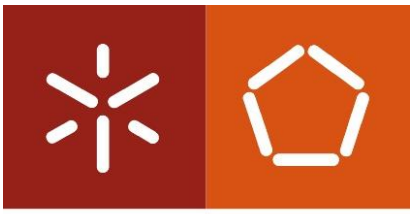

Universidade do Minho

Mestrado em Engenharia Informática

Engenharia Gramatical

Analisador de Código Fonte

Grupo 3

Duarte Parente (PG53791) Gonçalo Pereira (PG53834)

José Moreira (PG53963)

Ano Letivo 2023/2024

Índice

1 Introdução **3**

2 Linguagem Desenvolvida **4**

2.1 Componentes Implementados 4

2.1.1 Tipos de dados e Instruções de Atribuição, Leitura e Escrita 4

2.1.2 Instruções de Seleção 5

2.1.3 Instruções de Repetição 5

2.1.4 Declaração de Funções 6

3 Interpretador **7**

3.1 Estratégia de Implementação 7

4 Análise de Resultados **8**

5 Conclusão **11**

Capítulo 1

Introdução

O presente relatório visa apresentar o segundo trabalho prático desenvolvido no âmbito da Unidade Curricular de **Engenharia Gramatical**. Neste projeto, retomamos o tema das ferramentas avançadas de análise de código, que foram objeto de análise no primeiro trabalho prático. Como já referido, estas ferramentas desempenham um papel fundamental na inspeção e aperfeiçoamento de código fonte. Para além de permitirem, por exemplo, embelezar o código, têm também a capacidade de identificar potenciais vulnerabilidades e sugerir melhores práticas de codificação, sem comprometer o significado do programa

Desta vez, o objetivo do projeto é desenvolver um analisador de código para uma linguagem desenvolvida pelo grupo numa fase inicial do semestre. Esta linguagem, que será apresentada de forma detalhada no capítulo seguinte, é então interpretada pelo analisador, sendo por fim extraídas as estatísticas mais relevantes, em concordância com os requisitos propostos no enunciado, e apresentadas num ficheiro de HTML.

Este documento descreve todo o processo de desenvolvimento do analisador de código, contando também com exemplos de utilização do analisador, demonstrando sua utilidade e eficácia na deteção de potenciais problemas em programas escritos na linguagem de programação desenhada
.

Capítulo 2

Linguagem Desenvolvida

Relativamente à Linguagem de Programação Imperativa desenvolvida no início do semestre, procurou-se a aplicação de uma sintaxe que permitisse conciliar a simplicidade de escrita e leitura, com os requisitos propostos para o desenvolvimento da mesma. Dessa forma, a linguagem `C` assumiu o papel de principal fonte de inspiração, com algumas noções baseadas em outras linguagens imperativas, especialmente o `Python`.

2.1 Componentes Implementados

Os requisitos propostos para a linguagem a desenvolver passavam pela permissão da declaração de variáveis atómicas e estruturadas, instruções de atribuição, leitura e escrita, instruções de seleção (condicionais) e instruções de repetição, com pelo menos três variantes de ciclo.

2.1.1 Tipos de dados e Instruções de Atribuição, Leitura e Escrita

- **Variáveis Atómicas**: `Int`, `Float`, `String`, `Boolean`
- **Variáveis Estruturadas**: Dicionários, Listas, Tuplos, Sets

```
// Exemplos de Declarações

int a = 30;
float b = 3.5;
string c = "2das";
boolean d = true;
tuple e = (1,2);
dict f = {'lionel': 'messi'};
set g = {'abs', 'fgl'};
list h = a = [1, 2, 3];
```

A atribuição de valores a variáveis poderá também ser efetuada de inúmeras possibilidades.

```
int a = 30 + 5;                 //  Operações Binárias
boolean d = a > 90;

a++;                            // Operações Unárias

string y = concat(a, b, c);     // Resultados de Chamadas a Funções
```

Relativamente ao acesso a variáveis declaradas, este é efetuado de uma forma muito semelhante ao `Python`, devido à sua naturalidade. Em concreto, para o acesso a variáveis atómicas apenas será necessária a invocação do nome da variável. Em relação às variáveis estruturadas, apenas as listas e os tuplos suportam indexação. Para além disso, o conteúdo relativo aos dicionários poderá ser obtido através da chave associada ao valor que se pretende aceder.

