Mestrado em Engenharia Informática
Engenharia Gramatical Abril, 2024

TP2 - Analisador de Código Fonte

Grupo 4

Vasco Oliveira (pg54269), João Loureiro (pg53924) and Luís Fernandes (pg54019)

Resumo: Este relatório aborda o desenvolvimento de uma ferramenta de análise de uma linguagem de programação imperativa, definida pelo grupo num trabalho anterior (TPC2). A ferramenta consegue analisar diferentes tipos de variáveis, sendo elas simples (inteiros, decimais, frases, etc), ou complexos (tuplos, listas, conjuntos, etc), como também detetar estruturas de seleção como *ifs* e ciclos (*for*, *while*, etc). O intuito da ferramenta é no final gerar um ficheiro *html* onde estará presente as diferentes informações obtidas após a análise da LPI.

Palavras-chave: Lark, Regras Sintáticas, Regras Léxicas, Gramáticas, Programação Imperativa, Análise, Código Fonte

Link do Repositório: github.com/LuizFernandez/EG_Projetos

Table of Contents

1 Introdução

Este trabalho teve como objetivo criar um analisador de código fonte, com recurso ao módulo *Lark* de *Python*, de uma linguagem de programação cuja gramática foi esboçada por nós. O analisador deve conseguir detetar várias situações de erro (declarações erradas, tipos incompatíveis, etc...) bem como fornecer várias estatísticas sobre o código fonte, guardando-as numa página *HTML* para serem posteriormente observadas.

2 Gramática da Linguagem

Nesta secção será apresentada a nossa gramática e as regras sintáticas e lexicográficas da linguagem de programação em desenvolvimento.

Decidimos criar uma linguagem de uso fácil, tal como *Python*, optando por uma sintaxe mais simples e legível, mas tirando a preocupação que é preciso ter com a indentação, e ao mesmo tempo deixando-a mais consistente, onde os utilizadores ao declararem uma variável têm de especificar o seu tipo. A gramática encontra-se especificada no ficheiro *grammar.txt*.

2.1 Regras Sintáticas e Lexicográficas

2.1.1 Regras Sintáticas:

1. **start:** Define o ponto de entrada do programa, onde pode haver zero ou mais **statements**.
2. **statements:** Representa uma declaração, atribuição, seleção ou ciclo.
3. **declaration:** Define a sintaxe para declarar variáveis.
4. **attribution:** Define a sintaxe para atribuir valores a variáveis.
5. **type:** Define os tipos dos **values** que podem ser utilizados em declarações, estando separado em tipos primitivos [*primitve_types*] (int, float, string, bool) e tipos complexos [*complex_types*](*list*, *set*, *tuple*, *array*).
6. **value:** Representa tudo o que consiste num valor, uma variável, valores de tipos primitivos [*primitive_value*] e valores de tipos complexos [*complex_value*].
7. **list/set/tuple/array_construct:** Definem as diferentes estruturas possíveis para estes valores de tipos complexos.
8. **selection:** Define a estrutura de controle condicional *if-else*.
9. **cycle:** Define a estrutura dos diferentes ciclos, contendo ciclos de tipo *while*, *do-while* e *for*.
10. **operation**: Regras que estabelecem a estrutura e a ordem das operações aritméticas e lógicas, abordando os diferentes operadores que estão definidos num formato de prioridade.

2.1.2 Regras Lexicográficas:

1. **ID:** Define os padrões para identificadores (exemplos: a, a2, count_list, errorMap).
2. **INT:** Define os padrões para números inteiros.
3. **FLOAT:** Define os padrões para números de ponto flutuante.
4. **STRING:** Define os padrões para strings (exemplo: "string").
5. **BOOL:** Define os padrões para booleanos (True/False).
6. **IF/WHILE/DO/FOR:** palavras-chave para if, while, do e for. Usados para identificar o ínicio de uma estrutura de controlo.
7. **EQ, NE, LT, LE, GT, GE:** Definem os operadores de comparação, neste casos, '==', '!=', '>', '>=', '<', '<='.
8. **ADD, SUB, OR**: Representam os símbolos '+', '-', '||', respetivamente.
9. **MUL, DIV, AND, MOD**: Representam os símbolos '*', '/', '&&', '%', respetivamente
10. **POT**: Representa o símbolo de potência.

