Engenharia Gramatical (1º ano MEI)

Trabalho Prático 2

Relatório de Desenvolvimento

Gonçalo Ferreira (pg50404)

Rui Braga (pg50743)

1 de maio de 2023

Resumo

O segundo trabalho prático, de Engenharia Gramatical consistiu na criação de uma linguagem de programação imperativa e um analisador de código, capaz de gerar um relatório com informações acerca de um ficheiro de input, escrito na linguagem criada.

Esse relatório deverá conter diversas informações relacionados com as instruções utilizadas, variáveis declaradas e apresentar o código anotado, com todos os erros/avisos encontrados durante a análise.

No presente documento é descrita linguagem criada, como foi conseguida a análise do código e como foi compilado e gerado o relatório. Por fim, serão também apresentados os testes realizados de forma a demonstrar a capacidade do analisador.

Conteúdo

Capítulo 1

Introdução

Área: Processamento de Linguagens

Supervisor: Prf. Pedro Rangel Henriques e Tiago João Fernandes Baptista

Incorporado na componente prática da unidade curricular de Engenharia Gramatical (incluída no perfil de Engenharia de Linguagens do Mestrado em Engenharia Informática da Universidade do Minho), o presente documento, documentará o processo de desenvolvimento do segundo projeto prático da UC, que exige a criação de uma LPI (linguagem de programação imperativa), em conjunto com o seu analisador léxico, que deverá produzir um relatório sobre a análise realizada ao código.

Para isto, o grupo utilizou a linguagem, por si desenvolvida e já apreciada pelos docentes, com algumas alterações, como a base do projeto, restando assim, o desenvolvimento do analisado léxico que percorrerá a AST criada, para criar o relatório da análise, em HTML. Neste relatório constarão informações relativas às variáveis e instruções utilizadas, mudanças de contexto identificadas e o código anotado com possíveis avisos e erros encontrados no decorrer da análise.

Deste modo, este relatório começará por uma breve apresentação da gramática utilizada na definição na linguagem, dando especial atenção às alterações realizadas após a apreciação da mesma, avançando depois, para uma descrição do desenvolvimento do analisador léxico da AST; O capítulo seguinte incidirá sobre a criação e compilação dos resultados, no relatório HTML, e por fim será demonstrado o funcionamento da ferramenta, com alguns exemplos de utilização e testes, capazes de demonstrar todas as funcionalidades, descritas em capítulos anteriores. Na conclusão deste documento, será realizado um balanço geral sobre o projeto desenvolvido, realçando os pontos positivos e aspetos a melhorar na ferramenta.

Capítulo 2

Gramática

Uma linguagem imperativa é uma linguagem, composta por ordens simples e ordenadas, que é capaz de alterar o estado de um programa usando essas instruções (*statements*). O hardware de um computador é concebido para executar código máquina, que normalmente, é escrito num estilo imperativo. Por consequência, maior parte das linguagens de computação são imperativas, incluindo as mais conhecidas e usadas.

A LPI a ser implementada deverá permitir declarar e manipular variáveis que podem ser do tipo inteiro, string, booleano, array, lista ou tuplo, assim como, escrever instruções de atribuição, de leitura e escrita, seleção de elementos de objetos iteráveis (arrays, listas e tuplos), inserções e testes de pertença a listas, mecanismos de controlo de fluxo como *if* e *switch*, e ciclos do estilo *while*, *do while* e *for* para um determinado intervalo. A LPI criada deverá também ser capaz de inferir valores resultantes de expressões que usam operadores aritméticos, lógicos ou relacionais.

A linguagem desenvolvida é, em grande parte, inspirada pelo C, com algumas ideias oriundas do Python.

No exemplo abaixo, é demonstrado como são feitas atribuições para os diversos tipos na linguagem desenvolvida. Destaca-se a distinção que foi feita entre array e lista e a capacidade de selecionar um elemento nas estruturas que o permitem (arrays, listas e tuplos).

```
int i = 5;
bool b = true;
string s = "example string";
int[3] v = [1,2,3];
i = v[2];
list l = {"abc", true, 10};
tuple t = (15, "xyz")
```

Um array tem um tamanho fixo e todos os seus elementos são de um determinado tipo, enquanto que uma lista pode ter tamanho variável e não tem qualquer restrição de tipo.
No exemplo seguinte, estão explicitadas as instruções possíveis de construir com a gramática desenvolvida.

```
if i < 20 { string s = "i é menor que 20"; }
elif (i < 40) { string s = "i é menor que 40"; }
else { string s = "i é maior ou igual a 40"; }

while (i < 100) { i = i + 1; }

do { i = random(100, 200); } while i < 150

for var in [1,2,3] {}
for val in values {}
for d in [1 -> 10] {}

switch i
case 5 {}
case 10 {}
default {}
```

Realça-se a possibilidade de determinar a condição de um controlo de fluxo ou de um ciclo, com ou sem parênteses e a necessidade de colocar chavetas à volta do que é executado numa determinada estrutura cíclica ou de controlo. É possível executar ciclos *for* recorrendo a arrays (como exemplificado), listas e strings, bem como, também é possível usar funções existentes (como *cons* ou *write*) ou definidas no próprio código fonte como no seguinte exemplo.

```
def sum(arg1, arg2){
    return arg1 + arg2;
}
```

Com isso, resta especificar como foram concebidas as ordens de prioridades nas expressões que recorrem a operadores aritméticos e lógicos.

