Manutenção e Evolução de Software
2023

Grupo:
Catarina Gonçalves
Francisco Toldy
Joana Alves

TESTE E PROPRIEDADES DE UM PROCESSADOR DE LINGUAGEM

ÍNDICE

1 — Parser

2 — Gerador

3 — Mutantes

4 — Propriedades

PARSER

Características da Linguagem

Funções

Tanto function calls como definição de funções

```
func1(arg1, arg2);
fun func1(int a, string c){...};
```

Statements

- Ciclos For e While
- If Then Else
- Atribuições

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

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

a=1+2
b = ola
```

Declarações

Declaração de variáveis com o seu tipo de dados correspondente

```
int a          float i
string b       double j
```

Exp. Lógicas & Artiméticas

- +
- -
- *
- /
- <
- >
- <=
- =>
- ==
- !=
- !!
- &&
- true
- false
- !

```
a + 2 < 1 && 20 / 1 >= 0;
```

Statements

```
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
```

Exp

```
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
```

Grammar

```
data Grammar = Grammar [Statement]
```

Programação Estratégica

- optExp → otimizações expressões artiméticas
- optSmellsStat → eliminação de smells em Statement

Código Zipper:

```
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
```

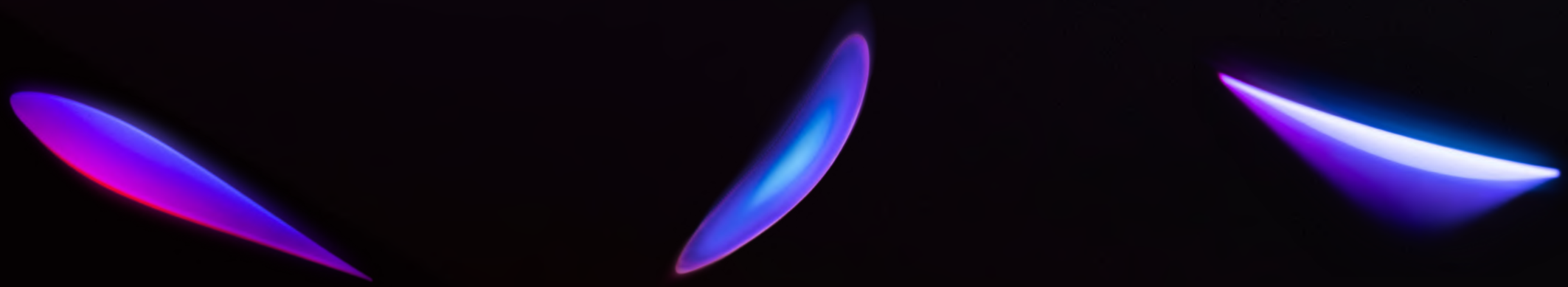
GERADOR

Geradores

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

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

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



MUTANTES

Catalogo de Mutações

Aritméticas

Mutações para diversos operadores aritméticos

Mul,Div,Minus → toAdd → Add

Mul,Div,Add → toMinus → Minus

Mul,Minus,Add → toDiv → Div

Div,Minus,Add → toMul → Mul

Relacionais

Mutações para diversos operadores relacionais

> , <= , >= , != , == →toLess → <

> , < , >= , != , == →toLEqual → <=

< , <= , >= , != , == →toMore → >

>, <= , > , != , == →toMEqual → >=

> , < , <= , >= , != →toEqual → ==

> , < , <= , >= , == →toNEqual → !=

Logicos

Mutações para operadores lógicos

OR → mutateLogic → AND

AND→ mutateLogic → OR

True→ trueFalse → False

False→ trueFalse → True

Var e Const

Mutações para Const e para Var

Const a →mutateConst → Const a+3

Var a →VarToConst → Const 1

Raciocínio

MainGen

Gera uma árvore recorrendo ao gerador genGrammar e executa o mutationGenerator no resultado

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

MutationGenerator

Executa o gerador de nº aleatorio escolherTipo, executa a função correspondente

```
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)
        ......
        13 -> return (mutateNotZipper ast)
        14 -> return (mutateConstZipper ast)
        15 -> return (mutateVarToConstZipper ast)
        _ -> return ast
```

Funcao zipper

Função que irá percorrer a árvore e aplicar a mutação no primeiro local adequado com a auxiliar

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

Função auxiliar

Função que irá aplicar a mutação

```
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
```

PROPRIEDADES

Propriedade 1

Objetivo: testar se fazer parsing após o unparsing de uma árvore abstrata, produz essa mesma árvore abstrata

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

Resultados: testes falham devido a precedência de leitura de operações aritméticas

Propriedade 2

Objetivo: testar se diferentes estratégias (top down, bottom up, innermost, etc) usadas na eliminação de smells e na otimização de expressões aritméticas são equivalentes

```
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
```

Resultados: testes falham devido à diferença acentuada no tipo de travessia das opções escolhidas

Propriedade 3

Objetivo: testar se a eliminação de smells e a otimização de expressões aritméticas é comutativo

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

Resultados: testes correram sem problemas, concluindo então que as duas operação são, de facto, comutativas