Desenvolvimento de um parser para linguagem PL/SQL para fins de análise estática

Oleksandr Sierov
(nrº 16991, regime diurno)

Orientação de
Joaquim G. P. Silva

Licenciatura em Engenharia em Sistemas Informáticos
Escola Superior de Tecnologia
Instituto Politécnico do Cávado e do Ave

Identificação do aluno

Oleksandr Sierov
Aluno número 16991, regime diurno
Licenciatura em Engenharia em Sistemas Informáticos

Orientação

Joaquim G. P. Silva

Informação sobre o Estágio

Checkmarx Portugal Lda
Avenida do Sol no18 1o Fração F, 4710-011 Braga
Eng. Xavier Pinho

Resumo

Neste relatório está descrito todo o percurso do estágio curricular. A proposta era constituída na elaboração da representação da árvore sintática da linguagem PL/SQL com a nova versão de Another Tool for Language Recognition V4 (ANTLR4).

Durante a fase inicial do estágio foram elaboradas pesquisas sobre os assuntos como PL/SQL, ANTLR4, análise sintática etc.

Após a fase inicial, procedeu se ao estudo da solução e o ao início da implementação do projeto. Foram desenvolvidas diversas *features* de PL/SQL em Abstract Syntax Tree (Árvore de Sintaxe Abstrata) (AST) no projeto. Em simultâneo, foram criados *samples* de PL/SQL para executar os testes posteriormente.

Por fim, foram elaborados testes unitários, capazes de correr em cima da árvore gerada após a implementação, para análise do trabalho desenvolvido.

Abstract

This report describes the entire course of the curricular internship. The proposal was constituted in the elaboration of the representation of the language's syntax tree PL/SQL with the new version of Another Tool for Language Recognition V4 (ANTLR4).

During the initial phase of the internship, research was elaborated on subjects such as PL/SQL, ANTLR4, syntactic analysis, etc.

After the initial phase, the solution was studied and the implementation of the project began. Several PL/SQL features have been developed in Abstract Syntax Tree (Árvore de Sintaxe Abstrata) (AST) on the project. Together, PL/SQL samples were created to run tests later.

Finally, unit tests were elaborated capable of running on top of the tree generated after implementation, for analysis of the work developed.

Agradecimentos

Neste tópico tenho um pouco mais de liberdade e vou utilizar da melhor forma. Foram 3 anos de estudo, um caminho longo, mas que olhando agora para trás parece que passou a correr. Quero agradecer a toda a minha família que me apoiou nestes 3 anos e também quero agradecer ao meu grupo de amigos que sempre foi unido e acolhedor no decorrer da licenciatura.

Conteúdo

1 Introdução **1**

1.1 Enquadramento 1

1.2 Objetivos do estágio 2

1.3 Plano de trabalhos 2

1.4 Estrutura do documento 3

2 Estudo e definição dos requisitos **5**

2.1 Segurança de *software* e análise estática 5

2.1.1 OWASP e CWE 5

2.1.2 Checkmarx SAST 6

2.1.3 Vulnerabilidades 6

2.2 Oracle PL/SQL 7

2.2.1 Estrutura básica 8

2.3 Descrição do problema 11

2.4 Especificação dos requisitos 11

2.4.1 Requisitos funcionais 11

2.4.2 Requisitos não funcionais 12

3 Arquitetura técnica **13**

3.1 Ferramentas utilizadas 13

3.2 Unified Abstract Syntax Tree 13

3.2.1 SyntaxNodes 15

3.2.2 SyntaxTokens 15

3.2.3 SyntaxKinds 15

3.3 DOM 15

3.4 Panorama geral 16

4 Implementação **17**

4.1 Metodologia 17

4.2 Blocos Anónimos e Declaração de Variáveis 17
4.3 Exceções e Atribuições 21
4.4 Expressões Unárias e Binárias 25
4.5 Declaração de Funções e If's 29
4.6 Invocações e Declarações For 34
4.7 Complementos 38

5 Análise dos resultados **41**
5.1 Apresentação da solução desenvolvida 41
5.1.1 Ferramentas 41
5.1.2 PL/SQL e gramática 42
5.1.3 *Visitors* e DOM 42
5.1.4 Testes unitários 43
5.2 Avaliação do nível de maturidade e cumprimento dos objetivos 43
5.3 Principais dificuldades encontradas 44
5.4 Parecer do Supervisor 44

6 Conclusão final **45**
6.1 Principais conclusões do estágio 45

Lista de Figuras

1.1 Gráfico de GANTT 3

3.1 Representação de uma variável 'i' na AST 14
3.2 Representação de uma variável 'i' na UAST 14
3.3 Cronograma de implementações 16

4.1 UAST - Bloco Anónimo 19
4.2 UAST - Declaração de Variável 21
4.3 UAST - Exceções 23
4.4 UAST - Exceções 25
4.5 UAST - Expressões Unárias 26
4.6 UAST - Expressões Binárias 29
4.7 UAST - Declaração de Função 31
4.8 UAST - Estrutura *If* com *Elsif* 33
4.9 UAST - Estrutura *If* com *Else* 33
4.10 UAST - Invocações 35
4.11 UAST - Ciclo For 37
4.12 Erro de *LowerCase* na gramática 38
4.13 Erro de *LowerCase Fixed* na gramática 38
4.14 *Parser* e *Lexer* antes da correção 39
4.15 *Parser* e *Lexer* depois da correção 39

5.1 Linguagens da *Tool AstTreeViewer* 41
5.2 Representação da DOM de um snippet de código PL/SQL 42
5.3 Testes Unitários 43

Lista de Tabelas

2.1 Requisitos funcionais 12
2.2 Requisitos não funcionais 12

5.1 Tabela de correspondências de linhas da figura 5.2 43

Listagens de Código

2.1 Bloco Anónimo Simples PL/SQL 8
2.2 Bloco Anónimo com Exceções PL/SQL 8
2.3 Declaração de Funções PL/SQL 9
2.4 Chamada da Função PL/SQL 10
2.5 Declaração de Procedimentos PL/SQL 10
2.6 Chamada do Procedimento PL/SQL 11
3.1 Declaração de variável 'i' em PL/SQL 14
3.2 *Visitor* de Demonstração 15
4.1 Declaração de Namespace e Class 18
4.2 PL/SQL - Bloco Anónimo 18
4.3 Gramática - Bloco Anónimo 18
4.4 Visitor - Bloco Anónimo 19
4.5 UAST *SyntaxNode* para Bloco Anónimo 19
4.6 PL/SQL - Declaração de Variável 20
4.7 Gramática - Declaração de Variável 20
4.8 Visitor - Declaração de Variável 20
4.9 UAST *SyntaxNode* para Declaração de Variável 21
4.10 PL/SQL - Exceções 22
4.11 Gramática - Exceções 22
4.12 Visitor - Exceções 22
4.13 UAST *SyntaxNodes* para Exceções 22
4.14 PL/SQL - Atribuições 24
4.15 Gramática - Atribuições 24
4.16 Visitor - Atribuições 24
4.17 UAST *SyntaxNodes* para Atribuições 24
4.18 PL/SQL - Expressões Unárias 25
4.19 Gramática - Expressões Unárias (Parte 1) 25
4.20 Gramática - Expressões Unárias (Parte 2) 26

4.21 Visitor - Expressões Unárias 26
4.22 UAST *SyntaxNodes* para Expressões Unárias 26
4.23 PL/SQL - Expressões Binárias 27
4.24 Gramática - Expressões Binárias (Parte 1) 27
4.25 Gramática - Expressões Binárias (Parte 2) 28
4.26 Visitor - Expressões Binárias 28
4.27 UAST *SyntaxNodes* para Expressões Binárias 28
4.28 PL/SQL - Declaração de Função 29
4.29 Gramática - Declaração de Função 30
4.30 Visitor - Declaração de Função 30
4.31 UAST *SyntaxNode* para Declaração de Função 30
4.32 PL/SQL - Estrutura If 31
4.33 PL/SQL - Estrutura Elsif 32
4.34 Gramática - Estruturas If/Else/Elsif 32
4.35 Visitor - Estruturas If 32
4.36 UAST *SyntaxNode* para Estruturas If 32
4.37 PL/SQL - Invocações 34
4.38 Gramática - Invocações 34
4.39 Visitor - Invocações 34
4.40 UAST *SyntaxNode* para Invocações 35
4.41 PL/SQL - Ciclo For 36
4.42 Gramática - Ciclo For 36
4.43 Visitor - Ciclo For 36
4.44 UAST *SyntaxNode* para Ciclos For 37

Siglas & Acrónimos

ANTLR4 Another Tool for Language Recognition V4. iii, v, 2, 11–13, 45

AST Abstract Syntax Tree (Árvore de Sintaxe Abstrata). iii, v, 2, 3, 13, 14, 41, 45

CD Continuous Delivery/Deployment(Entrega/Implementação Continua). 1, 2

CI Continuous Integration (Integração contínua). 1, 2

CWE Common Weakness Enumeration. 5, 6

CxSAST Checkmarx SAST. 6

IAST Interactive Application Security Testing (Teste de Segurança de Aplicações Interaticas). 1

OWASP Open Web Application Security Project. 5, 6

SAST Static Application Security Testing (Teste de Segurança de Aplicações Estáticas). xvii, 1, 6

SDLC Software Development Life Cycle. 6

UAST Unified Abstract Syntax Tree (Árvore de Sintaxe Abstrata Unificada). xv, xvi, 3, 11, 13–15, 17, 19–26, 28–33, 35, 37, 42

Glossário

DevOps É uma prática de engenharia de software com o objetivo de coordenar e colaborar para produzir produtos melhores e mais fiáveis.. 2, 6

DevSecOps Integra práticas de segurança dentro do processo DevOps.. 2

1. Introdução

Este documento foi realizado com objetivo de dar a conhecer o Estágio Curricular realizado na empresa em *software* de segurança, Checkmarx Portugal, para obtenção do grau Licenciado no curso de Licenciatura em Engenharia de Sistemas Informáticos do Instituto Politécnico do Cavado e do Ave.

Estágio Curricular, como o nome indica, é uma atividade curricular que faz parte do percurso académico de um estudante, sendo supervisionada por alguém qualificado por parte da empresa na qual decorre o estágio. Tem como objetivos colocar em prática a maior parte da matéria lecionada durante o percurso de aprendizagem teórica e também, confrontar o estagiário com situações reais de trabalho. Hoje em dia, a experiência de trabalho é cada vez mais valorizada, sendo assim, estagio curricular é uma mais-valia para uma pessoa recém-formada.

Neste relatório será descrito todo o caminho percorrido, de forma sucinta, durante o período que o estagiário esteve na empresa como estagiário. Será realizado de uma forma formal e breve para que seja compreendido por qualquer leitor.

1.1 Enquadramento

A Checkmarx, empresa na qual foi realizado este projeto como estágio curricular, está situada na Avenida do Sol nº18 1º Fração F, em Braga, Portugal. O estágio foi realizado durante o 2 Semestre do 3º ano curricular, tendo o início no dia 22 de fevereiro e terminado dia 9 de junho.

A Checkmarx foi fundada em Israel no ano 2006. Atualmente está presente em vários países espalhados pelo mundo. Checkmarx Portugal foi fundada em 2016 por um grupo pequeno de pessoas a crescer cada vez mais. A Checkmarx tem como objetivo ajudar empresas a reduzir os riscos de vulnerabilidades de *software* dispondo de um conjunto de produtos para análise de código não compilado e realização de testes sobre o mesmo.

A Checkmarx oferece como principais serviços (Checkmarx, 2013):

- Segurança

  Portal de segurança de *software* unificada mais completa do sector, que inclui de uma forma sólida o Static Application Security Testing (Teste de Segurança de Aplicações Estáticas) (SAST), Interactive Application Security Testing (Teste de Segurança de Aplicações Interaticas) (IAST) e entre outras para incorporar a segurança do *software* em todo o pipeline de Continuous Integration (Integração contínua) (CI)/Continuous Delivery/Deployment(Entrega/Implementação

Continua) (CD).

- Melhor adaptação para DevSecOps

  Líder da indústria no fornecimento de verificação de segurança automatizada como parte do processo DevOps.

- Velocidade de DevOps

  Permite gerir a exposição do *software* na velocidade dos DevOps - produção de forma rápida, segura e eficiente.

A Checkmarx oferece uma plataforma perfeita para ambientes DevOps e CI e redefine o papel da segurança no ciclo de desenvolvimento de *software*. A Checkmarx oferece um conjunto de produtos inovadores e de testes de segurança ao código de *software*. Também contribui para a redução significativa do risco de vulnerabilidades durante o ciclo de desenvolvimento de *software*. Permite que os próprios desenvolvedores testem o seu código de *software* para as questões de segurança, usando os resultados para correcção durante o próprio ciclo de desenvolvimento (Checkmarx, 2016).

1.2 Objetivos do estágio

Como todo o estágio curricular, para além dos objetivos globais do mesmo, também são citados objetivos por parte da empresa a serem executados pelo estagiário.

Objetivos colocados por parte da empresa:

1. Integração na área da empresa e respetivas formações.
2. Desenvolvimento de um *parser* para a linguagem PL/SQL para fins de análise estática.
3. Executar testes sobre o conteúdo desenvolvido em ponto (2).

