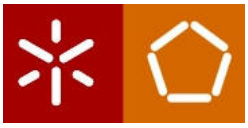

Universidade do Minho
Departamento de Informática

Perfil de Engenharia de Linguagens
Engenharia Gramatical

Trabalho Prático 2

Grupo 5

Gonçalo Costa - PG55944

José Correia - PG55967

Rodrigo Casal Novo - PG56006

Índice

1. Introdução

No âmbito do TPC2, foi desenhada uma linguagem de programação imperativa, cuja gramática foi esboçada com recurso ao módulo *Lark*, em *Python*. Dando continuidade a esse trabalho, o presente projeto teve como objetivo o desenvolvimento de uma ferramenta de análise para essa linguagem, permitindo extrair estatísticas relevantes sobre a estrutura e utilização dos elementos da linguagem.

O analisador desenvolvido tem como principais funcionalidades a deteção de diversas situações de erro no código fonte, bem como a extração de estatísticas relevantes sobre a sua estrutura. Toda a informação é apresentada num relatório em formato *HTML* para serem posteriormente observadas.

2. Gramática da Linguagem

Nesta secção será apresentada a nossa gramática e as regras sintáticas da linguagem.

Decidimos criar uma linguagem parecida com C, mas adicionamos algumas funcionalidades que achamos facilitar o seu uso.

2.1. Regras sintáticas

1. **start**: Define o ponto de entrada do programa. Este é composto por definições de funções (**function_def**) sendo obrigatória que a última seja chamada `main` (**main_func**);
2. **function_def**: Conjunto opcional de funções auxiliares. Cada função é composta por: nome, parâmetros, instruções e valor de retorno (e seu tipo);
3. **main_func**: Função a ser executada quando o programa é chamado. É constituída por um conjunto de comandos (**command**) e e por um valor de retorno (e seu tipo);
4. **command**: Representa uma (**variable_declaration**), alocação(**allocation**), chamada de função (**function_call**), função de escrita (**print_command**), condição (**conditional**) ou ciclo (**loop**);
5. **expr**: representa todos as expressões que correspondem a valores. Isto inclui valores simples como números ou strings, mas também operações aritméticas e lógicas, chamadas de funções (**function_call**) ou arrays (**array_expr**) e acessos aos mesmos (**array_access**);
6. **conditional**: nesta gramática implementamos dois tipos de condicionais:

   6.1. **check**: condicional tradicional, pode conter, caso a primeira verificação não seja válida outras verificações (**also**) ou comandos para quando não é válida nenhuma verificação(**otherwise**);

   6.2. **match**: condicional por comparação com valores previstos (**option**), este condicional também pode conter um conjunto de comandos a ser executados aquando nenhuma opção é escolhida(**standard**);
7. **loop**: nesta gramática implementamos três tipos de condicionais:

   7.1. **for_loop**: ciclo "for" tradicional do C;

   7.2. **loop_loop**: ciclo que só termina quando uma condição é verificada;

   7.3. **foreach_loop**: ciclo especificado para listas, executa o bloco de comandos para cada elemento da lista.

3. Interpretadores léxicos

Para analisar os programas desenvolvidos nesta linguagem e, ao mesmo tempo, modular o processo de análise, optámos por definir dois interpretadores léxicos independentes. Recorrendo ao *Interpreter* do módulo *Lark*: **InterpreterReservedWords** e **InterpreterCondicoesECiclos.** Esta separação permitiu isolar a lógica associada a variáveis, da lógica relacionada a instruções.

3.1. InterpreterReservedWords

Este interpretador visa agrupar a informação importante sobre todas as variáveis do programa, para assim ser possível:
- Listar todas as variáveis do programa.
- Calcular o total de varáveis declaradas por cada tipo de dados.
- Providenciar mensagens de aviso e erro sobre as mesmas.

Para isso, decidimos inicializa-la com os seguintes parâmetros:
- **varDec**: dicionário que guarda, para cada variável, a função onde foi declarada, o nome da variável, o tipo, o número de ocorrências e se foi redeclarada.
- **varReDec**: dicionário que guarda elementos do dicionário acima caso estes sejam redeclarados no decorrer do programa.
- **varNotDec**: lista de pares que contêm nomes de variáveis e a respetiva função onde foram utilizadas mas nunca declaradas.
- **_current_scope_vars** e **current_func**: parâmetros auxiliares que contêm o nome e as variáveis da função atual para que seja facilitado o acesso às mesmas nas verificações.

