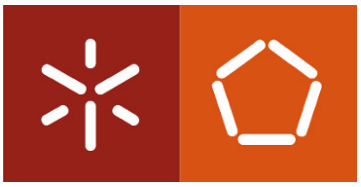

UMinho

Mestrado em Engenharia Informática

Manutenção e Evolução de Software

Processador de Linguagem

Simão Cunha(a93262)
Gonçalo Pereira (a93168)
Luís Silva (pg50564)

Braga, 12 de junho de 2023

Conteúdo

1 Introdução

Ferramentas capazes de processar linguagem natural são indispensáveis para a comunicação entre humanos e computadores. O presente relatório surge no âmbito do trabalho prático da UC de Manutenção e Evolução de Software. Neste trabalho foi-nos proposto o desenvolvimento de um processador de linguagem, aplicação de técnicas de programação estratégica e desenvolvimento de um modulo de testagem. No decorrer deste documento iremos apresentar os aspetos fundamentais do desenvolvimento e implementação do trabalho.

2 Gramática

Uma linguagem tem sempre como base um conjunto de regras que estabelecem o que são ou não frases válidas nessa mesma linguagem. Esse conjunto de regras é a gramática da linguagem que determina quais as "palavras"e a sua possível organização para que formem frases válidas. No âmbito deste trabalho prático, a linguagem que definimos é idêntica à linguagem C, apenas menos complexa. Abaixo apresentamos a gramática para esta linguagem:

```
Program     -> zeroOrMore Func
Func        -> spaces Type '(' Parameters ')' '{' Statements '}'
Parameters  -> Parameter ',' Parameters
            | empty
Parameter   -> Type
Type        -> int
            | char
            | string
            | void
Statements  -> Decl ';' Statements
            | Assign ';' Statements
            | If Statements
            | FuncCall ';' Statements
            | While Statements
            | For Statements
            | empty
Decl        -> Declare | DeclareAssign
Declare     -> Type
DeclareAssign -> Type '=' Exp
Assign      -> '=' Exp
If          -> "if" '(' Cond ')' Bloco
            | "if" '(' Cond ')' Bloco "else" Bloco
FuncCall    -> '(' Args ')'
While       -> "while" '(' Cond ')' Bloco
Cond        -> NestedCond "==" Cond
            | NestedCond "||" Cond
            | NestedCond "&&" Cond
            | NestedCond "<" Cond
            | NestedCond ">" Cond
            | NestedCond "<=" Cond
```

```
            | NestedCond ">=" Cond
            | "!" Cond
            | NestedCond
NestedCond  -> Exp
            | '(' Cond ')'
For         -> "for" '(' (zeroOrMore Decl) ';' Cond ';' (zeroOrMore Assign) ')' Bloco
Bloco       -> '{' Statements '}'
Exp         -> spaces Exp1
Exp1        -> Exp2 "||" Exp1 | Exp2
Exp2        -> Exp3 "&&" Exp2 | Exp3
Exp3        -> Exp4 "==" Exp3 | Exp4
Exp4        -> Exp5 '<' Exp4
            | Exp5 '>' Exp4
            | Exp5 "<=" Exp4
            | Exp5 ">=" Exp4
            | Exp5
Exp5        -> Exp6 '+' Exp5
            | Exp6 '-' Exp5
            | Exp6
Exp5        -> Exp6 '*' Exp5
            | Exp6 '/' Exp5
            | Exp6
Exp6        -> int
            | true
            | false
            | var
            | '(' Exp ')'
```

Como é possível observar nesta gramática, determinamos a constituição de um programa como um conjunto de funções, definimos o que são *statements* e expressões válidas, entre outros aspetos importantes da linguagem. Por fim, é possível notar que definimos uma expressão, onde mostramos os diferentes níveis de prioridade a esta inerentes.

3 Parser

A construção do módulo de *parsing* é baseada na gramática previamente definida, na medida em que o método de interpretar *inputs* aplica as regras da gramática. O *parser* é definido à custa de uma biblioteca de combinadores de *parsing* definida que expõem funções como:

- $< | >$: permite aplicar diferentes alternativas de *parsing*;
- $< * >$: permite *parsing* sequencial;
- $< \$ >$: permite descrever como os valores de retorno são processados;

Além disso, faz-se uso de funções como *symbol* e *token* para fazer o *parsing* de símbolos e de *tokens*, respetivamente. O *parser* tem como objetivo a geração da árvore sintática abstrata que reflete e sintetiza o *input* do qual foi feito *parse*. Após o processamento de cada parte

do *input*, é retornado o tipo de dados correspondente. No fim do processamento, o *input* é reduzido a uma árvore.