1.3 Plano de trabalhos

No âmbito do estágio, foram identificadas as seguintes tarefas:

- Formação nas atividades da empresa e na análise estática - realização de acções de formação no negócio e *software* da empresa.
- Formação em linguagem PL/SQL - conclusão de curso de PL/SQL na Udemy.
- Formação na ferramenta Another Tool for Language Recognition V4 (ANTLR4) - utilização juntamente com IntelliJ IDE.
- Criação de *samples* de PL/SQL - exemplos de código para usar como teste.
- Preparação do projeto - criação de uma *flag* para usar o PL/SQL rescrito.
- Inserção da gramática do PL/SQL - ajustar a gramática com o projeto.
- Criação de *visitors* - implementação de funções que serão a base para gerar a AST.

- Criação da nova *flag* para *AST Tool Viewer* da Checkmarx - adição da *flag* para ver a AST.
- Preparação do projeto para Testes Unitários - testes da PL/SQL AST.
- Criação de testes unitários e a sua *"run"* na AST.

Para planear a execução das tarefas, foi elaborado um diagrama de Gantt apresentado nas Figura 1.1

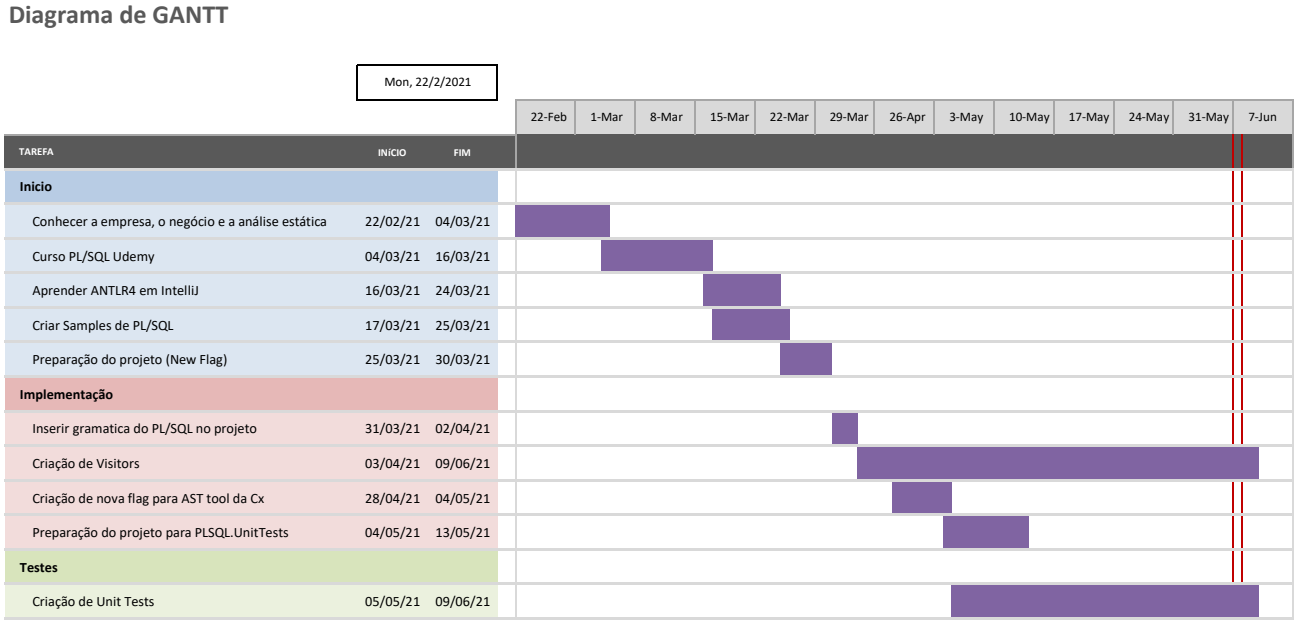


Figura 1.1: Gráfico de GANTT

1.4 Estrutura do documento

Este documento foi subdividido em 6 capítulos, inclusive o atual, para uma melhor estruturação do mesmo.

No capítulo seguinte será mencionado o estudo de matérias abordadas durante a primeira etapa do estágio. Ainda no mesmo, será analisado o problema e a definição de requisitos.

No capítulo 3 são definidas as ferramentas utilizadas, a demonstração e explicação de uma (UAST) e o panorama de desenvolvimento de funcionalidades.

De seguida, o capítulo 4 demonstra a implementação dos *visitors* e as representações das árvores.

No capítulo 5 é feita uma análise a tudo o que foi desenvolvido durante o estagio curricular e feita uma observação em relação ao cumprimento de objetivos.

Por fim, o capítulo 6 apresenta uma conclusão final.

2. Estudo e definição dos requisitos

A segurança do *software* é uma ideia implementada para proteger o *software* contra ataques maliciosos e outros riscos de *hacker* para que o *software* continue a funcionar corretamente sob tais riscos potenciais. A segurança é necessária para fornecer integridade, autenticação e disponibilidade.

O *software* de segurança é um termo amplo que engloba um conjunto de diferentes tipos de *software* que fornecem dados e segurança de computador e rede em várias formas. O *software* de segurança pode proteger um computador contra vírus, *malware*, utilizadores não autorizados e outras explorações de segurança originárias da Internet. Os tipos de *software* de segurança incluem *software* antivírus, *software* de *firewall*, *software* de segurança de rede, *software* de segurança da Internet, *software* de remoção e proteção de *malware/spamware*, *software* criptográfico, e muito mais.

Em ambientes de computação de utilizadores finais, o *software* antivírus e anti-correio publicitário não solicitado é o tipo de *software* mais comum utilizado, enquanto os utilizadores das empresas adicionam um sistema de deteção de *firewall* e intrusão em cima dele (Techopedia, 2003).

2.1 Segurança de *software* e análise estática

É difícil falar de desenvolvimento de *software* sem fazer a devida questão: "O quão segura é a minha aplicação? Como posso proteger a mim como desenvolvedor e aos meus clientes?". Segurança é um dos pilares mais importantes na construção de qualquer *software*, sendo um requisito necessário para que o cliente se sinta protegido na sua utilização.

"Nada é 100% seguro". É difícil dizer que isto é uma citação especifica de alguém porque qualquer *expert* de segurança informática afirma o mesmo. É uma realidade que nos persegue desde o inicio do aparecimento da tecnologia. Então o que fazer?

Entramos assim em dois conceitos que foram criados há bastante tempo e é impossível não os mencionar ao falar de segurança: OWASP e CWE, parecidos e diferentes ao mesmo tempo.

2.1.1 OWASP e CWE

Open Web Application Security Project (OWASP) é uma comunidade online, sem fins lucrativos, que disponibiliza artigos, metodologias, documentação, ferramentas, entre outros, no âmbito de segurança e aplicações web. OWASP desenvolveu uma lista *"TOP*

10" dos riscos de segurança mais críticos para aplicações web usada por desenvolvedores na segurança de aplicações.

Common Weakness Enumeration (CWE) é uma lista com tipos comuns de fraquezas e vulnerabilidades de *software* e *hardware*. O projeto é sustentado por uma comunidade com objetivos de compreender falhas no *software* e criar soluções e ferramentas automatizadas que podem ser utilizadas para prevenir, identificar e/ou corrigir essas falhas.

Tanto a CWE como a OWASP são utilizadas pelos desenvolvedores na maior parte das vezes em conjunto, pois ambas se complementam. A CWE estuda as vulnerabilidades de uma maneira mais profunda, enquanto a OWASP enquadra-se na segurança de uma forma mais abrangente.

2.1.2 Checkmarx SAST

Checkmarx SAST (CxSAST) é uma solução de análise estática de grau empresarial, flexível e precisa, utilizada para identificar centenas de vulnerabilidades de segurança em código personalizado. É utilizado por equipas de desenvolvimento, DevOps e de segurança para fazer o *scan* do código fonte no ciclo de vida do desenvolvimento do *software* SDLC, identificar vulnerabilidades e fornecer conhecimentos acionáveis para as corrigir. Suporta mais de 25 linguagens de codificação e de *scripting* e as suas estruturas com configuração zero para fazer *scan* a qualquer linguagem de programação (Checkmarx, 2021).

2.1.3 Vulnerabilidades

Uma das vulnerabilidades tratadas pela Checkmarx é o código obsoleto, isto é, código antigo e não funcional que pode levar a possíveis quebras no funcionamento da aplicação. De seguida, alguns exemplos, de uma forma detalhada, de vulnerabilidades de *software*, também analisadas pela Checkmarx. Os exemplo são parte da lista *TOP 10* da OWASP (OWASP, 2017).

- **Code Injection**

  Exemplo SQL:

  ```
  String query = "SELECT * FROM accounts
  WHERE custID='" + request.getParameter("id") + "'";
  ```

  Neste exemplo de código SQL conseguimos entender que existe a possibilidade de injeção de código, pois não existe qualquer verificação do parâmetro usado na procura. Alterado o parâmetro 'id' no pedido para

  ```
  http://example.com/app/accountView?id=' ou '1'='1
  ```

  Fará com que a *query* retorne todos os registos da tabela.

- **Cross-Site Scripting (XSS)**

  Normalmente um ataque XSS é bem-sucedido quando o invasor consegue executar HTML e JavaScript arbitrários no navegador da vítima. Isto acontece assim que a vítima interagir com um *link* malicioso ou anúncio.

  Existem 3 tipos de XSS: Reflected XSS, Stored XSS e DOM XSS.

  Imaginemos que uma página não executa a verificação de parâmetros de entrada. O seguinte exemplo é de alteração da língua numa página.

  A página é invocada pelo URL seguinte

  ```
  http://www.some.site/page.html?default=French
  ```

  Enviamos então um *link* malicioso para a vítima

  ```
  http://www.some.site/page.html?default=
  <script>alert(document.cookie)</script>
  ```

  Assim que houver um click, o browser envia o seguinte pedido

  ```
  /page.html?default=<script>alert(document.cookie)</script>
  ```

  O servidor responde com a página que contém o código Javascript acima. O código Javascript original na página não espera que o parâmetro default contenha marcação HTML e, como tal, simplesmente a decodifica e reproduz na página (DOM) em tempo de execução. O browser renderiza a página resultante e executa o script do atacante.

  ```
  alert(document.cookie)
  ```

2.2 Oracle PL/SQL

PL/SQL é uma linguagem procedural criada especificamente para incluir instruções SQL na sua sintaxe. Os blocos de código PL/SQL são compilados pelo servidor de base de dados Oracle e armazenados no mesmo (Oracle, 2008).

PL/SQL poupa tempo na produção e *debug* através de características fortes, tais como tratamento de exceções, encapsulamento, e tipos de dados orientados a objetos. As aplicações escritas em PL/SQL são totalmente portáteis, o que é uma vantagem.

2.2.1 Estrutura básica

Com PL/SQL, podemos usar SQL para manipular dados do Oracle e controlar o fluxo para processar os dados. Além disso, podemos declarar constantes e variáveis, definir procedimentos e funções. Assim, PL/SQL combina a manipulação dos dados de SQL com o processamento de dados das linguagens procedurais.

Blocos lógicos são unidades de estruturas básicas no PL/SQL sendo, blocos anónimos, funções e procedimentos exemplos dos mesmos. Cada bloco permite incluir declarações relacionadas ao mesmo, após a execução do bloco essas declarações deixam de existir.

De seguida, alguns exemplos de PL/SQL.

Bloco Anónimo Simples

No excerto de código em Listagem 2.1 observa-se a declaração de um bloco anónimo. A sintaxe "DECLARE" significa que o bloco contem o conjunto de instruções pertencente a declarações.

De seguida, vemos a sintaxe "BEGIN". É aqui que são escritas as instruções SQL que queremos que sejam executadas no bloco. Neste exemplo apenas temos um *"Assignment"*.

```
DECLARE
    vNumber INT := 10;
    vMult INT;
BEGIN
    vMult := vNumber * 2;
END;
```

Listagem 2.1: Bloco Anónimo Simples PL/SQL

Bloco Anónimo e Exceções

No excerto de código em Listagem 2.2 existe o uso de um `ACCEPT`, isto é um pedido para inserção de valores como por exemplo um `Console.Read();` em C#.

De seguida, a declaração de variáveis. A variável `VEMPLOYEE_ID` é inicializada com a variável que recebe o valor inserido pelo utilizador em cima.

Depois, temos a secção de comandos SQL. Neste caso é efetuado um `SELECT` onde adiciona nas variáveis o primeiro e o último nome do empregado com o código respetivo, *keyword* `WHERE`.