3.1.1. Variáveis e o seu estado

Para realizar a análise de variáveis, começámos por classificar os nós da gramática em quatro categorias, consoante o impacto que têm no estado das variáveis. Após essa categorização, cada caso é processado de forma distinta:

- **Declarações**: Quando identificada uma declaração de variável, adicionamos um novo elemento ao dicionário `_current_scope_vars`, associando-lhe o tipo e inicializando o contador de utilizações a 0.
- **Redeclarações**: Se uma variável for `redeclarada`, o seu estado atual é guardado na estrutura `varReDec`, usando como chave o nome da função atual (`current_func`). Em seguida, a variável é reiniciada no dicionário `_current_scope_vars`, desta vez com o campo de redeclaração marcado como verdadeiro.
- **Utilizações**: Sempre que uma variável previamente declarada é utilizada, o seu contador de ocorrências é incrementado.
- **Não-declarações**: Se uma variável for utilizada sem ter sido previamente declarada, é registada em `varNotDec` como um par composto pelo nome da função e o nome da variável.

As variáveis de estado (`_current_scope_vars` e `current_func`) são guardadas e reiniciadas no final da definição de cada função.

As produções da gramática foram classificadas da seguinte forma:

- **Declarações**: `param`, `foreach_loop`, `variable_declaration`
- **Redeclarações**: `variable_declaration`, `allocation`
- **Utilizações**: `allocation_aux`, `print_command`, `array_access`, `expr`
- **Não-declarações**: `allocation`, `allocation_aux`, `print_command`, `array_access`, `expr`

📦 Variáveis Declaradas

Função	Variável	Tipo	Ocorrências	Re-declarada
add	a	int	3	❌
add	b	int	3	❌
add	arroz	int	0	✅
sub	a	int	1	❌
sub	c	int	1	❌
main	i	int	0	✅
main	arr	array	0	✅
main	j	int	4	❌
main	b	int	1	❌
main	result	int	1	❌
main	x	int	3	❌

Figura 1: Exemplo de variáveis declaradas do input no anexo1

🚫 Variáveis Não Declaradas

Função	Variável	Status
main	a	❗ Não declarada
main	y	❗ Não declarada

Figura 2: Exemplo de variáveis não declaradas do input no anexo1

♻️ Variáveis Re-declaradas

Função	Variável	Tipo	Ocorrências
add	arroz	int	0
main	arr	array	4
main	i	int	4

Figura 3: Exemplo de variáveis redeclaradas do input no anexo1

3.1.2. Total de varáveis declaradas por cada Tipo de dados

No final da execução do interpretador, e uma vez que está guardado em **varDec** pelo menos um estado de cada variável declarada, é possível calcular o total de variáveis por tipo de dados percorrendo o dicionário.

Total de Variáveis por Tipo de Dados

Tipo de Dado	Total
int	10
array	1

Figura 4: Exemplo de total de variáveis por tipo do input no anexo1

3.1.3. Possíveis erros a serem detetados

Tendo estes dados devidamente guardados, podemos posteriormente detetar anomalias:
- Variaveis não declaradas;
- Variaveis redeclaradas;
- Variaveis redeclaradas antes de serem usadas;
- Variaveis redeclaradas mas nunca usadas depois.

Warnings e Erros

```
⚠️ Warnings:
  - [Unused] A última declaração de 'arroz' em 'add' nunca foi usada (ocorr=0) → pode ser
apagada.
  - [Redeclared] 'arroz' em 'add' foi redeclarada.
  - [Unused] A última declaração de 'i' em 'main' nunca foi usada (ocorr=0) → pode ser apagada.
  - [Redeclared] 'i' em 'main' foi redeclarada.
  - [Unused] A última declaração de 'arr' em 'main' nunca foi usada (ocorr=0) → pode ser apagada.
  - [Redeclared] 'arr' em 'main' foi redeclarada.
  - [Redeclared & Unused] 'arroz' em 'add' foi redeclarada, mas a anterior nunca foi usada → pode
ser apagada.

❌ Erros:
  - [Not Declared] 'a' foi usada em 'main' mas nunca foi declarada.
  - [Not Declared] 'y' foi usada em 'main' mas nunca foi declarada.
```

Figura 5: Exemplo de avisos e erros do input no anexo1

3.2. InterpreterCondicoesECiclos

Este interpretador visa agrupar a informação importante sobre todas as instruções do programa, para assim ser possível:
- Calcular o total de instruções que formam o corpo do programa bem como o tipo das mesmas;
- Calcular o total de situações em que estruturas de controlo surgem aninhadas;
- Listar as situações em que existam ifs aninhados que possam ser substituídos por um só if.

Para isso, decidimos inicializa-la com os seguintes parâmetros:

- **counts**: um conjunto de contadores para `Atribuições`, `escrita`, `condicionais` e `ciclos`;
- **aninhamentos**: um contador de estruturas aninhadas;
- **if_fundidos**: uma lista de tuplos com informação sobre os condicionais fundíveis;
- **control_stack**: uma *stack* auxiliar para que seja facilitado a identificação de estruturas aninhadas.