4 Árvore sintática abstrata

Como mencionado em 3 após o processamento de um *input* obtém-se a árvore sintática abstrata. A árvore resultante é constituída pelos tipos de dados abaixo apresentados.

```
-- Data type of program
data Program = Prog [Func]
             deriving (Show, Data)


-- Data type of function
data Func = FunctionDeclaration Type String [Par] [Stat]
            deriving (Show, Data)

-- Data type of parameter (function args in a function declaration)
data Par = Parameter Type String
                deriving (Show, Data)


-- Data type of statements
data Stat = Assign String Exp
          | Declare Type String
          | DeclAssign Type String Exp
          | ITE Exp [Stat] [Stat]
          | While Exp [Stat]
          | For [Stat] Exp [Stat] [Stat]
          | FunctionCall String [Exp]
          | Sequence [Stat]
          | Return Exp
          deriving (Show, Data)


-- Data type of type
data Type = Int
          | Char
          | String
          | Void
          deriving (Show, Data)

-- Data type of expressions
data Exp = Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp
```

```
    | Not Exp
    | Const Int
    | Var String
    | Boolean Bool
    | EqualsTo Exp Exp
    | Or Exp Exp
    | And Exp Exp
    | LessThen Exp Exp
    | MoreThen Exp Exp
    | LessEqualThen Exp Exp
    | MoreEqualThen Exp Exp
    | ExpFunctionCall String [Exp]
    deriving (Show, Data)
```

Estes tipos de dados definem o conceito de programa, função, parâmetros, declarações, tipos e expressões. Os tipos são auto explicativos à exceção do `Sequence [Stat]`. Este tipo de `Stat` foi definido com o propósito de facilitar a separação de *statements* durante a fase de otimizações da programação estratégica.

5 Programação estratégica

O uso de programação estratégica permite a navegação de árvores heterogéneas (diferentes tipo de dados) de uma forma simples e eficiente. Para a implementação de mecanismos de programação estratégica utilizamos a biblioteca *Ztrategic* fornecida pelos docentes.

5.1 Otimizações

De forma a aplicar algumas otimizações às àrvores obtidas, desenvolvemos uma estratégica TP (*type preserving*) em *innermost* com o objetivo de atravessar todos os caminhos da árvore e aplicar otimizações sempre que possível. As otimizações que implementamos foram:

- Aplicação do elemento neutro das operações;
- Tradução de valores numérico em ciclos para booleanos, e transformações de ciclos 'for' em ciclos 'while' (caso em que foi usado o **Sequence** referido na secção anterior).

5.2 Smell refactoring

De maneira semelhante implementamos uma estratégica TP (*type preserving*) em *innermost* para identificação e *refactoring* de *code smells*. Os *smells* definidos foram:

- ```
  if (not a) { b1 } else { b2 }
  ```

  ↓ refactoring

  ```
  if(a){b2}else{b1}
  ```
  ```

- ```
  if( f2(); ){
      return True
  }else{
      return False
  }
  ```

  ↓ *refactoring*

  ```
  return f2();
  ```

5.3 Bug counter

Implementamos uma estratégia TU (*type unifying*) em full_tdTU com o propósito de contar o número de bugs num dado programa. Até ao momento apenas estamos a considerar o bug da divisão por zero.

Executando a função bugCount do módulo Runner.hs para o input: `"void f(int x){\nint x = 2 / 0;\n\n\nint x = 2 / 0;}"`

obtém-se 2 como resultado, já que foram identificadas duas situações em que se divide por zero. Expandindo o catálogo de bugs é possível detetar e contar diferentes bugs num programa.

5.4 Gather Data

Implementamos uma estratégia TU (*type unifying*) em full_tdTU com o propósito de recolher alguns detalhes importantes sobre um dado programa. A estratégia definida permite mapear para cada programa as funções declaradas e em cada função as variáveis que foram declaradas e usadas. Além disso, esta estratégia permite que sejam facilmente adicionadas outras funcionalidades para recolha de outras métricas. Após a execução de `gatherProgramData "void main(string y)int x = 2*1;string x; x = 3*1; length(x,2+2); void f(int x,string y)int x = 2 + 0 + 0;"` , obteve-se o seguinte output:

```
[("main",["x"],["x"]),("f",["x"],[])]
```

O resultado obtido indica que o programa é constituído pelas funções *main* e *f*. Na primeira foi declarada e usada a variável x e na segunda foi declarada mas não foi usada a variável x.

6 Gerador automático de casos de testes

Foi desenvolvido um módulo que permite a geração automática de casos de teste na linguagem que definimos. Este gerador permite a geração de árvores sintaticamente corretas, respeitando variáveis previamente declaradas e não existirem declarações duplicadas da mesma variável (ao mesmo nível). Para implementar este módulo foi utilizado o tipo `Gen` e funções de geração da biblioteca *QuickCheck*. Em primeiro lugar, desenvolvemos os geradores dos tipos mais simples, ou seja, gerador de tipos, gerador de nomes, gerador de booleanos e gerador de inteiros. Posteriormente, implementamos um gerador de parâmetros, um gerador de
```

expressões que permite definir a profundidade máxima de uma expressão e um gerador de *statements* que permite definir o número máximo de *statements* a serem gerados. Por fim, implementamos o gerador de funções onde é mantido o estado de variáveis declaradas até ao momento e um gerador de programas que permite o número máximo de funções em cada programa.
Após executar `autoTestCaseGen 1 1 1` , obteve-se o caso de teste:

```
Prog [FunctionDeclaration Int "zrXTDIHB"
[Parameter Char "YNNQOXf",Parameter Char "wsAjiWOfR",Parameter Int "RT"]
[DeclAssign
"JjGAtsSn" (Boolean False)]]
```

Após correr `autoTestCaseGen 1 3 5` , obteve-se o caso de teste:

```
Prog [FunctionDeclaration String "Ftv" [Parameter Char "jaJOLIZtiF",Parameter Int "LBrh"]
[Return (Var "OUeAn"),While (Var "fEU")
[Assign "OUeAn" (Boolean False),DeclAssign Int "sGSlQgt" (Const 70)]]]
```

Acima é possível observar dois exemplos diferentes da execução do gerador de casos dados diferentes valores máximos para número de função, de *statements* e de profundidade das expressões.