Por fim, surge a *keyword* `EXCEPTION`, onde são tratadas as excepções. Neste exemplo, temos um tipo de exceção específico `NO_DATA_FOUND` que é lançada no caso de não existir nenhum empregado na tabela `EMPLOYEES` ao executar o comando `SELECT`. Existe ainda um outro tratamento de exceções neste exemplo, `OTHERS`, que é lançado no caso de qualquer falha diferente da `NO_DATA_FOUND`.

```
ACCEPT PEMPLOYEE_ID PROMPT 'Employee ID:'
DECLARE
    VFIRST_NAME    EMPLOYEES.FIRST_NAME%TYPE;
    VLAST_NAME     EMPLOYEES.LAST_NAME%TYPE;
```

```
    VEMPLOYEE_ID  EMPLOYEES.EMPLOYEE_ID%TYPE :=
                                &PEMPLOYEE_ID;
BEGIN
    SELECT FIRST_NAME, LAST_NAME
    INTO   VFIRST_NAME, VLAST_NAME
    FROM   EMPLOYEES
    WHERE  EMPLOYEE_ID = VEMPLOYEE_ID;

    DBMS_OUTPUT.PUT_LINE('EMPLOYEE: ' ||
    VFIRST_NAME || ' ' || VLAST_NAME);

EXCEPTION
WHEN NO_DATA_FOUND
THEN
    RAISE_APPLICATION_ERROR(-20001,
    'EMPLOYEE NOT FOUND, ID = ' ||
    TO_CHAR(VEMPLOYEE_ID));

WHEN OTHERS
THEN
    RAISE_APPLICATION_ERROR(-20002,
    'ORACLE ERROR - ' || SQLCODE
    || SQLERRM);

END;
```

Listagem 2.2: Bloco Anónimo com Exceções PL/SQL

Declaração de Funções

O excerto de código em Listagem 4.28 demonstra uma declaração de uma função, que recebe como parâmetro o `PEMPLOYEE_ID` e retorna uma variável do tipo `NUMBER`. A função executa um `SELECT` que retorna o valor de `SALARY` para a variável `VSALARY` que de seguida é retornada da função.

```
CREATE OR REPLACE FUNCTION FNC_EMPLOYEE_SALARY
(PEMPLOYEE_ID   IN NUMBER)
RETURN NUMBER
IS
    VSALARY  EMPLOYEES.SALARY%TYPE;
BEGIN
    SELECT SALARY
    INTO   VSALARY
    FROM   EMPLOYEES
    WHERE EMPLOYEE_ID = PEMPLOYEE_ID;
    RETURN (VSALARY);
EXCEPTION
WHEN NO_DATA_FOUND
THEN
    RAISE_APPLICATION_ERROR(-20001,
    'EMPLOYEE NOT FOUND, ID = ' ||
```

```
        TO_CHAR(PEMPLOYEE_ID));
    WHEN OTHERS
    THEN
        RAISE_APPLICATION_ERROR(-20002, 'Error ' || SQLCODE
        || ' - ' || SQLERRM);
    END;
```

Listagem 2.3: Declaração de Funções PL/SQL

No excerto em Listagem 2.4 é possível ver uma chamada da função declarada na Listagem 4.28.

```
DECLARE
    VEMPLOYEE_ID   EMPLOYEES.EMPLOYEE_ID%TYPE := 101;
    VSALARY        EMPLOYEES.SALARY%TYPE;
BEGIN
    VSALARY := FNC_EMPLOYEE_SALARY(VEMPLOYEE_ID);
    DBMS_OUTPUT.PUT_LINE('Salary: ' || VSALARY);
END;
```

Listagem 2.4: Chamada da Função PL/SQL

Declaração de Procedimentos

No excerto de código em Listagem 2.5 demonstra uma declaração de um procedimento. Ao contrario de uma função (Listagem 4.28), o procedimento não retorna qualquer valor, servindo apenas para guardar e executar comandos quando chamado. Neste exemplo, o procedimento recebe 2 parâmetros a partir da sua chamada. `PFIRST_NAME` e `PLAST_NAME` são os parâmetros que vêm de fora. Este procedimento executa um `INSERT` dos parâmetros que recebe na tabela `EMPLOYEES`

```
CREATE OR REPLACE PROCEDURE PRC_INSERT_EMPLOYEE
(PFIRST_NAME     IN VARCHAR2,
 PLAST_NAME      IN VARCHAR2)
IS
BEGIN
    INSERT INTO EMPLOYEES (
    EMPLOYEE_ID,
    FIRST_NAME,
    LAST_NAME)
VALUES (
    EMPLOYEES_SEQ.NEXTVAL,
    PFIRST_NAME,
    PLAST_NAME);
EXCEPTION
WHEN OTHERS
THEN
    RAISE_APPLICATION_ERROR(-20001, 'Oracle Error'
    || SQLCODE || SQLERRM);
END;
```

Listagem 2.5: Declaração de Procedimentos PL/SQL

No excerto em Listagem 2.6 é possível ver uma chamada do procedimento declarado na Listagem 2.5.

```
BEGIN
    PRC_INSERT_EMPLOYEE('David', 'Bowie');
    COMMIT;
END;
```

Listagem 2.6: Chamada do Procedimento PL/SQL

2.3 Descrição do problema

De momento, o *parsing* de PL/SQL encontra-se desatualizado, utilizando a versão anterior de ANTLR que não permite mais, de forma correta, a representação na UAST e transformação para a DOM. Tambem não suporta diversas *features* como declarações de variáveis globais permitidas sintaticamente pela linguagem. Foi então proposto desenvolver a rescrita do *parser* de PL/SQL com a versão mais recente da ANTLR4 com objetivo de superar as dificuldades da versão antiga e melhorar a sua estruturação.

2.4 Especificação dos requisitos

A especificação de requisitos de *software* deve ser dividida em duas partes, requisitos funcionais e requisitos não funcionais.

Os **requisitos funcionais** são, declarações das funcionalidades do sistema que devem fornecer como o sistema deverá responder a cada entrada e como se deverá comportar em dadas situações (Sommerville, 2015).

Os **requisitos não funcionais** são os constrangimentos às funcionalidades ou funções do sistema, como standards, restrições de tempo e restrições do processo de desenvolvimento, incluindo as propriedades comportamentais que o sistema deve ter, como desempenho, segurança e fiabilidade (Sommerville, 2015).

2.4.1 Requisitos funcionais

Na tabela seguinte, estão listados os requisitos funcionais encontrados após uma breve análise das funcionalidades principais necessitadas (Tabela 2.1).

ID	Descrição	MoSCoW
RF 01	Representação de Blocos Anónimos	Must
RF 02	Representação da Declaração de Variáveis	Must
RF 03	Representação das Exceções	Must
RF 04	Representação de Atribuições	Must
RF 05	Representação de Expressões Unárias	Must
RF 06	Representação de Expressões Binárias	Must
RF 07	Representação da Declaração de Funções	Must
RF 08	Representação de Declarações If	Must
RF 09	Representação de Invocações	Must
RF 10	Representação de Declarações For e Foreach	Must

Tabela 2.1: Requisitos funcionais

2.4.2 Requisitos não funcionais

De seguida, foi efetuado um levantamento de requisitos não funcionais, os quais são apresentados na tabela seguinte (Tabela 2.2).

ID	Descrição	MoSCoW
RnF 01	Implementação em Linguagem C#	Must
RnF 02	ANTLR4 requerido	Must

Tabela 2.2: Requisitos não funcionais

3. Arquitetura técnica

Neste tópico será elaborada uma descrição breve sobre as ferramentas que foram utilizadas e tecnologias que foram propostas para serem implementadas através de uma demonstração do panorama geral.

3.1 Ferramentas utilizadas

- **Visual Studio**, IDE utilizado na realização do projeto, produzido em C#.
- **IntelliJ IDEA**, é uma ferramenta para analise de código, foi utilizado para trabalhar juntamente com o ANTLR4.
- **ANTLR4**, é um poderoso gerador de analisadores para leitura, processamento, execução, ou tradução de texto estruturado ou ficheiros binários. É amplamente utilizado para construir linguagens, ferramentas, e *frameworks*. A partir de uma gramática, ANTLR gera um analisador que pode construir e percorrer *parse trees* (Parr, 2014).
- **Overleaf**, é um editor LaTeX *online* usado para escrever, editar e publicar qualquer tipo de documentos. Fornece uma enorme gama de modelos de documentos, disponíveis para utilização.

3.2 Unified Abstract Syntax Tree

A Unified Abstract Syntax Tree (Árvore de Sintaxe Abstrata Unificada), como o nome indica, é uma união de todas as AST's. É um projeto com o objetivo de unificar todas as AST's desenvolvidas de todas as linguagens. É a camada superior de abstração, para a representação partes de código ou da estrutura de um programa. Neste caso, o termo abstrato significa a possibilidade da representação de qualquer tipo de estrutura de qualquer tipo de linguagens de programação utilizando a mesma representação para a construir.

As AST's são fundamentais na forma como um compilador funciona. Também são utilizadas na análise de código, ferramentas para gerar código e ferramentas de refatoração. Cada *Syntax Tree* é composta de *syntax nodes* e *syntax tokens*.

Na listagem 3.1 está um excerto de código não compilável da declaração de uma variável 'i' que servirá como exemplo de representação na AST e UAST.

```
DECLARE
    i INT;
```

Listagem 3.1: Declaração de variável 'i' em PL/SQL

Na figura 3.1, está representada a AST da declaração de uma variável i em PL/SQL.

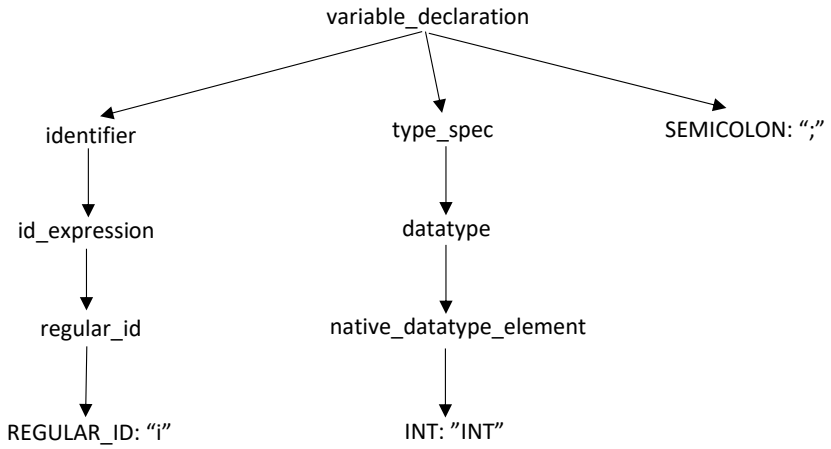


Figura 3.1: Representação de uma variável 'i' na AST

Na figura 3.2, está representada a UAST da declaração de uma variável i em PL/SQL. É de notar a existência do *Kind* que é abordado na secção 3.2.3, `VariableDeclaration[...]`. Neste caso é o *SyntaxNode* que vai ser abordado na secção 3.2.1 e o `i[Type:IdentifierNameSyntax, Kind:IdentierToken, Line:2, Column:2]` é um *SyntaxToken* abordado na secção 3.2.2.

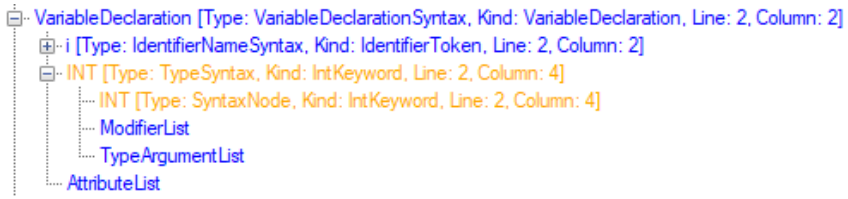


Figura 3.2: Representação de uma variável 'i' na UAST

Para obtermos esta representação foi necessário implementar o *visitor* correspondente. Neste caso, foi necessário implementar o `VisitVariable_declaration` demonstrado na Listagem 3.2.

Este *visitor*, como o nome indica, visita o nó `variable_declaration` para criar um *SyntaxNode* representável na UAST. Para isso é usada a classe `VariableDeclarationSyntax()` (linha 8 e 12). Antes de ser retornado é preciso trazer os valores pertencentes à variável, de acordo com o exemplo anterior, listagem 3.1. Apenas temos o nome da variável e o tipo da variável, sendo assim, irá ser feito apenas o *visit* ao `identifier` (linha 4) correspondente ao nome da variável e o *visit* ao `type_spec` (linha 5) correspondente ao tipo da variável. Caso existisse um *assign* a variável na sua declaração, por exemplo, ao invés de ser [`i INT;`] fosse [`i INT = 5;`], seria efetuado o *visit* ao `default_value_part` (linha 10/11) que corresponde à parte do *assign* da variável que está a ser declarada.

```
1  public override SyntaxNode VisitVariable_declaration([NotNull]
2  PLSqlParser.Variable_declarationContext context)
3  {
4      var identifier = Visit(context.identifier()) as IdentifierNameSyntax;
5      var type = Visit(context.type_spec()) as TypeSyntax;
6      if (context.default_value_part() == null)
7      {
8          return new VariableDeclarationSyntax(identifier, null, type);
9      }
10     var equals =
11         Visit(context.default_value_part()) as EqualsValueClauseSyntax;
12     return new VariableDeclarationSyntax(identifier, equals, type);
13 }
```

Listagem 3.2: *Visitor* de Demonstração

3.2.1 SyntaxNodes

Os *SyntaxNodes* são os principais componentes de uma *SyntaxTree* e são representados por uma classe nominada por *SyntaxNode*. *SyntaxNodes* são utilizados para a representação de toda e qualquer construção sintética de uma linguagem, por exemplo, declarações, *assigns*, expressões, classes, *statements* etc. Qualquer *SyntaxNode* é um nó não terminal, ou seja, tem sempre um nó filho e, se não for o nó raiz também, tem um nó pai associado. Os nós raiz, apesar de não ter nó pai, também se denominam de *SyntaxNodes*.

3.2.2 SyntaxTokens

Os *SyntaxTokens* são o outro componente de um *SyntaxTree* e são representados por uma classe nominada por *SyntaxToken*. *SyntaxTokens* nunca podem ser pais e maioritariamente representam palavras chaves, identificadores, pontuações e literais. *SyntaxToken* para alem de conter o seu tipo também é preenchido com o Texto que representa, a Linha e a Coluna onde se encontra no código.

