Universidade do Minho
Mestrado em Engenharia Informática

Manutenção e Evolução de Software

Processador de Linguagem

Grupo
Inês Ferreira - PG53879
Marta Sá - PG54084

25 de junho de 2024

Conteúdo

1 Introdução

Este trabalho prático foi redigido no âmbito da unidade curricular de Manutenção e Evolução de Software e tem como principal objetivo o desenvolvimento de um processador de linguagem, a aplicação de técnicas de programação estratégica, o desenvolvimento de um modulo de teste e a aplicação do algoritmo *Spectrum-Based Fault Localization*.

Assim, ao longo deste documento iremos apresentar e analisar os aspetos fundamentais do desenvolvimento e implementação do trabalho.

2 Gramática

No contexto de processamento de linguagens, é fundamental a definição de uma gramática capaz de representar a linguagem. Um gramática é, portanto, um conjunto de regras que define a estrutura e a formação de frases válidas numa linguagem formal. Estas regras especificam como os símbolos de uma linguagem podem ser combinados de forma a formar sentenças válidas.

Neste trabalho prático, a linguagem que definimos é semelhante à linguagem C, mas com menor complexidade. Desta forma, apresentamos de seguida a gramática para a linguagem *PicoC*:

⟨*pPicoC*⟩ -> pFuncs

⟨*pFuncs*⟩ -> zeroOrMore pFunc

⟨*pFunc*⟩ -> espacos pType pNomes '(' pArgs ')' '{' espacos pInsts '}'

⟨*pFuncCall*⟩ -> pNomes '(' pArgsCall ')'

⟨*pArgs*⟩ -> pArg
| pArg ',' pArgs
| empty

⟨*pArg*⟩ -> pType pNomes

⟨*pArgsCall*⟩ -> pArgCall
| pArgCall ',' pArgsCall
| empty

⟨*pArgCall*⟩ -> pNomes

⟨*pPrint*⟩ -> `"print("` pString ')'

⟨*pInsts*⟩ -> zeroOrMore pInst

⟨*pInst*⟩ -> pDeclAtrib ';' espacos
| pDecl ';' espacos
| pAtrib ';' espacos
| pWhile espacos
| pFor espacos
| pITE espacos
| pPrint ';' espacos
| pFuncCall ';' espacos
| pDeclAtribFuncCall ';' espacos

| pAtribFuncCall ';' espacos
| pReturn ';' espacos

⟨*pBlocoC*⟩ -> '{' pInsts '}'

⟨*pDeclAtribFuncCall*⟩ -> pType pNomes '=' pFuncCall

⟨*pAtribFuncCall*⟩ -> pNomes '=' pFuncCall

⟨*pAtribs*⟩ -> pAtrib
| pDeclAtrib

⟨*pDeclAtrib*⟩ -> pType pNomes '=' pExp1

⟨*pAtrib*⟩ -> pNomes '=' pExp1

⟨*pDecl*⟩ -> pType pNomes

⟨*pForInits*⟩ -> pAtribs
| pAtribs ',' pForInits
| empty

⟨*pForIncs*⟩ -> pAtrib
| pAtrib ',' pForIncs
| empty

⟨*pWhile*⟩ -> `while` '(' pExpLogicos ')' pBlocoC

⟨*pFor*⟩ -> `for` '(' pForInits ';' pExpLogicos ';' pForIncs ')' pBlocoC

⟨*pITE*⟩ -> `if` '(' pExpLogicos ')' `"then"` pBlocoC `"else"` pBlocoC
| `if` '(' pExpLogicos ')' pBlocoC

⟨*pReturn*⟩ -> `"return"` pExpLogicos

⟨*pType*⟩ -> `"int"`
| `"char"`
| `"string"`
| `"bool"`
| `"void"`

⟨*pExpLogicos*⟩ -> pExpEq `"&&"` pExpLogicos
| pExpEq `"||"` pExpLogicos
| `'!'` pExpLogicos
| pExpEq

⟨*pExpEq*⟩ -> pExp1 `"=="` pExpEq
  | pExp1 `'>'` pExpEq
  | pExp1 `'<'` pExpEq
  | pExp1 `">="` pExpEq
  | pExp1 `"<="` pExpEq
  | pExp1

⟨*pExp1*⟩ -> pExp0 '+' pExp1
  | pExp0 '-' pExp1
  | pExp0

⟨*pExp0*⟩ -> pFactor '*' pExp0
  | pFactor '/' pExp0
  | pFactor

⟨*pFactor*⟩ -> '-' pInt
  | pInt
  | pTrue
  | pFalse
  | pNomes
  | '(' pExpLogicos ')'

Através da observação da linguagem descrita acima podemos concluir que a mesma é constituída por um conjunto de funções, que são constituídas por instruções, que podem tratar-se de ciclos, condicionais, atribuições, declarações ou chamadas de outras funções. Para além disso, como era de esperar tanto as condições dos ciclos como as atribuições podem conter expressões, sendo que estas expressões apresentam uma ordem de prioridade definida.

