores relacionais
- Troca de operadores lógicos
- Mudança no valor de constantes
- Transformação de uma variável numa constante

Com base nestes conjuntos, foram desenvolvidos uma série de funções que recorrem a *Zippers* que serão detalhados na secção seguinte. Cada uma dessas funções terá associado um valor numérico num *switch* case na função gerador. Para uma dada execução do gerador, será executado o gerador aleatório auxiliar **escolherTipo** que irá gerar um número que irá indicar ao gerador principal **mutationGenerator** qual função aplicar. No total foram criadas **15** funções diferentes.

4.2 Funções com Zippers

Cada função criada tem como o propósito de percorrer a árvore e procurar um local apropriado para fazer a transformação pretendida, sendo que é feita apenas uma mutação por cada execução de uma dessas funções. O grupo considerou a aplicação de várias transformações numa mesma execução, no entanto, concluiu que seria mais útil e mais consistente com a noção de mutação executar apenas uma alteração. Assim, cada função implementada segue a seguinte estrutura:

```
mutationZipper :: Grammar -> Grammar
mutationZipper l =
    let arvZipper    = toZipper l
        Just listaNova = applyTP (once_tdTP step) arvZipper
            where step = failTP `adhocTP` mutationFunction
    in fromZipper listaNova
```

Como podemos observar, é utilizado o **once_tdTP** e o **failTP** para garantir o comportamento pretendido de travessia da árvore, uma vez que a travessia *once top down* (*once_tdTP*) apenas executa uma modificação na árvore e, de acordo com a documentação, necessita da função auxiliar *failTP* que, tal como referido anteriormente, consiste numa transformação pré-definida que falha em todos os casos. Assim, as funções definidas podem ser categorizadas de acordo com as secções seguintes.

4.2.1 Mutações de Operadores Aritméticos

Foram criadas 4 funções para permitir a criação de mutações de operações aritméticas. Cada uma dessas funções irá percorrer a árvore e executar uma auxiliar associada que irá transformar qualquer operador aritmético que encontre no operador aritmético "alocado"a essa função. Segue a função **mutateAddZipper** como exemplo:

```
mutateToAdd :: Exp -> Maybe Exp
mutateToAdd (Mul a b) = Just (Add a b)
mutateToAdd (Div a b) = Just (Add a b)
mutateToAdd (Minus a b) = Just (Add a b)
mutateToAdd e = Nothing

mutateAddZipper :: Grammar -> Grammar
mutateAddZipper l =
    let arvZipper    = toZipper l
        Just listaNova = applyTP (once_tdTP step) arvZipper
            where step = failTP `adhocTP` mutateToAdd
    in fromZipper listaNova
```

Esta função vai permitir a transformação de qualquer operador aritmético (multiplicação, divisão e subtração) numa adição. Além desta função, foram criadas funções específicas para modificar quaisquer operadores aritméticos para um dado operador:

- **mutateMinusZipper**
- **mutateDivZipper**
- **mutateMultZipper**

Com isto, conseguimos um total de **12** mutações individuais.

4.2.2 Mutações de Operadores Relacionais

Criadas as funções para as mutações relacionadas com os operadores aritméticos, o grupo mudou o seu foco para os operadores relacionais ($<$, $>$, $\leq$, $\geq$, $==$, $\neq$), mantendo o mesmo raciocínio que para as mutações de operadores aritméticos. Assim, segue o exemplo da função de mutação de qualquer operador relacional para um operador *less than* ($<$):

```
mutateToLess :: Exp -> Maybe Exp
mutateToLess (LEqual a b) = Just (Less a b)
mutateToLess (More a b) = Just (Less a b)
mutateToLess (MEqual a b) = Just (Less a b)
mutateToLess (Equal a b) = Just (Less a b)
mutateToLess (NotEqual a b) = Just (Less a b)
mutateToLess e = Nothing

mutateToLessZipper :: Grammar -> Grammar
mutateToLessZipper l =
    let arvZipper    = toZipper l
        Just listaNova = applyTP (once_tdTP step) arvZipper
            where step = failTP `adhocTP` mutateToLess
    in fromZipper listaNova
```

Como podemos observar, a construção da função mantém-se igual, com a função auxiliar que devolve a mutação a recorrer a *pattern matching* para transformar qualquer um dos operadores relacionais no operador relacional "less than".