3.2.3 SyntaxKinds

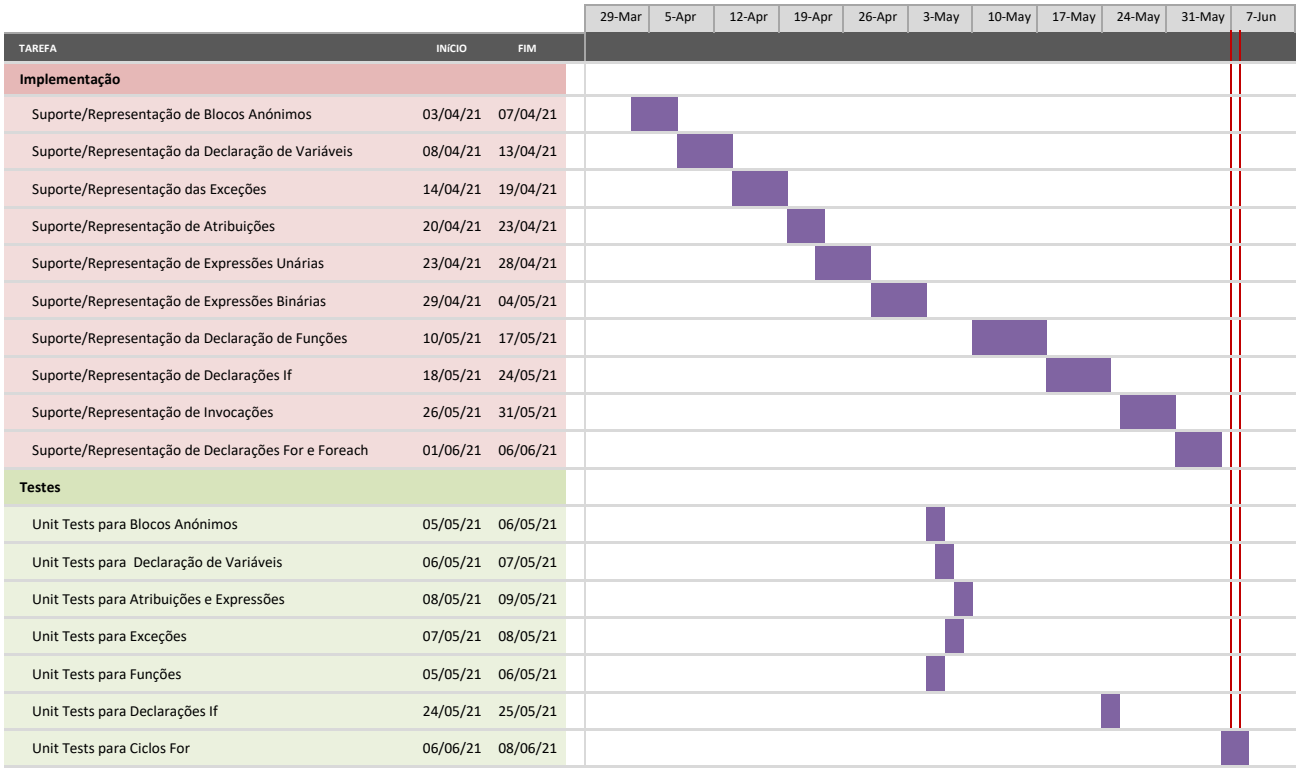
Os *SyntaxKinds* são parte tanto dos *SyntaxTokens* como dos *SyntaxNodes*. Estes identificam o elemento que o mesmo representa. Existe uma lista de todos os *Kinds* possíveis a ser usados. Esta lista está armazenada na *Engine* através de uma enumeração valida para todas as linguagens de programação.

3.3 DOM

A DOM é uma representação intermédia. Depois de um código fonte ser processado para a UAST, este é reescrito na DOM. A representação na DOM é elaborada através da linguaguem C# e é por cima da DOM que serão executadas as *queries*.

3.4 Panorama geral

Após o levantamento de requisitos na secção 2.4 foi elaborado um cronograma de implementação das funcionalidades analisadas (Ver Figura 3.3).



TAREFA	INÍCIO	FIM
Implementação		
Suporte/Representação de Blocos Anónimos	03/04/21	07/04/21
Suporte/Representação da Declaração de Variáveis	08/04/21	13/04/21
Suporte/Representação das Exceções	14/04/21	19/04/21
Suporte/Representação de Atribuições	20/04/21	23/04/21
Suporte/Representação de Expressões Unárias	23/04/21	28/04/21
Suporte/Representação de Expressões Binárias	29/04/21	04/05/21
Suporte/Representação da Declaração de Funções	10/05/21	17/05/21
Suporte/Representação de Declarações If	18/05/21	24/05/21
Suporte/Representação de Invocações	26/05/21	31/05/21
Suporte/Representação de Declarações For e Foreach	01/06/21	06/06/21
Testes		
Unit Tests para Blocos Anónimos	05/05/21	06/05/21
Unit Tests para Declaração de Variáveis	06/05/21	07/05/21
Unit Tests para Atribuições e Expressões	08/05/21	09/05/21
Unit Tests para Exceções	07/05/21	08/05/21
Unit Tests para Funções	05/05/21	06/05/21
Unit Tests para Declarações If	24/05/21	25/05/21
Unit Tests para Ciclos For	06/06/21	08/06/21

Figura 3.3: Cronograma de implementações

4. Implementação

Toda a implementação foi monitorizada através de 5 *sprints* realizadas ao longo do estado da implementação. Assim será dividido este capitulo. Em cada *sprint* foram implementados diferentes *visitors* para a linguagem PL/SQL, no fim uma secção de complementos as implementações realizadas.

4.1 Metodologia

De 15 em 15 dias era efetuada uma *demo* com apresentação do que foi implementado nas ultimas duas semanas. As *demos* consistiam na apresentação de um *PowerPoint* com a implementação efetuada, os mesmos foram armazenados e podem ser acedidos pelas seguintes *hiperligações* (Google Slides).

Sprint Demo 1 - t.ly/9sr7

Sprint Demo 2 - t.ly/yrmM

Sprint Demo 3 - t.ly/Ks46

Sprint Demo 4 - t.ly/1j47

Sprint Demo 5 - t.ly/cTon

4.2 Blocos Anónimos e Declaração de Variáveis

Na primeira *sprint* ao que consta as implementações de *visitors*. Para alem de implementação dos *visit* ao bloco anónimo e a declaração de variáveis, também foi implementado um *visit* (apenas uma função normal) para ser criada uma *SyntaxNode* de *NameSpace* e *Class*. Como o PL/SQL não necessita de uma *NameSpace* ou *Class* para funcionar, foi necessária a sua criação de uma forma predefenida, `DEFAULT_CLASS e DEFAULT_NAMESPACE`. A UAST funciona à base de hierarquias e é obrigatório a existência de uma *Class* e *NameSpace* antes de código compilável do PL/SQL, para que seja possível a sua representação.

Namespace && Class

Para implementar a declaração de Namespace e Class predefinidos foram utilizadas as seguintes classes, listagem 4.1

```
1  NamespaceDeclarationSyntax(token, name, members)
2  ClassDeclarationSyntax(classToken, className, members)
```

Listagem 4.1: Declaração de Namespace e Class

As *class's* da linha 1 e 2 da Listagem 4.1 recebe:

- **token / classToken:** *SyntaxToken* com a *name* `DEFAULT_NAMESPACE/CLASS` para a sua criação utilizando o *SyntaxKind* correspondente.
- **name / className:** Identificador da *Namespace/Class* utilizando a mesma *name* do ponto superior.
- **members:** Lista de membros que pertencem a *Namespace/Class*. Neste caso, o membro da *Namespace* será apenas a *Class* criada por definição, e os membros da *Class* serão todo o código fonte do `.sql`.

Bloco Anónimo

Na listagem 4.2 temos um exemplo da utilização de um bloco anónimo. Na listagem 4.3 encontra-se a gramática associada, que faz *parser* do exemplo. Foi efetuado um *visitor* ao *anonymous_ block* (linha 1 de 4.3 e Listagem 4.4).

```
1  DECLARE
2      (Declarations)
3  BEGIN
4      (Body)
5  EXCEPTION
6      (Exceptions)
7  END;
```

Listagem 4.2: PL/SQL - Bloco Anónimo

Na gramática do bloco anónimo conseguimos compreender que a sintaxe `DECLARE` e `EXCEPTION` são opcionais. O `seq_of_declare_specs` é uma lista de declarações, `seq_of_statements` é uma lista de `statements` e o `exception_handler` é uma lista de exceções.

```
1 anonymous_block
2     : (DECLARE seq_of_declare_specs)? BEGIN seq_of_statements
3       (EXCEPTION exception_handler+)? END SEMICOLON
4     ;
```

Listagem 4.3: Gramática - Bloco Anónimo

```
public override SyntaxNode VisitAnonymous_block
([NotNull] PLSqlParser.Anonymous_blockContext context){...}
```

Listagem 4.4: Visitor - Bloco Anónimo

Para fazer a representação do *SyntaxNode* do bloco anónimo na UAST foi utilizada a *class* representada na Listagem a seguir (4.5).

```
MethodDeclarationSyntax(methodName, parameters, body)
```

Listagem 4.5: UAST *SyntaxNode* para Bloco Anónimo

A *class* da Listagem 4.5 recebe:

- **methodName:** é um identificador. Sendo um bloco anónimo não tem um nome definido, então resolveu se criar um *Hash* para identificar os diferentes blocos anónimos num único ficheiro `.sql`
- **parameters:** são os parâmetros que entram no método. Será utilizado posteriormente, mas no caso da implementação do bloco anónimo é desnecessário.
- **body:** Aqui, entra a lista que provem do *visit* ao `seq_of_declare_specs` e `seq_of_statements` unida. Ambos os *visit* são definidos por uma lista em *SyntaxNode* denominada *BlockSyntax*.

A *keyword* `EXCEPTION` será tratada posteriormente.

Na figura 4.1 é possível ver toda a UAST que representa um bloco anónimo.

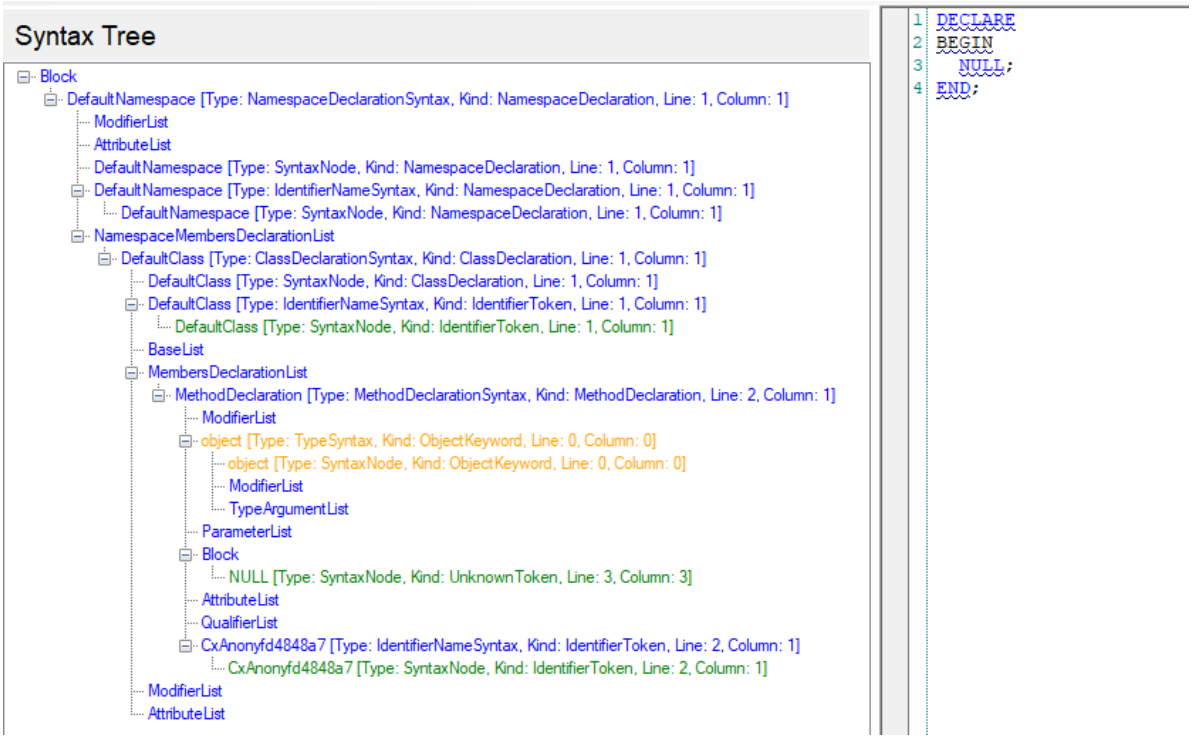


Figura 4.1: UAST - Bloco Anónimo

Declaração de Variáveis

Na listagem 4.6 temos um excerto de código com declaração de diferentes variáveis com diferentes tipos. Na listagem 4.7 encontra-se a gramática associada a declaração, que faz *parser* do exemplo. Foi efetuado um *visitor* à *variable_ declaration* (linha 1 de 4.7 e Listagem 4.8) e também a todos os nós envolventes para o seu correto funcionamento.

```
1     DECLARE
2         vFirstName          EMPLOYEES.FIRST_NAME%TYPE;
3         i                   INT := 5;
4         vTotal              BINARY_FLOAT := 15000.05;
5         vBool               BOOLEAN := TRUE;
6     (...)
```

Listagem 4.6: PL/SQL - Declaração de Variável

Na gramática da declaração de variável conseguimos compreender que a sintaxe/*keyword* `CONSTAN`, `NOT NULL_` e `default_value_part` são opcionais devido ao uso do `?`(ponto de interrogação).