3 *Lark Interpreter*

Recorrendo ao *Interpreter* do módulo *Lark*, criamos uma classe, **MyInterpreter**, que constrói uma árvore através do código fornecido e baseado na nossa gramática, sendo inicializada com os seguintes parâmetros:

- ***vars:*** dicionário que guarda o nome, tipo e valor de todas as variáveis declaradas, bem como se estas são posteriormente utilizadas.
- ***erros:*** lista que armazena todos os erros detetados relacionados às variáveis presentes no código.
- ***types:*** dicionário que armazena todas as variáveis declaradas no código para um dado tipo.
- ***count:*** dicionário que armazena a quantidade de atribuições, declarações, seleções e ciclos existentes no código.
- ***nesting:*** variável que toma conta da quantidade de estruturas de controlo aninhadas em outras estruturas de controlo.
- ***sub_ifs:*** lista que armazena várias listas cujos elementos pertencem a condições *if* aninhadas que podem ser substituídas por uma só condição *if*.

3.1 Variáveis e os seus estados

O *Interpreter* utiliza o seu parâmetro *vars* para armazenar todas as variáveis corretamente declaradas, isto é, cujos tipos e valores coincidem corretamente. Esta verificação é feita quando ele identifica uma declaração no código, e caso esteja incorreta (ex: `int a = "Hello World"`), é guardada no parâmetro *erros* com o tipo de erro (no exemplo dado, *type*), em vez de no *vars*.

Se encontrar uma atribuição, ele verifica primeiro se é uma atribuição válida (ex: `b = p`, sendo b e p do mesmo tipo). Se for, atualiza o valor da variável em *vars*, caso contrário, anota o erro em *erros*.

Após uma variável estar presente em *vars*, se o *Interpreter* detetar uma utilização dela no código, sinaliza como tal no campo *used* da entrada correspondente.

Variables and Types Used

Variable Name	Type	Value	Used?
a	int	10	True
b	string	"Hello World"	True
f	int	10	True
l	list	[1, 2, 3, 4, 5]	False
ar	array	[1, 2, 3]	False
l2	list	[1, 2, 3, 4, 4, 4, 5, 2]	True
s	set	{1, 2, 3, 4, 5}	True
z	int	10	False
a_list	list	[1, 2, 3]	False
b_list	list	[1, 2, 3]	False
c_list	list	[1, 2, 3]	False

Figure 1: Tabela de Variáveis e o seu respetivo tipo e valor

3.2 Deteção de erros

Como mencionado anteriormente, caso o *Interpreter* encontre um erro, este é registado com a sua devida classificação no parâmetro *erros* (Exemplo, Figura 2). Existem 7 tipos de erros:

- **Redeclaration:** código *'redeclaration'*. Redeclaração, quando se tenda declarar novamente uma variável previamente declarada (ex: `int a = 4 ... float a = 5.3`)
- **Not Declared:** código *notDeclared*. Não-declaração, menção de uma variável não declarada previamente. (ex: `r = 7`)
- **Type:** código *'typing'*. Tipagem errada, quando se tenta atribuir um valor do tipo diferente ao esperado pela a variável (ex: `int a = "Babba Booey"`)
- **Missing:** código *'missing'*. Utilização de uma variável que não foi previamente declarado (ex: com c não declarado, `if{c}{...}`)

- **Failed to Declare:** código *'failedD'*. Declaração de uma nova variável a partir de uma variável que não foi previamente declarada (ex: `bool d = p`)
- **Failed to Assign:** código *'failedA'*. Atribuição a uma variável com recurso a uma variável que não foi previamente declarada (ex: com *a* previamente declarado, `a = p`)
- **Invalid Argument Type:** código *faildArgC*. Utilização de variáveis de tipos diferentes aos exigidos por uma dada função (ex: `int a = 10 ... list l3 = list(a)`, *list()* necessita de uma coleção como argumento)