Para operações lógicas, a 1ª ordem de prioridade tem os elementos básicos, que são os booleanos existentes (true ou false), comparações que usam operadores relacionais (≪", ≫", "==", etc...) ou outras operações lógicas definidas com maior prioridade através de uso de parênteses. No nível de prioridade acima, encontra-se a negação de uma condição, que é feita com o terminal "not". Num nível acima, está a operação "e"que é feita com o terminal "and"e, por último, a operação "ou", que usa o terminal "or".

Para expressões que recorrem a operadores aritméticos, é usado um raciocínio análogo. É dada prioridade de reconhecimento aos elementos básicos, que são *int* ou qualquer elemento que possa representar um número, como variáveis ou o resultado de uma função. Essa prioridade inicial é dada, também a expressões que estejam dentro de parênteses e à exponenciação. O 2º nível de prioridade contém as operações de multiplicação (*), divisão (/) e resto (%) que são executadas primeiro que a adição (+) e a subtração (-).

Capítulo 3

Processamento da AST

Com o processamento da gramática concluído e a criação da "Abstract Syntax Tree"para o código de input (com recurso ao Lark), é agora necessário realizar a travessia da árvore para obter dela, a informação necessária para a composição do relatório de análise. Para isto será utilizado o módulo *Lark.Interpreter,* que nos permitirá percorrer a árvore de forma personalizada a cada ramo.

Para o relatório de análise, foram pedidos do analisador, os seguintes objetivos:

- A criação de uma listagem das variáveis declaradas e utilizadas no código fonte
- A marcação de casos de redeclaração, não-declaração, não-utilização e não inicialização de variáveis
- A contagem de instruções do corpo do programa, de acordo com o seu tipo (escrita, leitura, atribuição, condições e ciclos)
- Marcação de casos de aninhamento de estruturas de controlo
- Marcação de casos onde seria possível juntar condições "if"aninhadas, numa só

Aos quais o grupo decidiu adicionar os seguintes requisitos de forma a tornar, o analisador numa ferramenta mais dinâmica:

- Cálculo de expressões numéricas ou booleanas
- Verificação de tipos em atribuições, expressões numéricas e booleanas
- Criação de uma hierarquia de contextos, onde uma variável declarada num dado contexto, será utilizável, no código do contexto, e em todos os contextos descendentes, mas não em contextos paralelos ou ascendentes
- Marcação de erros ou avisos, em caso de deteção de erro ou inconsistência

Para poder preencher todos estes requisitos, o analisador necessitou de manter um estado complexo, composto pelas seguintes variáveis:

- **typeCount** - Dicionário de contadores, responsável por manter um contador, por cada tipo de operações: escrita, leitura, atribuição, condição e ciclo. Sempre que uma destas instruções é detetada na AST, é incrementado o respetivo valor do dicionário
- **instrCount** - Contador de instruções utilizadas no código de input
- **declVar** - Dicionário de variáveis, com base no contexto em que foram definidas. É assim composto por um dicionário de contextos, que por sua vez contém um dicionário de variáveis definidas no respetivo contexto. Cada variável é representada por um dicionário que contém o seu tipo, e uma lista de valores atribuídos a esta variável durante o programa
- **declFun** - Dicionário composto pelo conjunto das funções definidas no código fonte. Cada função é representada por um dicionário que contém os seus argumentos e uma listagem de tipos de retorno (que será atualizada sempre que for encontrado uma instrução "return"no interior de uma função
- **contextTree** - Árvore de contexto, que possui a hierarquia de contextos
- **unused / undeclared / notInit** - Listas de warnings/erros, que surgiram em consequência de um erro no código de input na atribuição/declaração de uma variável. Sempre que é detetado um caso destes, é adicionado a esta lista, um dicionário que especifica a linha, e carater de inicio e de fim, do token reportado
- **warnings** - Lista de warnings gerais encontrados no código
- **errors** - Lista de erros encontrados no código

A regra "start"retorna também o agregado das informações retiradas sobre todas as instruções, o que resulta numa lista de dicionários que representa o código analisado.

Sempre que é detetada uma declaração de função, ou uma estrutura de controlo, o seu conteúdo será considerado um novo contexto. Todos os contextos são descendentes do contexto global, e caso exista uma redeclaração de uma variável num contexto descendente, daquele onde a variável foi inicialmente declarada, esta será considerada uma nova variável.