O `identifier` indica o nome da variável. O seu *visit* passa por diferentes ramos até embater com o nó terminal que será um *token*. Dentro do *visitor* da declaração da variável será recebido um `IdentifierNameSyntax` apos ser feito o *visit* ao `identifier`.

O `type_spec` é o responsável para identificar o tipo da variável. Existe uma lista com os tipos possíveis de utilização em PL/SQL na gramática. Também foi criado o *visit* para o mesmo para que este seja representado de forma correta. A devolução do *visit* ao `type_spec` é um *SyntaNode* definido por *TypeSyntax*.

O `default_value_part` apesar de ser opcional, corresponde à parte do valor que será inicializada a variável, por isso ser opcional. O `ASSIGN_OP` e `DEFAULT` correspondem a *Tokens* `:=` e `DEFAULT` correspondentes. O `expression` será tratado posteriormente.

```
1  variable_declaration
2      : identifier CONSTANT? type_spec (NOT NULL_)?
3        default_value_part? ';'
4      ;
5
6  default_value_part
7      : (ASSIGN_OP | DEFAULT) expression
8      ;
```

Listagem 4.7: Gramática - Declaração de Variável

```
public override SyntaxNode VisitVariable_declaration
([NotNull] PLSqlParser.Variable_declarationContext context){...}
```

Listagem 4.8: Visitor - Declaração de Variável

Para fazer a representação do *SyntaxNode* da declaração de variáveis na UAST foi utilizada a *class* representada na Listagem a seguir (4.9).

```
VariableDeclarationSyntax(name, initializer , type)
```

Listagem 4.9: UAST *SyntaxNode* para Declaração de Variável

A *class* da Listagem 4.9 recebe:

- **name:** é um identificador e será definido pelo `IdentifierNameSyntax` que provem do *visit* ao `identifier`.
- **initializer:** apenas existirá se a variável for inicializada, ou seja, se a *keyword* `default_value_part` existir. O *visit* ao `default_value_part` retornará uma `EqualsValueClauseSyntax`, que será a parte que entra no lugar do `initializer` na *class*. Caso não exista inicializador, será declarado como `null`.
- **type:** corresponde ao tipo que a variável pertence. O *visit* ao `type_spec` retornará o nodo correspondente para ser recebido no lugar do `type`.

Na figura 4.2 é possível ver a UAST com algumas variáveis de diferentes tipos a ser representada.

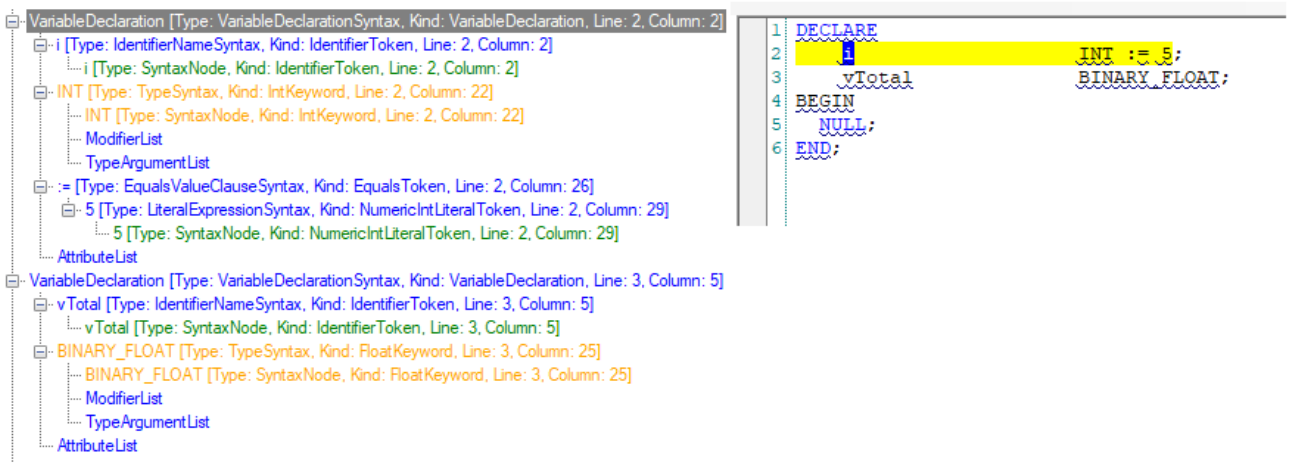


Figura 4.2: UAST - Declaração de Variável

4.3 Exceções e Atribuições

As exceções são utilizadas por diferentes métodos como por exemplo funções e procedimentos e obriga uma diferente forma de tratamento.

Exceções

É diferente a representação para a UAST de uma função ou bloco anónimo sem a *keyword* `EXCEPTION` e com. O facto de ser usada implica a criação de uma `TryStatementSyntax` e a `CatchClauseSyntax`, no fundo é necessário representar o código em forma de *Try-Catch*.

Na listagem 4.10 temos um excerto de código com o uso de diferentes exceções, na listagem 4.11 encontra-se a gramática que faz o *parsing* do mesmo. Foi efetuado um *visitor* ao *exception_handler* (linha 1 de 4.11 e Listagem 4.12).

```
1   EXCEPTION
2       WHEN NO_DATA_FOUND
3       THEN
4           (...)
5       WHEN EPRESIDENT
6       THEN
7           (...)
8       WHEN OTHERS
9       THEN
10          (...)
11  END;
```

Listagem 4.10: PL/SQL - Exceções

Na gramática da parte da exceção conseguimos compreender que a sintaxe/*keyword* `OR exception_name` pode ser opcional, pode ter uma ou muitas repetições devido ao uso do `*`(Asterisco).

O `exception_name` indica qual é a exceção, o seu *visit* passa por diferentes ramos nomeadamente o `identifier` (mencionado anteriormente) também, sendo assim, o retorno do mesmo, será um `IdentifierNameSyntax`.

O `seq_of_statements` é um bloco de *statements* que recebe a exceção executa caso "caia" na mesma.

```
1 exception_handler
2     : WHEN exception_name (OR exception_name)* THEN seq_of_statements
3     ;
```

Listagem 4.11: Gramática - Exceções

```
public override SyntaxNode VisitException_handler
([NotNull] PLSqlParser.Exception_handlerContext context){...}
```

Listagem 4.12: Visitor - Exceções

Para fazer a representação do *SyntaxNode* das exceções na UAST foram utilizadas as *class's* representadas na Listagem a seguir (4.13).

```
TryStatementSyntax(tryToken, block, catches)

CatchClauseSyntax(keyword, declaration, block)
```

Listagem 4.13: UAST *SyntaxNodes* para Exceções

A *class TryStatementSyntax* da Listagem 4.13 recebe:

- **tryToken:** é o *token* que define o inicio do *try*, no caso do PL/SQL será o *token* da *keyword* `BEGIN` de qualquer método.

- **block:** recebe a lista de *statements* que pertence a declaração das variáveis e ao código do programa em si, depois do `BEGIN`.
- **catches:** recebe a `CatchClauseSyntax` que será definida a seguir.

A *class* *CatchClauseSyntax* da Listagem 4.13 recebe:

- **keyword:** *token* que define o inicio do *catch*, será o *token* da *keyword* `EXCEPTION`.
- **declaration:** recebe um `CatchDeclarationSyntax`, esta *class* define o tipo e o identificador da exceção. O identificador da exceção é definido pelo visit a `exception_name`.
- **block:** tudo o que provem do `visit` ao `seq_of_statementes`.

Na figura 4.3 é possível ver a UAST com 2 exceções, `NO_DATA_FOUND` e `OTHERS`.

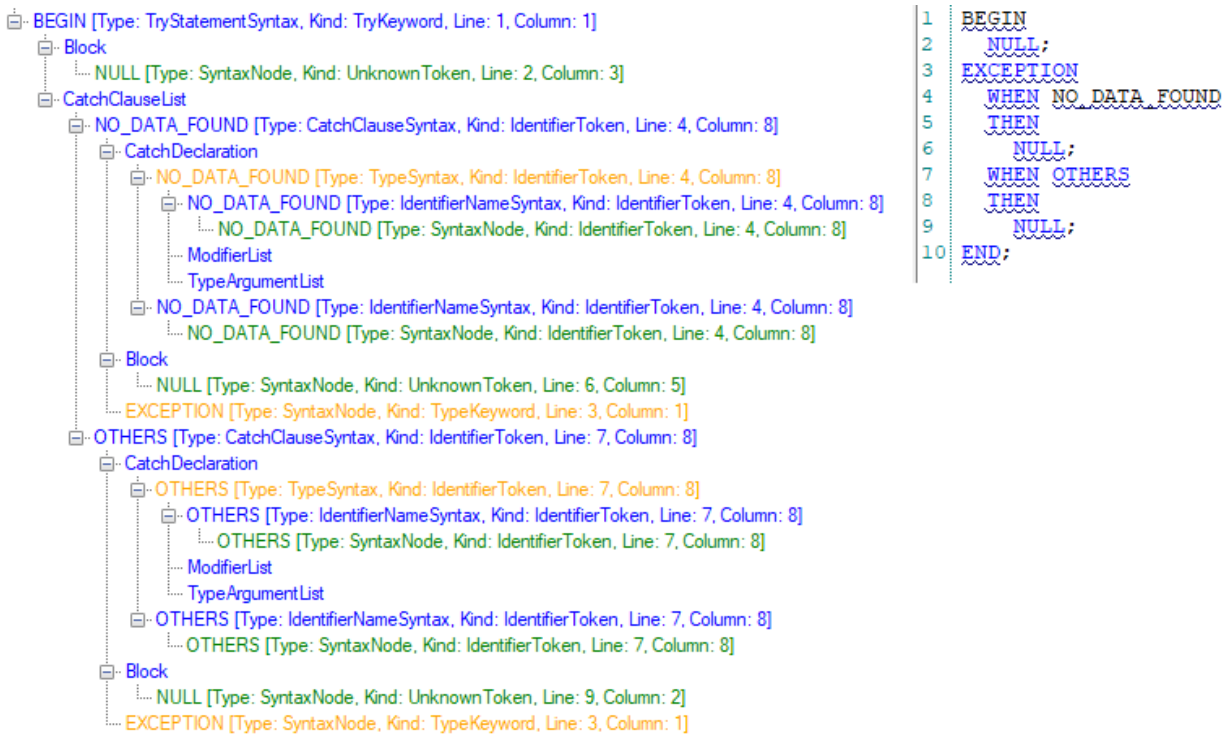


Figura 4.3: UAST - Exceções

Atribuições

Na listagem 4.14 está representado um exemplo de uma atribuição realizada com a linguagem de PL/SQL. Na listagem 4.15 encontra se a gramática que faz o *parsing* do exemplo.

```
1  BEGIN
2      i := 50;
3      (...)
4  END;
```

Listagem 4.14: PL/SQL - Atribuições

Ao analisar a gramática é possível compreender que esta se define essencialmente com 3 componentes. Temos o componente da esquerda, que tanto pode ser um `genenra_element` como um `bind_variable` representa o componente no qual vai ser feita a atribuição, o componente do meio sendo o `ASSIGN_OP` este nó é um no terminal logo é definido por *Token* e representa o *equal* ou `:=`. Por fim, temos o componente da direita definido como `expression` este representa aquilo que vai ser atribuído ao componente da esquerda.

```
1  assignment_statement
2      : (general_element | bind_variable) ASSIGN_OP expression
3      ;
```

Listagem 4.15: Gramática - Atribuições

Na listagem 4.16 esta representada a *head* do *visitor* ao `assignment_statement` da listagem 4.15 (linha 1).

```
public override SyntaxNode VisitAssignment_statement
([NotNull] PLSqlParser.Assignment_statementContext context){...}
```

Listagem 4.16: Visitor - Atribuições

Para fazer a representação do *SyntaxNode* das atribuições na UAST foi utilizada a *class* representadas na Listagem a seguir (4.17).

```
AssignmentExpressionSyntax(left, operatorToken, right)
```

Listagem 4.17: UAST *SyntaxNodes* para Atribuições

A *class AssignmentExpressionSyntax* da Listagem 4.13 recebe:

- **left:** representa-se como sendo uma *expression* e recebe o que provem do *visit* da parte da esquerda da gramática em listagem 4.15
- **operatorToken:** como o nome indica, recebe o *token* do nó terminal da gramática, neste caso o token do `ASSIGN_OP`.
- **right:** representa-se através de uma *expression* definida pelo *visit* ao `expression`, recebe a parte da direita da gramática, em listagem 4.15.

Na figura 4.4 é possível ver a UAST com 2 atribuições.