3 *Parser*

Com base na gramática definida, criou-se uma biblioteca *parser* constituída por combinadores de *parsing* definidos, principalmente, pelas seguintes funções:

- *symbol'* - permite fazer o *parse* de um símbolo ignorando os espaços.
- *token'* - permite fazer o *parse* de *strings* ignorando os espaços.
- <|> - permite selecionar um *parse* entre várias alternativas.
- <*> - permite sequencializar múltiplos *parsers*.
- <$> - permite o processamento dos valores de retorno de um *parser*.

4 Árvore sintática abstrata

O objetivo do *parser* é transformar a sequência de *tokens* ou caracteres de entrada numa estrutura hierárquica que representa a sua sintaxe, geralmente, na forma de uma árvore sintática. A árvore resultante é constituída pelos tipos de dados abaixo apresentados:

```
data PicoC = PicoC [Func]
        deriving (Show, Data)

data Func = Func Type String [Arg] [Inst]
       deriving (Show,Data)

data Arg = Arg Type String
       deriving (Show,Data)

data ArgCall = ArgCall String
       deriving (Show,Data)

data Inst = Decl Type String
        | DeclAtrib Type String Exp
        | Atrib String Exp
        | Print String
        | DeclAtribFuncCall Type String Inst
        | AtribFuncCall String Inst
        | While Exp BlocoC
        | For [Inst] Exp [Inst] BlocoC
        | ITE Exp BlocoC BlocoC
        | CallFunc String [ArgCall]
        | Return Exp
       deriving (Show,Data)

type BlocoC = [Inst]

data Type = Int
        | Char
        | String
        | Bool
        | Void
       deriving (Show,Data)

data Exp = Exp '+' Exp
       | Exp '-' Exp
       | Mult Exp '*' Exp
       | Exp '/' Exp
       | Neg Exp
       | Const Int
       | Var String
       | Boolean Bool
       | Greater Exp Exp
       | Less Exp Exp
       | Equal Exp Exp
       | GreaterEqual Exp Exp
       | LessEqual Exp Exp
       | And Exp Exp
       | Or Exp Exp
       | Not Exp
       deriving (Show,Data)
```

Figura 1: Árvore sintática.

5 Programação Estratégica

A utilização de programação estratégica possibilita a travessia genérica de uma estrutura de dados, aplicando-lhe uma transformação ou redução e codificando apenas as interações que interessam. Para a implementação de mecanismos de programação estratégica utilizamos a biblioteca *Ztrategic* fornecida pelos docentes.

5.1 Otimizações

De forma a otimizar uma árvore sintática, utilizou-se a programação estratégica para desenvolver estratégias de travessia do tipo *Type Preserving*, nomeadamente, *innermost* e *full top-down*. A estratégia *innermost*, ao contrário da *full top-down*, percorre os ramos da árvore aplicando sempre que possível a otimização.

Desta forma, definimos as seguintes otimizações aplicadas a expressões:

Expression	Optimization
exp + (Const 0)	exp
(Const a) + (Const b)	Const (a+b)
exp * (Const 1)	exp
exp * (Const 0)	Const 0
exp / 1	exp
0 / exp	Const 0
Neg (Neg exp)	exp
Neg (Const a)	Const -a
Const 0 || exp	exp
exp || Const 0	exp
Const _|| exp	Const 1
exp || Const _	Const 1
Const 0 && exp	Const 0
Const 1 && exp	exp
!(!exp)	exp
!(Const 0)	Const 1
!(Const _)	Const 0
(Const a == Const b)	Boolean (a == b)
(Const a > Const b)	Boolean (a > b)
(Const a < Const b)	Boolean (a < b)
(Const a >= Const b)	Boolean (a >= b)
(Const a <= Const b)	Boolean (a <= b)
(Const a != Const b)	Boolean (a != b)