2.1.2 Instruções de Seleção

As estruturas condicionais implementadas seguem também o formato natural de uma seleção baseada numa qualquer condição. Para além disso, é ainda conferida a possibilidade de estruturas condicionais encadeadas e também aninhadas.

```
// Exemplo de Utilização

int b = 0;
if (a < 50){
    b = 10;
    if (a <= 10){
         a += 8;
    }
}
else if (a > 9){
    b = 50;
    print(b, d);
}
else{
    b = 9;
}
```

2.1.3 Instruções de Repetição

De forma a corresponder positivamente aos requisitos propostos, foram implementados três tipos de ciclos: `for`, `while` e `do_while`.

```
// Exemplo de Utilização

for (int c = 10; c < 20; c = c + a;){
    if (c == 40){
        tuple a = (1, 2, 3);
        print(a);
    }
}

while (a < 10){
    b += 5;
}

do {
    a = 10;
    b = 5;
    y += a;
} while (c < 35)
```

2.1.4 Declaração de Funções

Para além da possibilidade de escrita de *Scripted Code*, é ainda possível a declaração de funções, através da `keyword` "function". Para o efeito deverá de ser indicado o tipo de retorno da função, ou `void` em caso de inexistência. As funções poderão também receber um número ilimitado de parâmetros, devendo também ser indicados os seus tipos.

```
// Exemplo de Declaração

int function list_sum (list a) {
    int inc = 0;
    int counter = 0;
    while (inc < 10) {
        counter = counter + a[inc];
        inc++;
    }

    return counter;
}

void function last (list a, int b) {
    int list result;
    result = cons(b, a);
    print('nice!');
}
```

Capítulo 3

Interpretador

O interpretador desenvolvido teve por base a utilização do módulo `Visitors` para a geração de processadores de linguagens `Lark`. De forma a respeitar os requisitos propostos para a análise do código relativo à linguagem de programação, primeiramente foi necessário pensar nas estruturas de dados apropriadas para a gestão e acesso de toda a informação necessária.

3.1 Estratégia de Implementação

No relatório final a apresentar sobre a análise efetuada, um dos componentes de estudo passaria pela sinalização e respetiva contagem de alguns parâmetros:

- Lista de todas as variáveis do programa indicando os casos de: redeclaração ou não-declaração; variáveis usadas mas não inicializadas; variáveis declaradas e nunca mencionadas.
- Total de variáveis declaradas por cada tipo de dados usado;
- Total de instruções que formam o corpo do programa, indicando o número de instruções de cada tipo (atribuições, leitura e escrita, condicionais e cíclicas);
- Total de situações em que estruturas de controlo surgem aninhadas em outras estruturas de controlo do mesmo ou de tipos diferentes.
- Lista de situações em que existam `ifs` aninhados que possam ser substituídos por um só `if`.

Para a manipulação dos dados relativos a métricas quantitativas a solução passou pela implementação de um contador global, com o único objetivo de controlar os casos relativos a estes parâmetros. No sentido de enriquecer o relatório final, apresentando um relatório com um nível de detalhe mais apurado foi também adicionado um contador relativo a cada função. Para a deteção destes casos, e dos restantes sinalizados foi também necessária a elaboração de uma estrutura que abstraísse o conceito de uma tabela de variáveis, de forma a manter as suas informações atualizadas. Esta estrutura seria utilizada num contexto de uma **stack** de estados do programa, onde a raiz seria o estado global, relativo a todo o código declarado fora de funções. O `push` de um novo estado, acompanhado da respetiva tabela de variáveis, é atividado a partir da declaração de uma função, estrutura condicional ou estrutura cíclica.

Capítulo 4

Análise de Resultados

No presente capítulo são apresentados os resultados obtidos a partir da aplicação do analisador de código desenvolvido para a Linguagem de Programação Imperativa (LPI). A título de exemplo, será apresentado o resultado obtido para a seguinte porção de código da linguagem:

```
int a = 30;
if (a < 50){
    int b = 10;
    int c = 30;
    print(b, c);
}
else {
    int b = 50;
    string d = 10;
    print(b, d);
}

int j = b + d;

int function sum (list a, list b){
    int ff = 10;
    while (ff < 30){
        if (ff == 'yesyes'){
            a = [1, 2, 'ye'];
        }
    }

    return b > 14;
}
```

Começando pelo início da página HTML resultante, o primeiro item a ser apresentado é a secção das funções. Como se observa na imagem seguinte, são exibidas 3 colunas: a primeira contém contadores para os diferentes elementos da função em questão (declarações, ciclos, condicionais e atribuições), a segunda mostra os *logs* das variáveis da função (variáveis declaradas e não declaradas, variáveis que não são utilizadas, etc) e a terceira e última lista os diferentes erros encontrados.