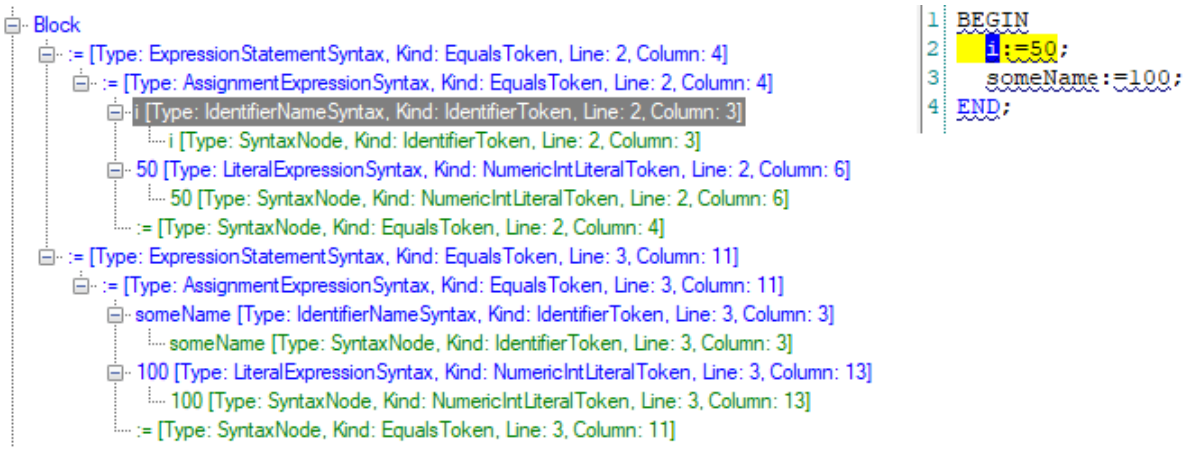


Figura 4.4: UAST - Exceções

4.4 Expressões Unárias e Binárias

Expressões Unárias

Na listagem 4.18 está representado um exemplo de duas atribuições com o uso de expressões unárias realizada com a linguagem de PL/SQL, na linha 2 temos um negativo `-` e na linha 3 um positivo `+`. Na listagem 4.19 encontra se a gramática que faz o *parsing* do exemplo.

```
1  BEGIN
2      i := -50;
3      i := +12;
4      (...)
5  END;
```

Listagem 4.18: PL/SQL - Expressões Unárias

Nas listagens 4.19 e 4.20 estão representadas partes da gramática responsável pelo *parsing* das *unary expressions*. Através de uma analise, podemos compreender que na representação de um `i := -50;` pela gramática, o *visit* feito será na segunda linha da listagem 4.19. O `-` é representado pelo `MINUS_SIGN` e o 50 pelo `unary_expression`. É de notar que o `unary_expression` da linha 2 é o mesmo que da linha 1, significa que é uma recursividade. O 50, é representado então após entrar no `atom` (listagem 4.20), dentro do `atom` faz o *visit* ao `constant` onde será definido o 50 como uma *literal expression* através do uso do *token* já no nó terminal.

```
1 unary_expression
2     : (PLUS_SIGN | MINUS_SIGN) unary_expression
3     (...)
4     |  atom
5     ;
```

Listagem 4.19: Gramática - Expressões Unárias (Parte 1)

```
1 atom
2     : constant
3     | '(' expressions ')'
4     (...)
5     ;
```

Listagem 4.20: Gramática - Expressões Unárias (Parte 2)

Na listagem 4.21 esta representada a *head* do *visitor* ao `unary_expression` da listagem 4.19 (linha 1) e a *head* do *visitor* ao `atom` da listagem 4.20 (linha 1).

```
public override SyntaxNode VisitUnary_expression
([NotNull] PLSqlParser.Unary_expressionContext context){...}

public override SyntaxNode VisitAtom
([NotNull] PLSqlParser.AtomContext context)
```

Listagem 4.21: Visitor - Expressões Unárias

Para fazer a representação do *SyntaxNode* das atribuições na UAST foi utilizada a *class* representadas na Listagem a seguir (4.22).

```
PrefixUnaryExpressionSyntax(kind, operatorToken, operand)
```

Listagem 4.22: UAST *SyntaxNodes* para Expressões Unárias

A *class* *PrefixUnaryExpressionSyntax* da Listagem 4.22 recebe:

- **kind:** este indica o tipo de prefixo que esta a ser criado.
- **operatorToken:** aqui será inserido ou o *token* do `PLUS_SIGN` ou o *token* do `MINUS_SIGN`.
- **operand:** Expressão a qual ser irá aplicar a expressão unária, *visit* do `unary_expression` da linha 2 da listagem 4.19

Na figura 4.5 é possível ver a UAST com 2 atribuições e expressões unárias.

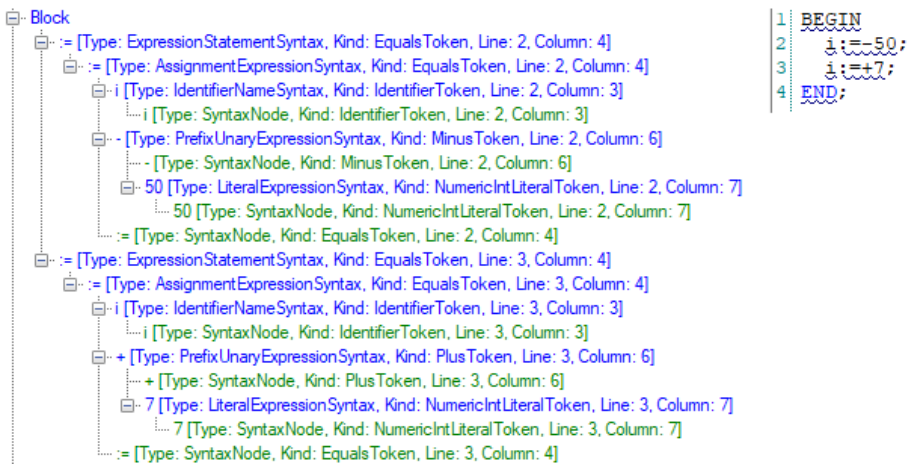


Figura 4.5: UAST - Expressões Unárias

Expressões Binárias

Na listagem 4.23 está representado um exemplo de duas atribuições com o uso de expressões binárias realizada com a linguagem de PL/SQL, na linha 2 temos uma subtração e na linha 3 uma multiplicação e de seguida uma soma. Na listagem 4.24 encontra se parte da gramática que faz o *parsing* do exemplo.

```
1  BEGIN
2      i:=5-50;
3      i:=2*a+1;
4      (...)
5  END;
```

Listagem 4.23: PL/SQL - Expressões Binárias

Nas listagens 4.24 e 4.25 estão representadas partes da gramática responsável pelo *parsing* das *binary expressions*. Na linha 2 da listagem 4.24, o símbolo `model_expression` será o responsável por definir as expressões que irão ser, somadas | subtraídas | divididas | multiplicadas, compreendemos que tudo o que esta a a seguir ao `model_expression` é opcional (de momento não esta a ser tratado).

Os símbolos `left=` e `right=` são apenas para diferenciar os `concatenation`. `PLUS_SIGN`, `MINUS_SIGN`, `ASTERISK`, `SOLIDUS` e `BAR` são *tokens* das expressões binárias, significam `+`, `-`, `*`, `/` e `+` (de concatenação) respetivamente.

```
1  concatenation
2      : model_expression
3          (AT (LOCAL | TIME ZONE concatenation) | interval_expression)?
4          (ON OVERFLOW (TRUNCATE | ERROR))?
5                  #concat_modelexp
6      | left=concatenation (ASTERISK | SOLIDUS) right=concatenation
7                  #concat_mult_div
8      | left=concatenation (PLUS_SIGN | MINUS_SIGN) right=concatenation
9                  #concat_plus_minus
10     | left=concatenation BAR BAR right=concatenation
11                 #concat_concat
12     ;
```

Listagem 4.24: Gramática - Expressões Binárias (Parte 1)

Na listagem 4.25 esta a parte das expressões binárias de lógica, a lógica de funcionamento é equivalente ao da listagem 4.24. Os *tokens* `AND` e `OR` representam-se pela tradução nos símbolos `&&` e `||`. Tanto o `concatenation` da listagem 4.24 como o `logical_expression` da listagem 4.25 são métodos recursivos.

```
logical_expression
    : unary_logical_expression
                #exp_logical_unary
    | left=logical_expression AND right=logical_expression
                #exp_logical_and
    | left=logical_expression OR right=logical_expression
                #exp_logical_or
    ;
```

Listagem 4.25: Gramática - Expressões Binárias (Parte 2)

Na listagem 4.26 estão representados os *visitors* para cada regra do `concatenation` e do `logical_expression`. O `#` nas gramáticas é utilizado para definir um nome único a regra, para assim, o *visit* ser elaborado de uma forma independente.

```
public override SyntaxNode VisitConcat_concat
([NotNull] PLSqlParser.Concat_concatContext context){...}

public override SyntaxNode VisitConcat_plus_minus
([NotNull] PLSqlParser.Concat_plus_minusContext context){...}

public override SyntaxNode VisitConcat_mult_div
([NotNull] PLSqlParser.Concat_mult_divContext context){...}

public override SyntaxNode VisitExp_logical_or
([NotNull] PLSqlParser.Exp_logical_orContext context){...}

public override SyntaxNode VisitExp_logical_and
([NotNull] PLSqlParser.Exp_logical_andContext context){...}
```

Listagem 4.26: Visitor - Expressões Binárias

Para fazer a representação do *SyntaxNode* das atribuições na UAST foi utilizada a *class* representadas na Listagem a seguir (4.27).

```
BinaryExpressionSyntax(kind, left, operatorToken, right)
```

Listagem 4.27: UAST *SyntaxNodes* para Expressões Binárias

A *class PrefixbinaryExpressionSyntax* da Listagem 4.27 recebe:

- **kind:** este indica o tipo de expressão se trata.
- **left:** expressão que esta a esquerda, obtém se pelo *visit* ao `left`.
- **operatorToken:** aqui será inserido o *token*, depende de qual *visit* estiver a ser elaborado. Indica o operador da expressão.
- **right:** expressão que esta a direita, obtém se pelo *visit* ao `right`.

Na figura 4.6 é possível ver a UAST com 2 atribuições e expressões Binárias.



Figura 4.6: UAST - Expressões Binárias

4.5 Declaração de Funções e If's

Declaração de Funções

Na listagem 4.28 temos um exemplo da declaração de uma função PL/SQL, na listagem 4.29 encontra-se a gramática associada, que faz *parser* do exemplo. O `OR REPLACE` é opcional, com no PL/SQL as funções são armazenadas na base de dados, caso seja preciso alterar uma função com o mesmo nome terá que ser usada a sintaxe `OR REPLACE` para prosseguir com a alteração, ao contrario, ocorrerá um erro.

```
1  CREATE OR REPLACE FUNCTION FNC_Name
2      (i IN NUMBER)
3  RETURN NUMBER
4  IS
5      (var decl...)
6  BEGIN
7      (...)
8  EXCEPTION
9      (...)
10 END;
```

Listagem 4.28: PL/SQL - Declaração de Função

Na listagem 4.29 encontra se a gramática da declaração de uma função, o `create_function_body`. Como referido em cima o `OR REPLACE` é opcional na sua declaração. O ponto principal ao que se deve notar aqui é a linha 9, esta é a parte do corpo da

função, o `DECLARE` é opcional e não é necessário estar presente, o `seq_of_declare_specs` é equivalente ao do bloco anónimo já implementado. O `body` é obrigatório executar o seu *visitor* para conseguirmos representar a sua declaração.

A regra `body` (linha 13) é composto por um `begin` com a sequência de *statements*, `seq_of_statements` e a possibilidade de uso de exceções, `(EXCEPTION exception_handler+)?`, estas exceções são elaboradas da mesma maneira que no bloco anonimo, caso sejam utilizadas, realiza se um *Try Catch*

```
1  create_function_body
2      : CREATE (OR REPLACE)? FUNCTION function_name
3          ('(' parameter (',' parameter)* ')')?
4        RETURN type_spec
5          (invoker_rights_clause | parallel_enable_clause |
6              result_cache_clause | DETERMINISTIC)*
7
8        ((PIPELINED? (IS | AS)
9        (DECLARE? seq_of_declare_specs? body | call_spec)) |
10       (PIPELINED | AGGREGATE) USING implementation_type_name) ';'
11     ;
12
13 body
14     : BEGIN seq_of_statements
15         (EXCEPTION exception_handler+)? END label_name?
16     ;
```

Listagem 4.29: Gramática - Declaração de Função

Para a implementação do *parsing* da declaração de funções foram elaborados estes *visitors*, Listagem 4.30.

```
public override SyntaxNode VisitCreate_function_body
([NotNull] PLSqlParser.Create_function_bodyContext context){...}

public override SyntaxNode VisitBody
([NotNull] PLSqlParser.BodyContext context){...}
```

Listagem 4.30: Visitor - Declaração de Função

Para fazer a representação do *SyntaxNode* da função na UAST foi utilizada a *class* representada na Listagem a seguir (4.31).

```
MethodDeclarationSyntax(methodName, parameters, body, returnType)
```

Listagem 4.31: UAST *SyntaxNode* para Declaração de Função

A *class* da Listagem 4.31 recebe:

- **methodName:** é um identificador da função, o *visit* a `function_name` será retornado o *token* e criado um `IdentifierNameSyntax` com o mesmo.