3.2.1. Instruções que formam o corpo do programa

Durante a analise do input, o interpretador incrementa os respetivos contadores sempre que encontra:
- um instrução onde altera o valor de uma variável (`variable_declaration` ou `allocation`);
- uma instrução de impressão (`print_command`);
- uma condicional (`match_command`,`option_command`,`standard_command`, `check_command`, `also_command`, `otherwise_command`);
- uma instrução de ciclo(`loop_loop`,`foreach_loop`,`for_loop`).

Total de Instruções por Tipo

Tipo de Instrução	Total
Atribuições	15
Escrita	8
Condicionais	11
Ciclos	3

Figura 6: quantidade de instruções por tipo do input no anexo1

3.2.2. Estruturas de controlo surgem aninhadas

Durante a análise, sempre que é encontrada uma condicional ou uma instrução de ciclo, o nome da instrução é adicionado à *stack* auxiliar antes de serem visitados os seus nodos internos.

Caso essa instrução esteja aninhada dentro de outra estrutura de controlo, a *stack* terá um tamanho superior a zero. Este facto permite identificar aninhamentos e, consequentemente, incrementar o contador de estruturas aninhadas.

Após a análise dos nodos internos da condicional ou do ciclo, o nome da instrução é então removido da *stack*, garantindo que a estrutura da pilha reflete corretamente o contexto atual da análise.

Estruturas de Controlo Aninhadas

3

Figura 7: Aninhamentos identificados usando como input o anexo1

3.2.3. Otimização de Condicionais

Para identificarmos situações em que condicionais `check_command` podem ser fundidos, adicionámos uma verificação adicional durante a análise deste nodo.

Sempre que um `check_command` contiver apenas outro `check_command` no seu interior, e não for seguido por um `also_command` ou um `otherwise_command`, consideramos que essa estrutura pode ser simplificada.

Nesses casos, a informação relevante é guardada na variável **if_fundivel**, permitindo que posteriormente essa ocorrência seja apresentada como uma possível melhoria ao código analisado.

Ifs Aninhados que Podem Ser Fundidos

#	Função	Condição Exterior	Condição Interior	Nova condição
1	add	a < 10	b > 0	a < 10 && b > 0

Figura 8: Proposta de junção de condicionais usando como input o anexo2

4. Construção do html

Como requerido no enunciado, o resultado da análise ao código-fonte deve ser representado numa página *HTML* dedicada. Esta tarefa é realizada através da execução do ficheiro **executador.py**, que, após efetuar a travessia da árvore sintática utilizando os interpretadores previamente descritos, recolhe o seus outputs (fornecido sob a forma de parâmetros) e procede à construção do ficheiro **relatorio_analise.html**. Este é então constituído por uma cópia do código usado como input e pelas diferentes tabelas apresentadas nos pontos anteriores.