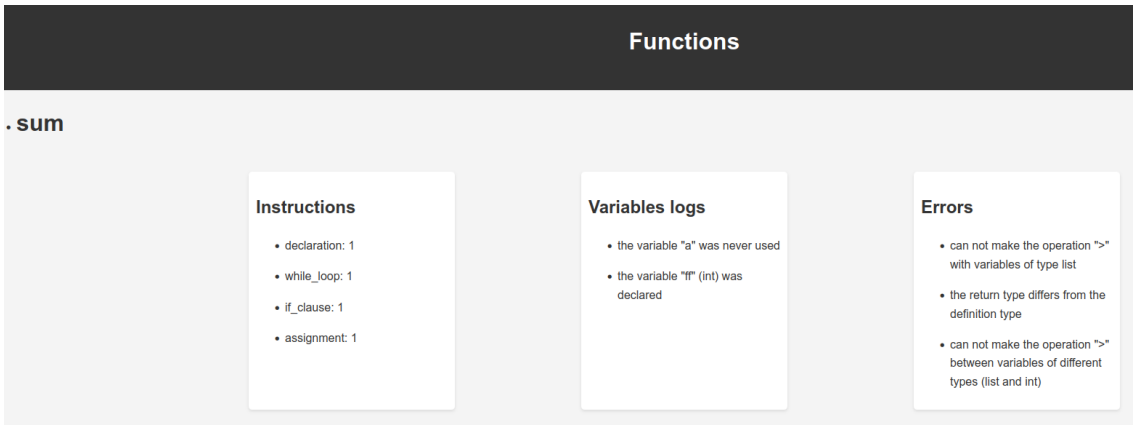

Figura 1: Output - Funções

O segundo item a ser apresentado é o código *script*, ou seja, o código lcoalizado fora das funções. Tal como no item anterior, as colunas apresentadas são a do contador de elementos do código, os *logs* das variáveis e a lista de erros.

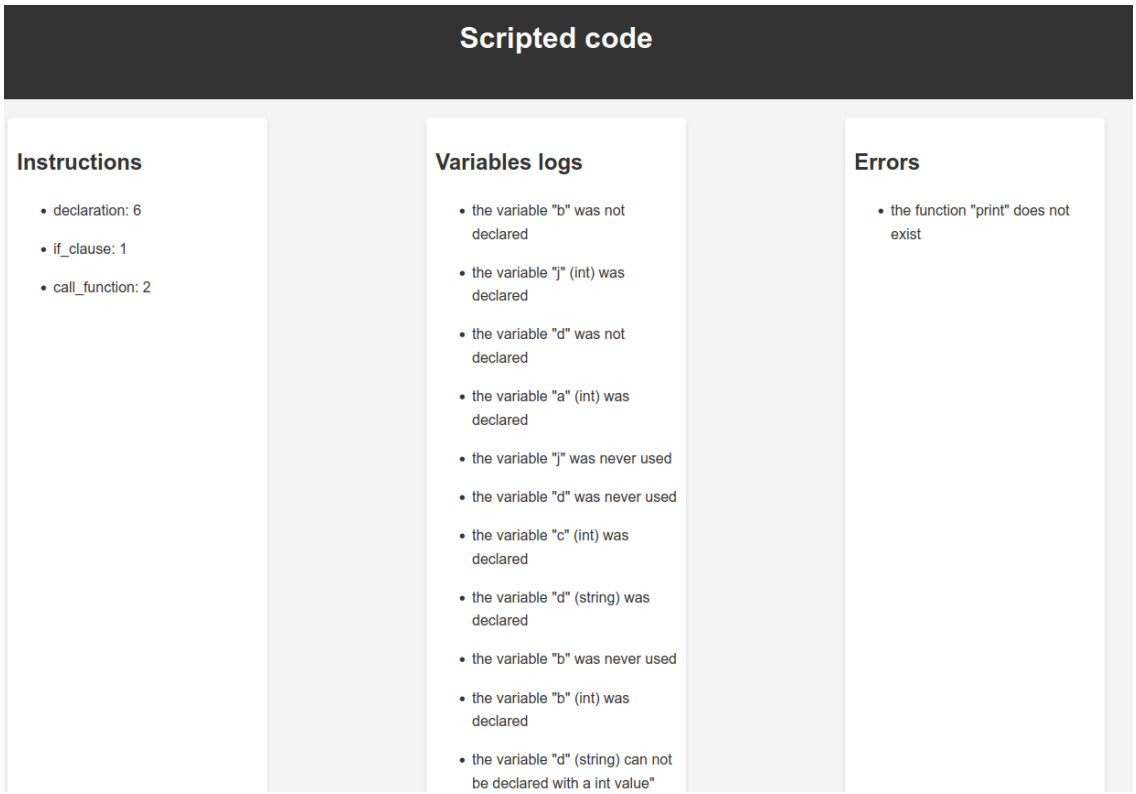

Figura 2: Output - Código *script*

Por fim, é apresentada a secção das estatísticas globais. Nesta secção são exibidas quatro