- **parameters:** os parâmetros que entram no método, todos os *visit* elaborados aos `parameter` (listagem 4.29 linha 3) são adicionados numa `ParameterListSyntax` que é inserida no `parameters`.
- **body:** Aqui, entra a lista que provem do *visit* ao `seq_of_declare_specs` (listagem 4.29 linha 9) e `seq_of_statements` (listagem 4.29 linha 14) unida, ambos os *visit* são definidos por uma lista em *SyntaxNode* denominada *BlockSyntax*.
- **returnType:** `TypeSyntax`, é o tipo de *return*, obtido através do *visit* ao `type_spec` (listagem 4.29 linha 4)

Na figura 4.7 é possível ver toda a UAST que representa uma declaração de função.



Figura 4.7: UAST - Declaração de Função

Estruturas If

Na listagem 4.32 esta representado um exemplo de uma estrutura *If* com a parte *Else*, na listagem 4.33, esta representado um exemplo de *If* com a parte *Elsif*. Na listagem 4.34 é demonstrada a gramática que executa o *parsing* dos exemplos.

```
1  IF varNumb = 80
2  THEN
3      (...)
4  ELSE
5      (...)
6  END IF;
```

Listagem 4.32: PL/SQL - Estrutura If

```
1  IF varNumb = 50
2  THEN
3      (...)
4  ELSIF varNumb > 2
5  THEN
6      (...)
7  END IF;
```

Listagem 4.33: PL/SQL - Estrutura Elsif

De seguida, na listagem 4.34, é possível compreender que para ser representado uma estrutura *if* apenas basta a regra da linha 1, sendo que a parte *elsif_part* e *else_part* são opcionais. Para a representação da parte *elsif*, é usada a regra da linha 6 (`elsif_part`) onde entra uma `condition` tal como em `if_statement`. Para a representação da parte *else* é usada a regra da linha 10 (`else_part`).

```
1  if_statement
2      : IF condition THEN seq_of_statements elsif_part*
3        else_part? END IF
4      ;
5
6  elsif_part
7      : ELSIF condition THEN seq_of_statements
8      ;
9
10 else_part
11     : ELSE seq_of_statements
12     ;
```

Listagem 4.34: Gramática - Estruturas If/Else/Elsif

A seguinte listagem 4.35 representa as *heads* dos *visitors* implementados para ser possível a sua representação.

```
public override SyntaxNode VisitIf_statement
([NotNull] PLSqlParser.If_statementContext context){...}

public override SyntaxNode VisitElsif_part
([NotNull] PLSqlParser.Elsif_partContext context){...}
```

Listagem 4.35: Visitor - Estruturas If

Para fazer a representação do *SyntaxNode* das estruturas *if* na UAST foi utilizada a *class* representada na Listagem a seguir (4.36).

```
IfStatementSyntax(keyword, condition, blockTrue, blockFalse)
```

Listagem 4.36: UAST *SyntaxNode* para Estruturas If

A *class* da Listagem 4.36 recebe:

- **keyword:** *SyntaxToken*, aqui entra o que provem do *visit* ao IF e ELSIF da listagem 4.34 linha 2 e 7 respetivamente.
- **condition:** é a condição que define o funcionamento do *if*. Representada por *ExpressionSyntax*.
- **blockTrue:** Sequência de *statements*, tudo o que faz parte do bloco verdadeiro de uma estrutura *if*, representado através de *BlockSyntax*.
- **blockFalse:** Sequência de *statements*, tudo o que faz parte do bloco falso de uma estrutura *if*, representado através de *BlockSyntax*.

Nas figuras4.8 e 4.9 é possível ver toda a UAST que representa uma estrutura *if*, *elsif* e *else*.

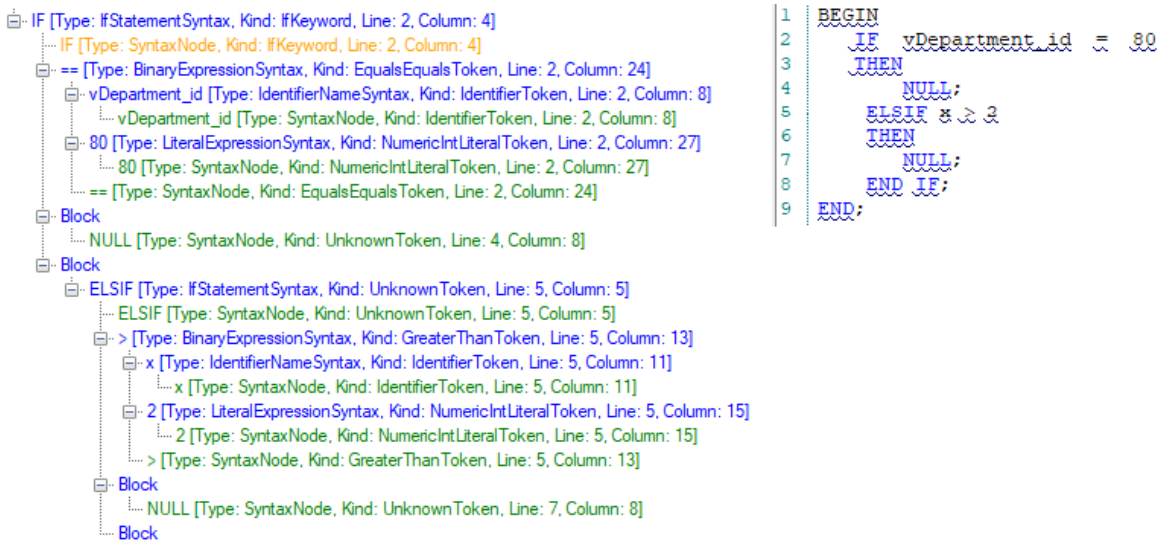


Figura 4.8: UAST - Estrutura *If* com *Elsif*



Figura 4.9: UAST - Estrutura *If* com *Else*

4.6 Invocações e Declarações For

Invocações

Na listagem 4.37 encontra se um exemplo com 3 invocações em PL/SQL, a primeira (linha 2) recebe dois parâmetros como string, a segunda (linha 3) recebe 3 parâmetros dos quais uma variável, uma string e um inteiro, por fim, a terceira (linha 4) invocação apenas recebe uma variável.

```
1  BEGIN
2      a:= b('5','aaa');
3      i:= c(a,'ddd',3);
4      g:= d(i);
5      (...)
6  END;
```

Listagem 4.37: PL/SQL - Invocações

De seguida, a gramática que efetua o *parsing* das invocações, são três as regras as quais será necessário efetuar os *visitors*, o restantes usados já foram implementados à medida que foi necessário o seu uso. As partes opcionais, excluindo o `function_argument`, ainda não foram suportadas (Listagem 4.38).

```
1  general_element_part
2      : (INTRODUCER char_set_name)? id_expression ('.' id_expression)*
3        ('@' link_name)? function_argument?
4      ;
5
6  function_argument
7      : '(' (argument (',' argument)*)? ')' keep_clause?
8      ;
9
10 argument
11     : (identifier '=' '>')? expression
12     ;
```

Listagem 4.38: Gramática - Invocações

A seguinte listagem 4.39 representa as *heads* dos *visitors* implementados para ser possível a sua representação.

```
public override SyntaxNode VisitGeneral_element_part
([NotNull] PLSqlParser.General_element_partContext context){...}

public override SyntaxNode VisitFunction_argument
([NotNull] PLSqlParser.Function_argumentContext context){...}

public override SyntaxNode VisitArgument
([NotNull] PLSqlParser.ArgumentContext context){...}
```

Listagem 4.39: Visitor - Invocações

Foi utilizada a seguinte *class* (listagem 4.40) para representar as Invocações em *SyntaxNode* da UAST.

```
InvocationExpressionSyntax(expression , arguments ,(...)){...}
```

Listagem 4.40: UAST *SyntaxNode* para Invocações

A *class* da Listagem 4.40 recebe:

- **expression:** *ExpressionSyntax*, definido pelo conjunto de `id_expression`'s visitados, linha 2 listagem 4.38.
- **arguments:** *ArgumentListSyntax*, lista de argumentos que provem do *visit* ao `function_argument`, este, executa o *visit* ao `argument` que devolve um *ArgumentSyntax*.

Na figura 4.10 é possível ver parte da UAST que representa três exemplos de invocações.



Figura 4.10: UAST - Invocações

Declarações For

Na listagem a seguir, 4.41, é um exemplo de uso de ciclo *for* que foi implementado no projeto. O `i` é a variável que vai ser incrementada e o `vStart` e `vEnd` definem o *range* do ciclo, tanto pode ser definido por uma variável como através de números, por exemplo `FOR i IN 10..20 LOOP (...)`.

```
1    FOR i IN vStart..vEnd LOOP
2        (...);
3    END LOOP;
```

Listagem 4.41: PL/SQL - Ciclo For

De seguida, a listagem 4.42 representa a gramática que executa o *parsing* do exemplo de cima (4.41). Não foram implementados os reconhecimentos de *label* nem o `WHILE condition` neste projeto. O `cursor_loop_param` representa a parte do `i IN vStart.. vEnd` do exemplo. Existe a *keyword* `REVERSE` que é opcional e também foi implementada, faz o incremento de uma forma decrescente, por exemplo se for entre 10 a 20 o `i` começará no 20 ate ao 10.

```
1  loop_statement
2      : label_declaration? (WHILE condition | FOR cursor_loop_param)?
3        LOOP seq_of_statements END LOOP label_name?
4      ;
5
6  cursor_loop_param
7      : index_name IN REVERSE? lower_bound
8        range_separator='..' upper_bound
9      | (...)
10     ;
```

Listagem 4.42: Gramática - Ciclo For

A seguinte listagem 4.43 representa a *head* do *visitor* implementado para ser possível a sua representação. O `ForVisitorHelper` foi uma função implementada a parte para criar o *SyntaNode* do *for* da forma mais correta, elaborado desta maneira por questões de *NULL's* (para serem evitados).

```
public override SyntaxNode VisitLoop_statement
([NotNull] PLSqlParser.Loop_statementContext context){...}

public SyntaxNode ForVisitorHelper
(PLSqlParser.Cursor_loop_paramContext context,
SyntaxToken forToken, BlockSyntax stmtblock){...}
```

Listagem 4.43: Visitor - Ciclo For

Foi utilizada a seguinte *class* (listagem 4.44) para representar os Ciclos *FOR* em *SyntaxNode* da UAST.

```
ForStatementSyntax(keyword, initializers, condition, incrementors, body)
```

Listagem 4.44: UAST *SyntaxNode* para Ciclos For

A *class* da Listagem 4.44 recebe:

- **keyword:** *SyntaxToken*, *visit* do *token* da *keyword* `for`.
- **initializers:** *StatementListSyntax*, lista de inicializadores, no caso de PL/SQL apenas é permitido um por ciclo *for*.
- **condition:** condição elaborada através de um *BinaryExpressionSyntax*.
- **incrementors:** variável que inicializa o ciclo (`index_name`) é criado um *PostfixUnaryExpressionSyntax* para fazer a representação da incrementação [++], e caso haja `REVERSE`, será um [- -].
- **body:** *BlockSyntax*, lista de *statements* que serão executadas.

Na figura 4.11 é possível ver parte da UAST que representa o exemplo de uso de ciclo *for*.



Figura 4.11: UAST - Ciclo For

4.7 Complementos

Para alem dos *visitors* implementados e descritos acima, houve necessidade de criação de *visitors* complementares para o correto funcionamento do *parsing*, por exemplo, os *visitors* `seq_of_statements` e `statements`, representam as *statements* e são agrupados num *BlockSyntax* no *visitor* `seq_of_statements`.

Também, foi necessário fazer correções no ficheiro da gramática, pois não era possível representação de variáveis em minúsculo o que era um erro enorme para fazer *parsing* numa linguagem a qual permite o *lower case*.

Exemplo de erro com o *parsing* antes de ser alterada (figura 4.12):

```
VARIABLE gMedia NUMBER
DECLARE
  vNumber        NUMBER(11,2) := 1200.50;
  vChar           CHAR(100) := 'Char Declare';
VFIRSTNAME   EMPLOYEES.FIRST_NAME%TYPE := 'Bob';
VLASTNAME    employees.last_name%type;
VSALARY      employees.salary%type;
VEMPLOYEESID employees.employee_id%type := 121;
```

Figura 4.12: Erro de *LowerCase* na gramática

Então, foi alterado o fragmento do ficheiro `PlSqlLexer.g4`

`line:2366 | fragment SIMPLE_LETTER : [A-Z];`

para

`line:2366 | fragment SIMPLE_LETTER : [A-Z]|[a-z];`

Assim tornou se possível a utilização de *lowercase* (figura 4.13).

```
CREATE OR REPLACE PROCEDURE PRC_UPDATE_SALARY2
  (ppercentual IN NUMBER)
AS
  vLimit CONSTANT INTEGER(2) := 30;
  TYPE employee_id_table_type IS TABLE OF employees.employee_id%TYPE
  INDEX BY BINARY_INTEGER;  -- Type Associative Array
  employee_id_table  employee_id_table_type;
  CURSOR employees_cursor IS
    SELECT employee_id
    FROM employees;
BEGIN
```

Figura 4.13: Erro de *LowerCase Fixed* na gramática

Foram elaboradas alterações na gramática para reconhecimento direto de operadores lógicos. Apesar da linguagem suportar, era complicado fazer a sua leitura como *visitor*, então procedeu se a seguinte alteração.