No que toca a otimizações feitas a nível de instruções, definimos as seguintes:

Instruction	Optimization
while (Const 0) b	while (False) b
while (Const 1) b	while (True) b
for (_; 0; _) b	for (_; False; _) b
for (_; 1; _) b	for (_; True; _) b
for (; r2;) b	while (r2) b
for (; r2; r3) b	while (r2) { b; r3; }

5.2 *Refactoring*

A estratégia que utilizamos para identificação e *refactoring* de *code smells* trata-se de uma estratégia TP(*Type Preserving*), nomeadamente, *innermost*. Desta forma, definimos os seguintes *smells*:

```
if (!cond) b1; else b2;  -> if (cond) b2; else b1;

if ( exp ) return true; else return false;  -> return exp;
```

6 Gerador Automático de Testes

No que toca à geração automática de casos de teste, utilizou-se as funções de geração da biblioteca *QuickCheck* e o tipo *Gen* para gerar árvores na linguagem especificada.

Desta forma, começamos por desenvolver geradores básicos, isto é, os geradores de tipos, de inteiros, de caracteres, de *strings* e de booleanos. De seguida, tratamos de criar um gerador de instruções que inclui um gerador para cada tipo de instrução existente no *PicoC* e, de maneira semelhante, implementamos um gerador de expressões. Por fim, desenvolvemos um gerador de funções que recebe como argumentos o número máximo de funções, de instruções e de expressões a serem geradas. Estes dois últimos argumentos são, por sua vez, passados aos geradores de instruções e de expressões, respetivamente.

De seguida, apresentamos alguns exemplos de geração de teste feitos a partir da função *testGen 1 1 1*:

```
PicoC [Func String "o6" [Arg Bool "pOXl",Arg Char "wvJ",Arg Char "rChgt"]
[ITE (Neg (Const 53)) [Print "7Y3S Ge tkR C5Qm"] [Return (Var "o7k")]]]
```

O seguinte resultado foi, por sua vez, obtido após executar o comando *testGen 2 2 3*:

```
PicoC [Func Int "jnp" [Arg Char "sG",Arg Char "jEryrl",Arg Bool "u7GR",
Arg Int "f6vmo",Arg String "tPufXF"] [While (And (Neg (Const 66)) (Var "eUvJ"))
[Print "IxtP L0gKb"]],Func Void "zN" [Arg Int "eI",Arg Bool "clwKQ",
Arg Int "adCw",Arg String "ph0"] [Print "vPSx M R4",
DeclAtrib Int "rNF" (Neg (Const 65))]]
```

7 Gerador de Mutações

No contexto do módulo *MutationGenerator.hs*, foram implementadas diversas funcionalidades para gerar mutações em expressões de programas com o objetivo de aplicar, posteriormente, algoritmos de deteção de falhas. Desta forma, o gerador de mutações necessita de uma expressão original de forma a gerar mutações muito idênticas e difíceis de detetar, tal como, a troca de uma soma por uma subtração.

Para aplicar as mutações utilizamos a função *chooseExp* para, primeiramente, agregar todas as expressões de um programa numa lista, selecionando, de seguida, uma delas. Posteriormente, utilizamos a função *chooseMutation* para selecionar a mutação a ser feita. Por fim, aplicamos a mutação à expressão selecionada a partir de um estratégia *full_tdTP* de forma a obter um programa mutado.

8 Testes de Propriedades

Esta abordagem de teste consiste na definição de propriedades que um programa deve satisfazer. No desenvolvimento deste módulo, foram definidas as seguintes quatro propriedades:

- *prop_PicoC* - testa se o *unparse* é o inverso do *parse*.
- *prop_Innermost* - testa se o *innermost* é idempotente.
- *prop_Strategies* - testa se diferentes estratégias são equivalentes, nomeadamente, *innermost* e *top-down*.
- *prop_OptCommutativeRef* - testa a comutatividade de otimizações com o *refactoring*.

9 Fault Location

No contexto de engenharia de software, a localização de falhas mostra ser uma área fundamental, dado que permite uma identificação rápida e eficiente de *bugs* no programa. Desta forma, começamos por definir 3 programas que consideramos representativos da linguagem *PicoC*.