7 Testes baseados em mutações

Os testes baseados em mutações são uma abordagem eficaz para verificar a robustez de conjuntos de testes e identificar possíveis falhas ou fragilidades nos programas. Nessa abordagem, mutações controladas são introduzidas no código-fonte original para avaliar a capacidade dos casos de teste em detetar essas alterações.

No contexto do módulo "MutatorGenerator", foram implementadas diversas funcionalidades para gerar mutações em expressões de programas. O objetivo dessas mutações é verificar se os testes existentes são capazes de identificar as alterações introduzidas, indicando assim a eficácia dos casos de teste.

O nosso gerador de mutações necessita de uma expressão original de forma a gerar mutações muito idênticas e difíceis de detetar. Por exemplo, é possível trocar um *Add* por um *Sub* e manter as expressões nas quais estas estão a aplicar. Este tipo de mutações utiliza funções do QuickCheck como o *elements*, mas também existem mutações mais básicas como a transição de um *True* para um *False*.

A expressão utilizada é fornecida a partir da árvore do programa. Utiliza-se uma estratégia TU idêntica à do *Bug counter* para agregar todas as expressões de um `Program` numa lista. Utiliza-se então a função *elements* para selecionar uma das expressões para mutar. Depois de aplicada esta mutação na expressão, utiliza-se o par (expressão original, expressão mutada) numa estratégia once_tdTP (aplica a mutação apenas uma vez) sobre o programa inicial de forma a obter um programa mutado.

8 Testes baseados em propriedades

Esta filosofia de testagem baseia-se na definição de propriedades sobre um programa que, a partir de casos de testes, se afere se esses propriedades se verificam ou não. No desenvolvimento deste módulo definimos as seguintes três propriedades:

- A AST gerada após o *parse* e após um *unparse* seguido de um *parse* é a mesma;
- Aplicar otimizações em *innermost* e em *full top down* não gera a mesma AST;
- Operações de *smell refactoring* e otimização são comutativas;

Para implementar estas propriedades foi necessário definir *Eq instances* sobre os tipos de dados das AST's de modo a definir como é que se efetua a comparação entre duas AST's. A primeira propriedade garante que a impressão de uma AST equivale ao programa correspondente a essa mesma AST. Além disso, o reprocessamento desse programa resulta na mesma AST. No caso da segunda propriedade provamos que a estratégia *top down* e *innermost* são diferentes na medida em que a primeira não consegue efetuar a otimização total das expressões do programa. Por último, a terceira propriedade permite verificar a comutatividade das operações de *smell refactoring* e otimização de expressões.

9 Discussão

Neste trabalho tivemos a oportunidade de escolher uma linguagem e desenvolver um processador. Aplicamos técnicas de programação estratégica para implementar mecanismos de otimização, *refactoring* e análise. Por fim, desenvolvemos um gerador de testes para a linguagem e implementamos dois métodos distintos de testagem. Consideramos que o catálogo de *smells*, otimizações, bugs e de dados a recolher poderia ser maior, mas isso teria implicado aumentar o *scope* da nossa linguagem. Como o objetivo era trabalhar com um pequeno e simplificado *subset* da linguagem escolhida achamos por bem manter essa simplicidade e não aumentar os catálogos.

10 Pós apresentação

Como foi sugerido durante a apresentação, o grupo adaptou o property check com novas funções que aceitam geradores de programas. Com isto o grupo foi capaz de corrigir alguns erros (nomeadamente com a primeira propriedade) em que não estavamos a involver expressões com parenteses ao dar unparse e depois as prioridades das operações eram perdidas. Com isto, foi inserido um novo problema que não nos deixou progredir mais na correção de erros, o nosso parser ficou lento. Na primeira fase o grupo deteteu que ao fazer parse de uma expressão como "((2))", o parser demora uma muito tempo. Este problema é proveniente da solução feita nas aulas e o prefessor Saraiva disse que não iamos ser prejudicados. No entanto ficamos impedidos de completar o primeiro property testing com sucesso. Em relação ao segundo, o gerador do grupo não é suficientemente complexo para gerar árvores que falhem na propriedade mas, no entanto, o "test_6"que o grupo criou demonstra esta propriedade a falhar. Devido á alta complexidade da nossa linguagem, o grupo decidiu não fezer um gerador mais abrangente e contentou-se com o cenário do "test_6". Para além disso, o grupo corrigiu o smell que estava incorreto durante a apresentação.