Antes da alteração, é possível ver que o *Parser* apesar de conter os símbolos terminais no *Lexer* definidos, não os estava a utilizar (Figura 4.14). Então precedeu se a alteração juntamente com adição de 2 *tokens*, `GREATER_THAN_EQUALS_OP` (linha 2328 da figura 4.15) e `LESS_THAN_EQUALS_OP` (linha 2330 da figura 4.15)

```
3774  relational_operator
3775      : '='
3776      | (NOT_EQUAL_OP | '<' '>' | '!' '=' | '^' '=')
3777      | ('<' | '>') '='?
3778      ;
```

```
2324  CARRET_OPERATOR_PART:       '^';
2325  TILDE_OPERATOR_PART:        '~';
2326  EXCLAMATION_OPERATOR_PART: '!';
2327  GREATER_THAN_OP:            '>';
2328  LESS_THAN_OP:               '<';
2329  COLON:                      ':';
2330  SEMICOLON:                  ';';
```

Figura 4.14: *Parser* e *Lexer* antes da correção

```
3774  relational_operator
3775      : EQUALS_OP
3776      | NOT_EQUAL_OP
3777      | (LESS_THAN_OP | GREATER_THAN_OP)
3778      | (LESS_THAN_EQUALS_OP|GREATER_THAN_EQUALS_OP)
3779      ;
```

```
2324  CARRET_OPERATOR_PART:       '^';
2325  TILDE_OPERATOR_PART:        '~';
2326  EXCLAMATION_OPERATOR_PART: '!';
2327  GREATER_THAN_OP:            '>';
2328  GREATER_THAN_EQUALS_OP:     '>=';
2329  LESS_THAN_OP:               '<';
2330  LESS_THAN_EQUALS_OP:        '<=';
2331  COLON:                      ':';
2332  SEMICOLON:                  ';';
```

Figura 4.15: *Parser* e *Lexer* depois da correção

5. Análise dos resultados

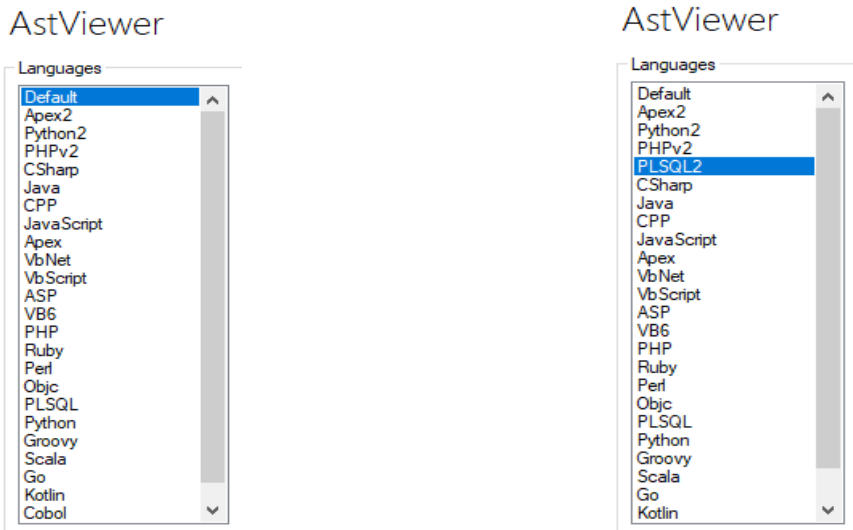
Neste capitulo, encontra se o resumo do progresso global do estágio e estagiário, incluindo uma revisão geral do que foi abordado e desenvolvido e uma breve nota do supervisor.

5.1 Apresentação da solução desenvolvida

Esta secção pode ser sub dividida em 4 tópicos para demonstrar o que foi desenvolvido durante o período do estagio.

5.1.1 Ferramentas

AstTreeViewer é uma ferramenta da Checkmarx que permite a visualização da árvore sintática de um excerto de código de uma certa linguagem. Foi implementada a opção da representação da AST da linguagem PL/SQL reescrita nessa mesma *tool* (figura 5.1).



(a) Linguagens *AstTreeViewer* - Antes (b) Linguagens *AstTreeViewer* - Depois

Figura 5.1: Linguagens da *Tool AstTreeViewer*

Também, foi implementado na solução a possibilidade de uso de uma *flag* e a *flag* em si, para poder intercalar entre o uso do *parsing* antigo e o *parsing* novo.

5.1.2 PL/SQL e gramática

A nível de PL/SQL foram produzidos 106 *samples* de código compilável para a utilização em testes à solução nova. Os 106 *samples* estão disponíveis e podem ser consultados no GitHub através da hiperligação.
GitHub Samples - https://github.com/AlekxxxP1K4/PL-SQL_Samplers

Foram elaboradas alterações na gramática já demonstradas no capítulo 4, tópico 4.7.

5.1.3 *Visitors* e DOM

Toda a implementação foi com o objetivo de possibilitar a representação da linguagem em UAST para ser processada para a DOM onde irão ser corridas *queries*. Ao todo foram implementados 42 *visitors* e já se encontram na solução.

Na figura 5.2, consta um exemplo onde se encontra a representação da DOM depois das implementações de todos os *visitors*. Na tabela 5.1 estão identificadas as correspondências de código entre as figuras 5.2a e 5.2b juntamente com a sua descrição.

```
DECLARE
    i                   INT := 5;
    vTotal              BINARY_FLOAT;
BEGIN
    i:=50;
    someName:=100;
    i:=5-50;
    i:=2*a-1;

    a:= FNC_nome(5);

    i:=-50;
    FOR i IN vStart..vEnd  LOOP
        NULL;
    END LOOP;

EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        NULL;
END;

CREATE OR REPLACE FUNCTION FNC_nome
   (v IN INTEGER)
   RETURN NUMBER
IS
BEGIN
    IF  v  =  50
    THEN
        NULL;
    ELSE
        NULL;
    END IF;
    RETURN v;
END;
```

(a) *Source* PL/SQL *code*

```
namespace DefaultNamespace
{
    class DefaultClass
    {
        object CxAnony3c052a07()
        {
            INT i = 5;
            BINARY_FLOAT vTotal;
            try
            {
                {
                    i = 50;
                    someName = 100;
                    i = (5 - 50);
                    i = ((2 * a) - 1);
                    a = FNC_nome(5);
                    i = !50;
                    for (INTEGER i = vStart; (i <= vEnd); i++)
                    {
                    }
                }
            }
            catch (NO_DATA_FOUND)
            {
            }
            finally
            {
            }
        }

        static public NUMBER FNC_nome(INTEGER v)
        {
            if ((v == 50))
            {
            }

            return v;
        }
    }
}
```

(b) Representação DOM

Figura 5.2: Representação da DOM de um snippet de código PL/SQL

Figura 5.2a	Figura 5.2b	Descrição
2	7	Declaração de Variável com inicialização
3	8	Declaração de Variável
5	12	Atribuição
6	13	Atribuição
7	14	Binária Subtração
8	15	Binária Multiplicação e Subtração
10	16	Invocação de Função
12	17	Unária Negativo
13	18	Ciclo *For*
17	23	Exceção
23	31	Declaração de Função
28	33	Estrutura *If*
34	37	Retorno

Tabela 5.1: Tabela de correspondências de linhas da figura 5.2

5.1.4 Testes unitários

Foram desenvolvidos, ao longo da implementação, testes unitários capazes de testar a parte da implementação elaborada. Estes indicam se tudo está a ser representado da forma esperada. Durante a implementação, foram desenvolvidos 31 testes unitários, dos quais todos são positivos (ver figura 5.3).

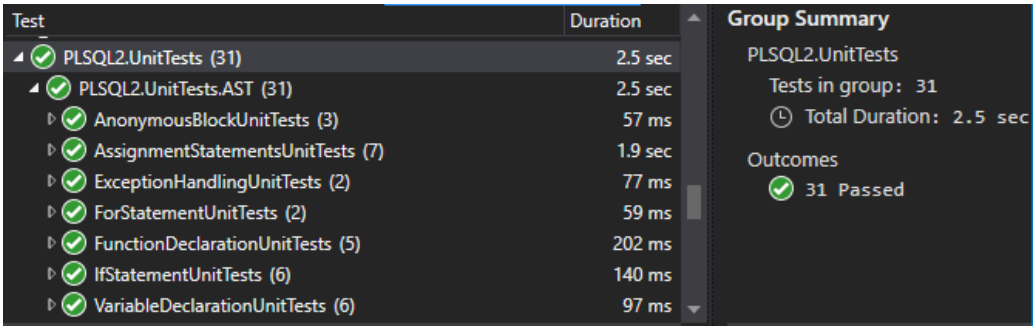


Figura 5.3: Testes Unitários

5.2 Avaliação do nível de maturidade e cumprimento dos objetivos

Após a apresentação da solução desenvolvida (capitulo 4 e tópico 5.1), foi elaborada um análise de comparação com os objetivos e requisitos definidos para este estágio. A análise foi bastante positiva e houve o total cumprimento dos objetivos. Durante a implementação houve sempre a necessidade de investigação e auto motivação para o cumprimento dos objetivos colocados.

5.3 Principais dificuldades encontradas

Durante a implementação, a principal dificuldade encontrada foi o primeiro impacto com a solução. A compreensão inicial do código e da estrutura da solução foi um desafio bastante valioso. Muitas das barreiras durante o estágio foram ultrapassadas com pequenas pesquisas sobre o assunto.

5.4 Parecer do Supervisor

Apresenta-se de seguida o parecer do supervisor de estágio na Checkmarx, Eng. Xavier Pinho:

"O desempenho do Oleksandr nestes três meses foi exemplar, demonstrando, quotidianamente, interesse, iniciativa, disciplina, responsabilidade e cooperação.

Temas que inicialmente desconhecia (e.g. PL/SQL, ANTLR, *Visitor pattern*, etc.) prontamente os dominou suficientemente para as intenções deste projecto.

Pese embora o estatuto de estudante, correspondeu sempre ao padrão do colaborador típico, o que implica, para além da actividade de programação, reportar diariamente ao seu supervisor e quinzenalmente a uma equipa mais alargada, documentar as funcionalidades implementadas, e colaborar com as equipas de QA para garantir que as funcionalidades implementadas estão completas e/ou não têm comportamentos inesperados. Também o código contribuído não foi poupado das exigências típicas, e os *code-reviews*, até quando exaustivos, decorreram fluidamente, um marco de disciplina, responsabilidade e cooperação.

Enquanto seu supervisor, sinto que o Oleksandr prosperou."

6. Conclusão final

A realização deste relatório permite verbalizar o caminho percorrido durante o tempo do estágio e permite descrever o aproveitamento e evolução dos conhecimentos adquiridos no decorrer do curso de Engenharia de Sistemas Informáticos. Neste relatório foram abordados quatro tópicos principais, a introdução na empresa, a análise e definição de objetivos, a implementação dos mesmos e a realização dos testes sobre a solução.

6.1 Principais conclusões do estágio

Efetivamente, este estágio permitiu a abordagem de matérias estudadas durante o período de aprendizagem na licenciatura, nomeadamente a cadeira de Processamento de Linguagens que se encaixa bastante neste estágio. A aprendizagem inicial de PL/SQL foi uma mais-valia, pois aumentou o meu conhecimento em programação de linguagens a nível de base de dados. O estudo de ANTLR4 ajudou imenso a fixar e evoluir os conhecimentos em áreas de gramática e compreender as ferramentas que o utilizam, por exemplo compiladores. O desenvolvimento da solução em C# permitiu praticar mais sobre esta linguagem de programação. Não foi possível, nem estava previsto, a implementação de todas as *features* de PL/SQL na nova Abstract Syntax Tree (Árvore de Sintaxe Abstrata) (AST) mas todas as funcionalidades previstas para o tempo estimado foram elaboradas com sucesso. Durante o decorrer do estágio foi possível evoluir devido a investigações que eram exigidas pelo próprio objetivo, assim contribuindo na minha evolução e no desenvolvimento de novas estratégias, como por exemplo as de observação, tornando me um profissional melhor.

O estágio ofereceu-me uma experiência inicial, mas muito importante, no trabalho em equipa, as *daily meetings* contribuíam imenso para um bom funcionamento do dia e exigiam uma organização justa de tarefas diárias. Apesar de toda a ajuda por parte da organização acolhedora, que era oferecida, fui sempre incentivado a abrir horizontes *"by my own"* o que foi uma mais-valia e um dos pontos mais importantes na minha evolução.

Por fim, elaborando uma análise ao projeto desenvolvido, é visível a dedicação e o esforço realizado da minha parte. O objetivo, na totalidade, ainda está longe de concluído, mas o objetivo para o estágio foi considerado como cumprido, pois foram concluídos todos os requisitos definidos, foram superados os obstáculos esperados e houve um desenvolvimento de competências úteis para mim como futuro profissional.

Bibliografia

Checkmarx. 2013. *Software security for devops and beyond.* https://www.checkmarx.com/about-checkmarx/.

Checkmarx. 2016. *A quebrar barreiras com devsecops.* https://www.checkmarx.com/internship-summer-program/.

Checkmarx. 2021. *Static application security testing (CxSAST).* https://www.checkmarx.com/products/static-application-security-testing/.

Oracle. 2008. *Pl/sql inherits database robustness, security, and portability.* https://www.oracle.com/pt/database/technologies/appdev/plsql.html.

OWASP. 2017. *Owasp top ten.* https://owasp.org/www-project-top-ten/.

Parr, Terence. 2014. *What is antlr?* https://www.antlr.org/.

Sommerville, Ian. 2015. *Software engineering* 10th edition. Pearson Education Limited.

Techopedia. 2003. *What does software security mean?* https://www.techopedia.com/definition/4536/security-software.