4.1. Exemplo Práticos

4.1.1. Exemplo 1

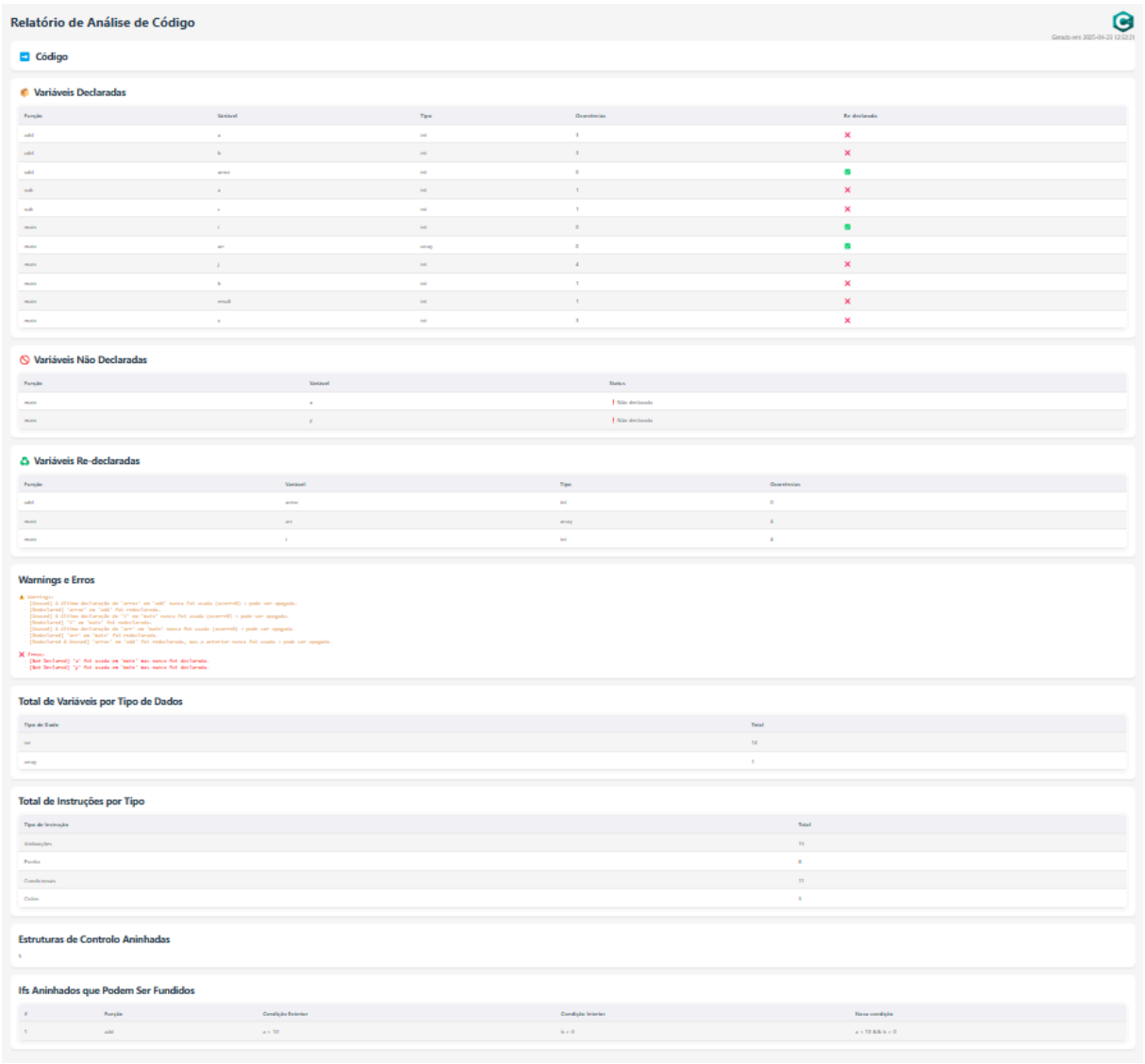

Figura 9: Exemplo de relatório-analise usando como input o anexo1

5. Conclusão

O desenvolvimento deste projeto representou não só um avanço significativo na compreensão da linguagem de programação definida na nossa gramática, como também contribuiu para a identificação de potenciais melhorias a nível da sua estrutura e utilização. Paralelamente, permitiu aprofundar o conhecimento sobre o poder e a versatilidade dos interpretadores no contexto da análise e desenvolvimento de linguagens de programação.

Assim, este estudo vem reforçar a importância da Engenharia Gramatical como pilar fundamental no desenvolvimento de linguagens de programação, promovendo soluções que garantem maior consistência, legibilidade e facilidade de manutenção do código, ao mesmo tempo que incentivam a automatização e análise de alto nível.

6. Anexos

6.1. Anexo A: Exemplo 1 de input

```
task add(int a, int b) -> int {
    let int arroz = 10;
    let int arroz = 10;
    check (a < 10) {
        check (b > 0) {
            b += 1;
        }
    }
    a++;

    return a + b;
}

task sub(int a, int c) -> int {
    return a - c;
}

task main() -> int {
    let int i = 0;
    a++;
    let array arr = [3,2,1];

    loop (i <= 5) {
        show(i);
        i++;
    }

    for (let int j = 0; j < 3; j++) {
        check(i == j) {
            break;
        }
        show(j);
    }

    foreach (b in arr) {
        show(b);
    }

    let int result = add(3, 4);
    show(result);

    let int x = 10;
    check (x < 5) {
        check (y > 11) {
            show("x is less than 5");
        }
    } also (x == 10) {
            show("x is 10");
    } otherwise {
        show("x is greater than 5 and not 10");
    }
```

```
    show(arr[2]);

    arr[2] = arr[3];

    arr = [1,2,3];

    i = 0;

    match (x) {
        option 5:
            show("x is 5");
        option 10:
            show("x is 10");
        standard:
            show("x is neither 5 nor 10");
    }



    return 0;
}
```

6.2. Anexo B: Exemplo 2 de input

```
task add(int a, int b) -> int {
    check (a < 10) {
        check (b > 0) {
            b += 1;
        }
    }
    return a + b;
}

task main() -> int {
    let int x = 10;
    let int y = 10;

    check (x < 5) {
        check (y > 11) {
            show("x is less than 5 and y is greater than 11");
        }
    } also (x == 10) {
            show("x is 10");
    } otherwise {
        show("x is greater than 5 and not 10");
    }
    return 0;
}
```