Errors

Variable	Type	Description
r	Not Declared	Variable is not declared
k	Type	Value type ("Rhello Uerlde" => str) doesn't match variable type (int)
a	Redeclaration	variable is already declared
b	Type	Value type (2 => int) doesn't match variable type (string)
p	Missing	Variable p isn't declared for use
p	Missing	Variable p isn't declared for use
a	Failed to Assign	Variable a of type int failed to recive a new value beacuse variable p doesn't exist
p	Missing	Variable p isn't declared for use
d	Failed to Declare	Variable d of type bool failed to be declared because variable p doesn't exist
c	Type	Value type (10 => int) doesn't match variable type (float)
a	Invalid Argument Type	Function list() does not accept variable a of type int, because of being of a different type
l3	Failed to Declare	Variable l3 of type list failed to be declared because variable a doesn't exist
c	Missing	Variable c isn't declared for use
d	Missing	Variable d isn't declared for use
e	Missing	Variable e isn't declared for use
c	Redeclaration	variable is already declared

Figure 2: Tabela de Erros detetados

3.3 Total de variáveis por tipo de dados

Consoante as frases de *input* são analisadas, sempre que uma dada variável passa pelos testes de declaração, isto é, não apresenta erros no formato da sua declaração, esta será armazenada numa estrutura ***types*** que contêm todas as variáveis declaradas de um dado tipo (Figura 3, exemplo de tabela).

Number of Variables Declared per Type

Type	Vars	Number
int	a, f, z	3
string	b	1
list	l, l2, a_list, b_list, c_list	5
array	ar	1
set	s	1

Figure 3: Tabela de variáveis por tipo

3.4 Total de instruções que formam o corpo do programa

À medida que o *Interpreter* analisa as frases dadas como *input*, sempre que encontrar um caso de **declaração**, **atribuição**, **seleção** ou **ciclo**, este irá utilizar a variável *count*, para assim armazenar cada ocorrência de cada instrução que apareça no *input*. Contudo, as instruções são apenas tomadas como ocorrência caso elas tenham sucesso, isto é, se na declaração ou atribuição, ocorrer um erro de tipos (valor atribuído não é do tipo da variável), esta ocorrência não será tida em conta.(Figura 8, exemplo de tabela)

Number of Instructions Found in the Code

Type	Number
Attributions	1
Declarations	17
Selections	8
Cycles	0

Figure 4: Tabela de ocorrência de instruções

3.5 Total de situações em que estruturas de controlo surgem aninhadas em outras estruturas de controlo

Para conseguirmos identificar estas situações, tivemos primeiro que as definir concretamente. Uma estrutura de controlo, como mencionado anteriormente, pode ser uma condição *if*, ciclo *while*, *do* ou *for*, ou seja, uma estrutura de controlo está aninhada noutra quando temos, por exemplo, uma condição ou um ciclo dentro do corpo de outra condição ou ciclo (ex: `while(a){ if(b){...} }`).

Dentro do Interpreter, para calcular os aninhamentos, usamos o parâmetro de class **nesting** em que cada estrutura de controlo vê quantas estruturas de controlo estão a um nível a baixo dele, por exemplo:

```
if(a){
    if(b){
        if(c){

        }
    }
    if(d){

    }
}
```

Listing 1: Exemplo

Aqui o *if* de valor "a" vai ver que tem os *ifs* de valor "b" e "d" dentro dele, enquanto que o *if* de valor "b" vai ver que tem o *if* de valor "c" dentro dele, acabando, assim, contando 3 estruturas de controlo aninhadas.

3.5.1 Mais exemplos:

```
if(a){
    if(b){
        if(c){

        }
    }
}
```

Listing 2: Exemplo 1

Resultado: 2 estruturas aninhadas.

Number of Control Structures Nested Within Other Control Structures

Found **2** nested control structures.

Figure 5: Resultado do primeiro exemplo

```
if(a){
    if(b){
        int c = 10
        if(c){

        } else if (d){

        }
    }
    while(e){
        c = c + 1
```

```
    }
}

int f = 0
do{
    f = f + 1
}while(f)
```

Listing 3: Exemplo 2

Resultado: 3 estruturas aninhadas, sendo que a estrutura *do* não conta porque não está aninhada a outra estrutura de controlo e if + else if também só contam como uma estrutura de controlo.

Number of Control Structures Nested Within Other Control Structures

Found **3** nested control structures.