Posteriormente, criamos a função *runTestSuite* que, para cada programa definido, irá correr testes unitários disponíveis e validar se todos passaram, comparando o resultado esperado de cada um com o resultado obtido. Com o auxílio do módulo *MutationGenerator.hs*, tratamos de inserir uma mutação em cada um dos três programas. De seguida, corremos os mesmos com vários inputs diferentes e verificamos, novamente, se os resultados obtidos iam de encontro aos resultados esperados. Por fim, criamos a função *instrumentation* responsável por instrumentar um programa para auxiliar na localização de falhas, inserindo *prints* em cada instrução com a respetiva linha. Desta forma, a partir da função *instrumentedTestSuite*, cada programa é instrumentado antes de ser executado, permitindo visualizar quais as instruções executadas pelo mesmo.

Programa 1

```
1  int main(int a){
2      int margem = a;
3      if (margem > 30)
4      then { margem = 4 * 23 + 3 ; }
5      else { margem = 0; }
6      return margem;
7  }
```

Figura 2: Programa 1 sem mutação.

Programa 1 Mutado

```
1  int main(int a){
2      int margem = a;
3      if (margem > 30)
4      then { margem = (4 * 23) / 3 ; }
5      else { margem = 0; }
6      return margem;
7  }
```

Figura 3: Programa 1 com mutação.

Programa 2

```
1  int main(int y, int z){
2      bool teste = False;
3      for(int i=0; i<5 && !teste; i=i+1){
4          y = y + i * z;}
5      if(y>60){
6          y = y - 25;}
7      return y;
8  }
```

Figura 4: Programa 2 sem mutação.

Programa 2 Mutado

```
1  int main(int y, int z){
2      bool teste = False;
3      for(int i=0; i<=5 && !teste; i=i+1){
4          y = y + i * z;}
5      if(y>60){
6          y = y - 25;}
7      return y;
8  }
```

Figura 5: Programa 2 com mutação.

Programa 3

```
1  int main(int a){
2      int c;
3      c=2+1;
4      if(a < 3) then{
5          int i=0;
6          int j=5;
7          while(i<7){
8              j=j-1;
9              i=i+1;}
10         c = c * j;
11     } else {c = 0;}
12     print(" CHEGUEI AQUI! ");
13     return c;
14 }
```

Figura 6: Programa 3 sem mutação.

Programa 3 Mutado

```
1  int main(int a){
2      int c;
3      c=2+1;
4      if(a<3) then{
5          int i=0;
6          int j=83;
7          while(i<7) {
8              j=j-1;
9              i=i+1;}
10         c=c*j; }
11     else{c=0;}
12     print(" CHEGUEI AQUI! ");
13     return c;
14 }
```

Figura 7: Programa 3 com mutação.

Com a informação das instruções de cada programa executadas em cada teste unitário, incluímos numa folha de cálculo o código de cada programa juntamente com as tabelas que são usadas no algoritmo de *Spectrum-Based Fault Localization* para localizar as instruções com mais probabilidade de erro em cada um dos 3 programas. Neste algoritmo é utilizado o coeficiente de Jaccard que é definido da seguinte forma:

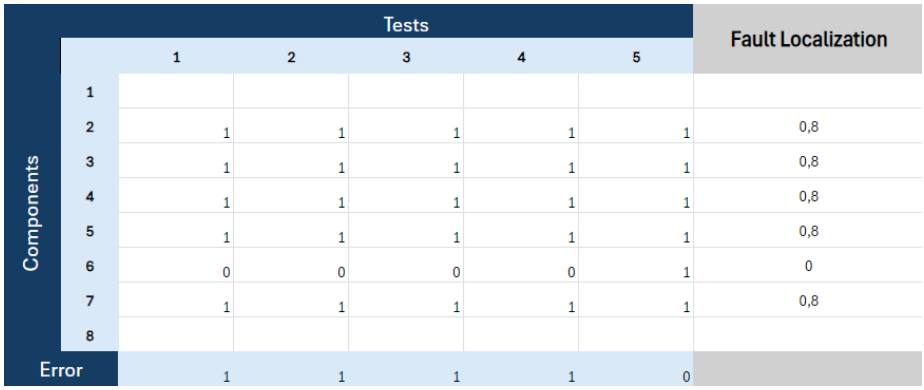
$$\text{Coeficiente de Jaccard} = \frac{n11}{n11 + n10 + n01} \tag{1}$$

Onde, *n11* representa o número total de atributos onde a componente e o erro apresentam ambos o valor 1, *n10* representa o número total de atributos onde a componente tem o valor 1 e o erro o valor 0 e onde *n01* representa o número total de atributos onde a componente tem o valor 0 e o erro o valor 1.