colunas de informação. A primeira coluna mostra um contador dos tipos de variáveis presentes no código. Na segunda coluna, encontramos a lista de funções de código, incluindo não só os seus nomes, mas também os tipos de retorno e os tipos dos argumentos. A terceira coluna exibe contadores para os diferentes tipos de instruções, como declarações, atribuições, operações, ciclos, chamadas de funções e estruturas condicionais. Por fim, a quarta coluna apresenta as estruturas aninhadas, alertando o utilizador sobre o número de condicionais "if"aninhados que podem ser substituídos por um, de forma a simplificar o código.

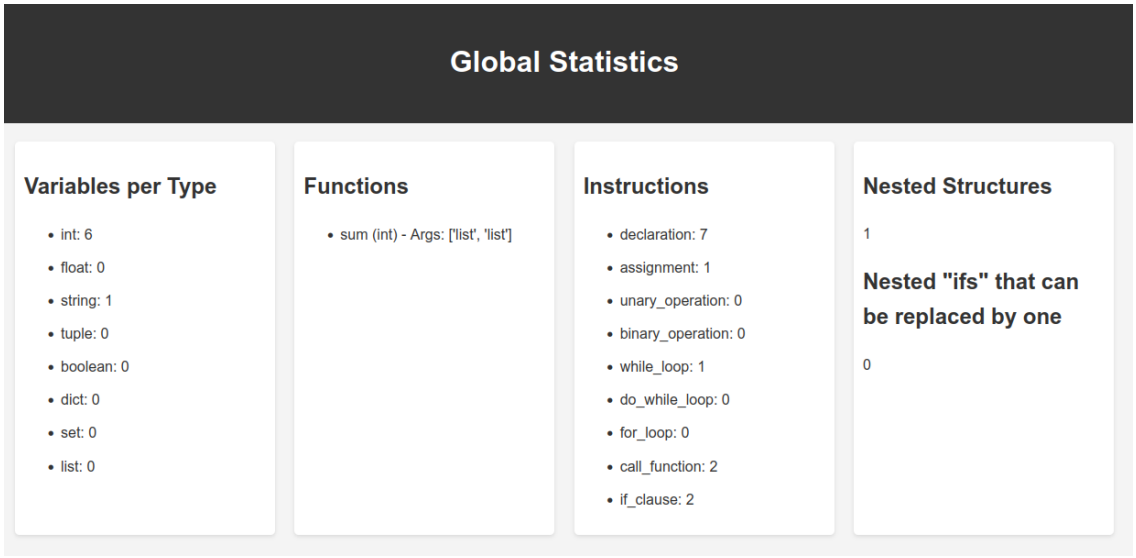

Figura 3: Output - Estatísticas globais

Capítulo 5

Conclusão

O objetivo principal deste analisador foi fornecer uma ferramenta eficaz para a melhoria de programas escritos na linguagem de programação desenvolvida pelo grupo, visando aumentar a qualidade e segurança do código produzido.

Através de exemplos representativos, foi possível demonstrar a capacidade do analisador em identificar violações de boas práticas, potenciais vulnerabilidades durante a execução e oferecer sugestões para aprimoramento do código. Os resultados obtidos foram promissores, evidenciando a utilidade e eficácia do analisador de código desenvolvido. A análise das estatísticas globais, em particular, destaca-se pela sua capacidade de oferecer observações relevantes sobre a estrutura e qualidade do código, além de auxiliar o programador na identificação de possíveis áreas de melhoria.

Durante o desenvolvimento do analisador de código, o grupo enfrentou algumas dificuldades, especialmente ao produzir o interpretador. Foi necessário realizar alguns ajustes na gramática que já havia sido elaborada, um processo desafiador que exigiu atenção detalhada e revisões cuidadosas. No entanto, apesar disto, o grupo se mostra satisfeito com o resultado final alcançado. O facto de surgirem estas dificuldades também demonstra a importância da flexibilidade e da capacidade de adaptação ao lidar com problemas inesperados durante o desenvolvimento de software.

Relativamente ao desenvolvimento futuro, o grupo espera implementar mais estruturas, como por exemplo ciclos *for each* e condicionais de uma só linha. É também objetivo enriquecer a deteção de erros, de forma a que a sua análise cubra mais casos e forneça informações ainda mais detalhadas e concretas, conseguindo assim melhorar a experiência de programação do utilizador, otimizando a gestão de tempo e a eficiência do código produzido.