O conjunto de funções relativas aos operadores relacionais é fechado com as seguintes funções criadas:

- **mutateToLEqualZipper**
- **mutateToMoreZipper**
- **mutateToMEqualZipper**
- **mutateToEqualZipper**

No total, conseguimos um total de **20** mutações individuais possíveis via este conjunto de funções.

4.2.3 Troca de Operadores Lógicos

Finalizados os operadores aritméticos e relacionais, o grupo procurou opções de mutações para operadores lógicos presentes na gramática criada, concluindo que as únicas mutações úteis a aplicar neste contexto seriam a troca entre operador AND e OR e a remoção do operador NOT. Segue o código dessas funções:

```
-- mutante para operadores AND e OR
mutateLogic :: Exp -> Maybe Exp
mutateLogic (And a b) = Just (Or a b)
mutateLogic (Or a b) = Just (And a b)
mutateLogic e = Nothing

mutateLogicZipper :: Grammar -> Grammar
mutateLogicZipper l =
    let arvZipper    = toZipper l
        Just listaNova = applyTP (once_tdTP step) arvZipper
            where step = failTP `adhocTP` mutateLogic
    in fromZipper listaNova


-- mutante para operador NOT
mutateNot :: Exp -> Maybe Exp
mutateNot (Not e) = Just e
mutateNot e = Nothing

mutateNotZipper :: Grammar -> Grammar
mutateNotZipper l =
    let arvZipper    = toZipper l
        Just listaNova = applyTP (once_tdTP step) arvZipper
            where step = failTP `adhocTP` mutateNot
    in fromZipper listaNova
```

4.2.4 Constantes e Variáveis

Terminadas as implementações das funções para mutações possíveis com os diferentes operadores da gramática definida pelo grupo, o foco mudou para as variáveis e constantes e que mutações seriam interessantes de implementar aí. Foram criadas 2 funções, uma cujo objetivo era alteração do valor de uma constante em qualquer operação aritmética:

```
mutateConst :: Exp -> Maybe Exp
mutateConst (Add (Const a) b) = Just (Add (Const (a+3)) b)
mutateConst (Add a (Const b)) = Just (Add a (Const (b+3)))
mutateConst (Minus (Const a) b) = Just (Minus (Const a+3) b)
mutateConst (Minus a (Const b)) = Just (Minus a (Const b+3))
mutateConst (Mul (Const a) b) = Just (Mul (Const (a+3)) b)
mutateConst (Mul a (Const b)) = Just (Mul a (Const (b+3)))
mutateConst (Div (Const a) b) = Just (Div (Const a+3) b)
mutateConst (Div a (Const b)) = Just (Div a (Const b+3))
mutateConst e = Nothing

mutateConstZipper :: Grammar -> Grammar
mutateConstZipper l =
    let arvZipper    = toZipper l
        Just listaNova = applyTP (once_tdTP step) arvZipper
            where step = failTP `adhocTP` mutateConst
    in fromZipper listaNova
```

E outra cujo objetivo era alteração de uma variável para um valor constante:

```
mutateVarToConst :: Exp -> Maybe Exp
mutateVarToConst (Add a (Var _)) = Just (Add a (Const 1))
mutateVarToConst (Add (Var _) b) = Just (Add (Const 1) b)
mutateVarToConst (Minus a (Var _)) = Just (Minus a (Const 1))
mutateVarToConst (Minus (Var _) b) = Just (Minus (Const 1) b)
mutateVarToConst (Mul (Var _) b) = Just (Mul (Const 1) b)
mutateVarToConst (Mul a (Var _)) = Just (Mul a (Const 1))
mutateVarToConst (Div (Var _) b) = Just (Div (Const 1) b)
mutateVarToConst (Div a (Var _)) = Just (Div a (Const 1))
mutateVarToConst e = Nothing

mutateVarToConstZipper :: Grammar -> Grammar
mutateVarToConstZipper l =
    let arvZipper    = toZipper l
        Just listaNova = applyTP (once_tdTP step) arvZipper
            where step = failTP `adhocTP` mutateVarToConst
    in fromZipper listaNova
```