Desta forma, as tabelas abaixo apresentam os resultados do algoritmo aplicado a cada programa mutado:

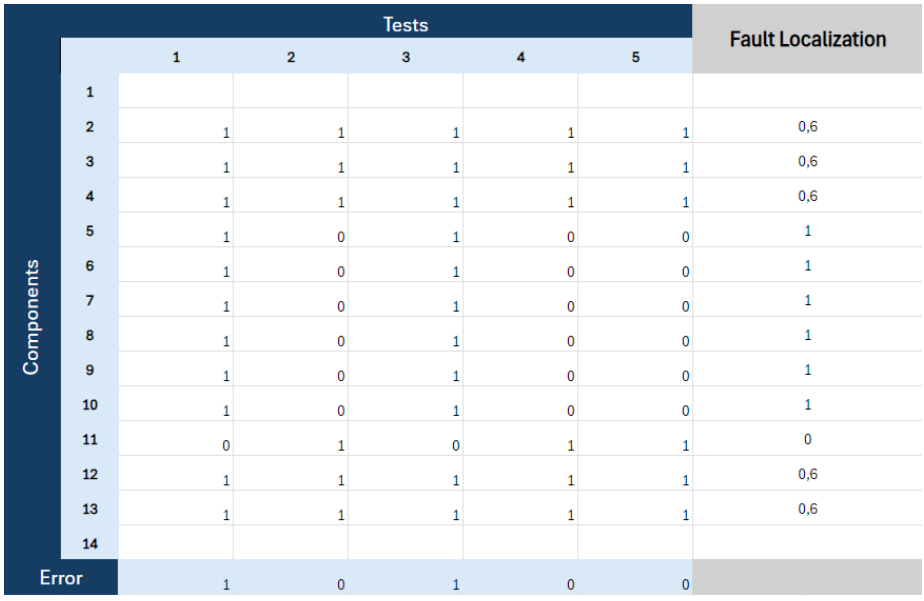
		Tests					Fault Localization
		1	2	3	4	5	
Components	1						
	2	1	1	1	1	1	0,4
	3	1	1	1	1	1	0,4
	4	1	0	0	1	0	1
	5	0	1	1	0	1	0
	6	1	1	1	1	1	0,4
	7						
Error		1	0	0	1	0	

Figura 8: Fault localization programa 1.



Components	Tests 1	2	3	4	5	Fault Localization
1						
2	1	1	1	1	1	0,8
3	1	1	1	1	1	0,8
4	1	1	1	1	1	0,8
5	1	1	1	1	1	0,8
6	0	0	0	0	1	0
7	1	1	1	1	1	0,8
8						
Error	1	1	1	1	0	

Figura 9: Fault localization programa 2.



Components	Tests 1	2	3	4	5	Fault Localization
1						
2	1	1	1	1	1	0,6
3	1	1	1	1	1	0,6
4	1	1	1	1	1	0,6
5	1	0	1	0	0	1
6	1	0	1	0	0	1
7	1	0	1	0	0	1
8	1	0	1	0	0	1
9	1	0	1	0	0	1
10	1	0	1	0	0	1
11	0	1	0	1	1	0
12	1	1	1	1	1	0,6
13	1	1	1	1	1	0,6
14						
Error	1	0	1	0	0	

Figura 10: Fault localization programa 3.

10 Conclusão

Este trabalho prático proporcionou uma experiência abrangente na aplicação de técnicas essenciais para a manutenção e evolução de software, desde a definição de uma gramática até à criação de testes e localização de falhas.

A construção da gramática *PicoC* e o desenvolvimento do *parser* asseguraram a correta interpretação das instruções da linguagem. A árvore sintática abstrata facilitou a visualização hierárquica da sintaxe, essencial para otimizações e *refactoring*.

Por sua vez, a programação estratégica permitiu a implementação eficiente de otimizações e *refactorings*, melhorando a eficiência e legibilidade do código. A geração automática de testes com *QuickCheck* validou extensivamente as funcionalidades do processador de linguagem.

O gerador de mutações e a aplicação do algoritmo *Spectrum-Based Fault Localization* foram eficazes na identificação e correção de falhas, demonstrando a importância dessas técnicas na manutenção de software.

No geral, o projeto consolidou conhecimentos teóricos e práticos, desenvolvendo competências essenciais para a carreira de um engenheiro informático, promovendo a criação de software de alta qualidade e fácil manutenção.