Figure 6: Resultado do segundo exemplo

3.6 Lista de situações em que existam *ifs* aninhados que possam ser substituídos por um só *if*

Para identificarmos estas situações usamos mais ou menos a mesma lógica que a situação anterior, onde uma estrutura condicional *if* verifica se dentro dela só está outra estrutura condicional *if*, dando a sugestão de as juntar, exemplo:

```

if(a){
    list a_list = [1,2,3]
    if(b){
        if(c){
            if(d){
               list b_list = [1,2,3]
            }
        }
        list c_list = [1,2,3]
    }
    while(True){

    }
}

if(e){
   if(f){
        list c_list = [1,2,3]
   }
}
```

Listing 4: Exemplo

Nested 'If' Conditions That Can be Combined

Condition Nº1	Condition Nº2	Suggestion
if c	if d	if c and d
if e	if f	if e and f

Figure 7: Resultado do segundo exemplo

Neste exemplo o *if* de valor "e" verifica que dentro dele só tem outro *if* de valor "f", criando assim a sugestão de os juntar (`if(e \&\& f)`), e o mesmo caso acontece para os *ifs* de valor "c" e "d". Já o *if* de valor "a", apesar de ver que tem outro *if* de valor "b" dentro dele, ele também repara que tem outro **statment** dentro dele (`while(True)`) acabando por não sugerir juntar esses *ifs*, o mesmo acontece para o *if* de valor "b" que tem também outro **statment** dentro dele (`list c_list = [1,2,3]`).

4 Construção do *HTML*

Como requisitado no enunciado, o resultado da análise do código-fonte deve ser representada numa página *HTML* construída para esse propósito. Isto é efetuado através da execução do ficheiro *ling.py* que, após fazer a travessia com recurso ao *interpreter* previamente explicado, pega no seu output (parâmetros deste) e constrói um ficheiro *HTML*, ***analise.html*** onde os resultados são expostos através de secções e tabelas.

Adicionalmente, cria também 4 tabelas no formato *markdown*, cada uma no seu respetivo ficheiro *.md* na pasta *Tables* que contém a mesma informação das tabelas do *html*, se o utilizador desejar observar nesse formato.

4.1 Exemplo de *output*/*output*

Input:

```
if(a){
    if(b){
        int c = 10
        if(c){

        } else if (d){

        }
    }
    while(e){
        c = c + 1
    }
}

int f = 0
do{
    f = f + 1
```

```
18 }while(f)
```

Listing 5: Input

Output:

Variables and Types Used

Variable Name	Type	Value	Used?
a	int	10	True
b	int	2	True
c	int	12	False

Errors

Variable	Type	Description
flag	Type	Value type (True => bool) doesn't match variable type (string)
f	Not Declared	Variable is not declared
flag	Missing	Variable flag isn't declared for use

Number of Variables Declared per Type

Type	Vars	Number
int	a, b, c	3

Number of Instructions Found in the Code

Type	Number
Attributions	3
Declarations	4
Selections	3
Cycles	0

Number of Control Structures Nested Within Other Control Structures

Found **1** nested control structures.

Nested 'If' Conditions That Can be Combined

Condition Nº1	Condition Nº2	Suggestion
if flag	if 2 >= 10	if flag and 2 >= 10

Figure 8: Output

5 Conclusão

Com o desenvolvimento desta ferramenta, conseguimos aprofundar o nosso conhecimento obtido nas aulas sobre as ferramentas *lark* e *Interpreters* e aprender a importância de uma boa definição das regras sintáticas e léxicas.

Um dos maiores problemas que a equipa encontrou no desenvolvimento foi devido a uma má definição das regras que constituíam a gramática inicialmente, o que levou a diversas alterações na sua estrutura para assim ficar mais conveniente a abordagem que o *Interpreter* teria sobre uma dada frase.

Contudo, após a finalização do projeto, é de notar que certos aspetos ficaram incompletos, como por exemplo, nas atribuições e declarações não são tidas em conta todas as situações de erro, por exemplo, uma atribuição em que contêm duas variáveis não declaradas.

Estamos bastante orgulhosos com o resultado final e consideramos que fizemos uma ferramenta de análise de código bastante competente, bem como uma linguagem de programação que desenvolvemos para utilizar neste projeto.