4.3 Funcionamento do Gerador

Como já explicado, o gerador "final"de todas estas mutações começa por recorrer a um gerador auxiliar para obter um número aleatório de 1 a 15, sendo que depois esse número é utilizado num *switch* case, onde irá, consoante o valor, aplicar uma das funções anteriormente mencionadas à árvore que foi passada como argumento a este gerador:

```
escolherTipo :: Gen Int
escolherTipo = elements [1,2,3,4,5,6,7,8,9,10,11,12,13]


mutationGenerator :: Grammar -> Gen Grammar
mutationGenerator ast = do
  tipo <- escolherTipo
  case tipo of
    1 -> return (mutateAddZipper ast)
    2 -> return (mutateMinusZipper ast)
    3 -> return (mutateDivZipper ast)
    4 -> return (mutateMultZipper ast)
    5 -> return (mutateTrueFalseZipper ast)
    6 -> return (mutateToLessZipper ast)
    7 -> return (mutateToLEqualZipper ast)
    8 -> return (mutateToMoreZipper ast)
    9 -> return (mutateToMEqualZipper ast)
    10 -> return (mutateToEqualZipper ast)
    11 -> return (mutateToNEqualZipper ast)
    12 -> return (mutateLogicZipper ast)
    13 -> return (mutateNotZipper ast)
    14 -> return (mutateConstZipper ast)
    15 -> return (mutateVarToConstZipper ast)
    _ -> return ast
```

Este gerador é executado por uma função **mainGen**, que irá aplicar o gerador de mutantes a uma árvore obtida através da aplicação do gerador de árvores abstratas da liguagem.

```
mainGen :: IO ()i
mainGen = do
    original <- generate genGrammar
    mutated <- generate $ mutationGenerator original
    putStrLn $ "Initial AST: " ++ show original
    putStrLn $ "Mutated AST: " ++ show mutated
```

5 Property-based Testing

Nesta secção vamos apresentar as propriedades das várias componentes do processador, desenvolvidas de maneira a testar o *Parser*, referindo as adições ou modificações que precisamos de realizar para obter bons resultados. É de notar que estas propriedades foram todas referidas no enunciado pelos docentes.

5.1 Propriedade 1

A primeira propriedade desenvolvida tem como objetivo testar se fazer *parsing* após o *unparsing* de uma árvore abstrata, produz essa mesma árvore abstrata. Assim, o código desenvolvido foi o seguinte, notando que não foram necessárias alterações à estrutura de nenhum dos ficheiros:

```
prop_ParseAfterPrinting :: Grammar -> Bool
prop_ParseAfterPrinting ast = fst( head( parser(unparser ast) ) ) == ast
```

5.2 Propriedades 2 e 3

No enunciado, estas propriedades têm como objetivo testar se diferentes estratégias (*top down*, *bottom up*, *innermost*, etc) usadas na eliminação de *smells* e na otimização de expressões aritméticas são equivalentes. Desta forma, foi necessário desenvolver várias funções de travessia da árvore abstrata, modificando apenas o seu tipo. Assim, como algumas travessias necessitam da transformação **idTP** e outras a transformação **failTP**, e como cada uma destas necessita de um valor de retorno diferente para o caso em que não são aplicadas nenhumas modificações, decidimos separar estas travessias em ficheiros diferentes de acordo com a função auxiliar utilizada.

Assim, no ficheiro **Opt.hs** estão presentes as travessias que utilizam a função *failTP* e no ficheiro **OptID.hs** estão presentes as travessias que utilizam a função *idTP*. De seguida, foi apenas necessário aplicar as funções de eliminação de *smells* e de otimização de expressões aritméticas em cada uma delas.

```
prop_DifferentStrategies :: Grammar -> Bool
prop_DifferentStrategies ast =
    optGrammar ast == optGrammarOuter ast &&
    optGrammarOuter ast == optGrammarOnceTD ast &&
    optGrammarOnceTD ast == optGrammarOnceBUp ast &&
    optGrammarOnceBUp ast == optGrammarFullBUp ast &&
    optGrammarFullBUp ast == optGrammarFullTDown ast
```

Para uma melhor compreensão das várias travessias utilizadas, apresentamos a correspondência entre o nome da função e o nome da travessia:

- **optGrammar:** travessia *innermost*
- **optGrammarOuter:** travessia *outermost*
- **optGrammarOnceTD:** travessia *once top down*
- **optGrammarOnceBUp:** travessia *once bottom up*
- **optGrammarFullTD:** travessia *full top down*
- **optGrammarFullBUp:** travessia *full bottom up*

5.3 Propriedade 4

Esta última propriedade tem como objetivo testar se a eliminação de *smells* e a otimização de expressões aritméticas é comutativo. Desta forma, implementamos a seguinte função:

```
prop_SmellsExp :: Grammar -> Bool
prop_SmellsExp ast = optSmellsBefore ast == optGrammar ast
```

5.4 Resultados

Após o teste de cada propriedade através da utilização do **quickCheck**, denotamos alguns problemas nas suas execuções, passando a detalhar os mesmos nas secções seguintes.

5.4.1 Propriedade 1

Esta propriedade, como referido acima, testa a igualdade entre a árvore abstrata de *input* e o resultado de executar *parsing* após o *unparsing* da mesma. Assim, após analisarmos os casos em que esta propriedade **falhava**, chegamos à conclusão que se trata da ordem de reconhecimento das operações aritméticas, isto é, a multiplicação e divisão têm precedência na gramática desenvolvida, no entanto, as árvores de teste colocavam as restantes operações com precedência. Desta forma, ao fazer o *unparser* respetivo, obtínhamos uma expressão que, quando feito o *parsing*, alterava a ordem de precedência para a definida pela gramática.

5.4.2 Propriedades 2 e 3

Esta propriedade testa a igualdade entre resultados de aplicações de diferentes travessias de modo a executar quer otimizações de expressões aritméticas, quer de eliminação de *smells*. Desta forma, tal como era esperado, esta propriedade **falha**, uma vez que existem grandes diferenças na forma em como as travessias testadas atravessam a árvore abstrata.

Para além disto, notamos que nas travessias *once top down* e *once bottom up*, os testes resultavam em *Exceptions* de *non-exhaustive patterns* em alguns casos. Estas situações correspondiam aos casos onde a árvore abstrata não possuía nenhumas das otimizações possíveis.

5.4.3 Propriedade 4

Esta propriedade testava se a eliminação de *smells* e a otimização de expressões aritméticas são operações comutativas. Foi concluído que esta propriedade se **verifica**, uma vez que em várias execuções do *quickCheck*, em todas elas foram obtidos resultados positivos.

6 Conclusão

Terminado o período de desenvolvimento deste projeto, o grupo encontra-se satisfeito com o trabalho desenvolvido. Recorrendo ao conhecimento adquirido nas aulas desta unidade curricular, o grupo conseguiu apresentar uma versão do trabalho que vai de encontro aos objetivos estabelecidos em ambos os enunciados.

Apesar de algumas falhas identificadas pelo grupo em alguns pontos do projeto, foi feito um esforço para apresentar um trabalho o mais completo possível. Foi desenvolvido um *parser* e correspondente *unparser* totalmente funcionais para a gramática desenvolvida. Além disso, foi também desenvolvido um gerador de árvores para propósitos de teste, sendo essa funcionalidade aproveitada para a geração de mutantes nessas mesmas árvores, recorrendo a um amplo catálogo de mutações criado pelo grupo.

No entanto, pontos como a aplicação de programação estratégica encontraram alguns problemas, notáveis na sua utilização nas propriedades de teste criadas.

Em suma, aprofundamos o conhecimento lecionado nas aulas quer teóricas quer práticas relativamente a processadores de linguagem e combinadores de *parsing*, mas maioritariamente em relação a criação e implementação de geradores e testes baseados em propriedades.