Quanto a deteção de possibilidade de junção de "if"s aninhados, o grupo implementou a seguinte condição: ***Uma condição "if"só pode ser conjugada com uma condição parente, se e só se, as variáveis utilizadas na condição parente, não são utilizadas na condição aninhada; se as variáveis utilizadas na condição aninhada, não foram alteradas entre o corpo da condição parente e a condição aninhada e se a condição aninhada não possui "elif"ou "else"s associados.*** Esta condição apenas reportará possibilidades de junção, quando a alteração do código necessária, apenas implicaria a conjunção entre as duas condições, apesar de existirem outros casos onde o aninhamento poderá ser desfeito com mais alterações ao código.

Capítulo 4

Composição do Relatório HTML

Com o processamento da AST completo, basta agora compilar a apresentar os resultados sobre a forma de um relatório HTML. O relatório é composto pelos seguintes elementos:

- Tabela das variáveis utilizadas - tabela que apresenta o nome, tipo e valores atribuídos à variável
- Quadro das funções declaradas - tabela que apresenta o nome, argumentos e tipos de retorno de cada uma das funções declaradas
- Contador de instruções - apresenta o número total de instruções do programa, incluindo o número total de instruções por tipo
- Árvore de contexto criada - elemento html que utiliza um pouco de javascript e css para apresentar a árvore de contexto criada, em conjunto com o número de instruções utilizadas em cada contexto
- Código anotado - secção que apresenta o código de input, com as anotações de erros/warnings encontrados durante a execução do analisador
- Output do analisador - lista de dicionários resultante da travessia do analisador pela árvore de sintaxes abstrata, criada a partir do código fonte

Antes de construir o HTML, foi ainda necessário algum processamento das variáveis retornadas pelo analisador, nomeadamente a organização dos erros numa estrutura de dados que permitisse a procura dos mesmos linha a linha, e por sua vez, carater a carater, para permitir a funcionalidade de anotação do código fonte.

Capítulo 5

Demonstração e Testes

Para a secção de testes, será apresentado um excerto de código escrito com a linguagem criada, que servirá de input ao analisador, e que levará à criação do relatório, que será também apresentado em secções para análise. Serão também incluídos erros de forma propositada, para demonstrar o comportamento do analisador perante estas situações.

```
int global_i = 10 + 30;
string frase = 3;
string global_i= "oi";

global_i = myFunc(1,2);

bool[3] arrayBool = [true,false,true];
x = 13;
bool k = arrayBool[4];

def foo(int l){
    for i in [1 -> 10] {
        if (i < global_i) {
            write("foo","STDOUT");
        }
        elif (i > global_i) {
            write("bar","STDOUT");
        }
        if (i == l){
            write("Found it!","STDOUT");
        }
    }
    return l;
}
int x = foo(10);
```

Neste código conseguimos encontrar os seguintes erros:

- O tipo da variável "frase"e o valor que lhe é atribuído não são compatíveis
- A variável "global_i"está a ser redefinida na 3ª linha
- A função "myFunc"não existe
- A variável "x"ainda não foi declarada
- A atribuição do elemento 4 do array "arrayBool"a k é impossível
- Várias variáveis são declaradas mas não utilizadas

5.1 Relatório HTML

Quanto as variáveis, o relatório apresenta a seguinte tabela:

Code Analyzer Report:

Variables Used

Name	Type	Context	Values
global_i	int	global	[{'int': 40}]
arrayBool	{'darray': {'type': 'bool', 'size': '3'}}	global	[{'array': [{'bool': True}, {'bool': False}, {'bool': True}]}]
k	bool	global	[None]
x	int	global	[{'func': {'name': 'foo', 'args': {'int': 10}, 'retType': ['int']}}]
l	int	foo	['arg']
i	int	floop1	[{'range_explicit': {'min': '1', 'max': '10'}}]

Figura 5.1: Tabela com as variáveis

- O valor da expressão global_i foi calculado como a soma de $10 + 30 = 40$
- A declaração do array especifica o tipo dos valores interiores ao array e o tamanho fixo do mesmo
- Devido ao erro, k é inicializado sem valor, podendo assim ser redefinido
- A "x"é atribuido o valor da func "foo"com o argumento 10
- A "l"é atribuído o valor "arg"que especifica que este valor terá um valor atribuído com base nos argumentos passados à função da qual faz parte
- A variável "i"só existe no contexto do ciclo for possuindo o valor da range especificada

Quanto as funções declaradas, podemos verifcar que o analisador é capaz de identificar o tipo de retorno da função corretamente, bem como os seus argumentos. Quanto a contagem de instruções utilizadas, o analisador conta 16 instruções que não incluem a def..... A contagem de tipos parece

estar correta tendo em conta que a atribuição de ”i”ao valor da range, também corresponde a uma atribuição.

Functions Declared

Name	Arguments	Return Type
foo	[('int', 'l')]	['int']

Intructions Used

In total, there were 16 instructions used!

In the table bellow, we can see these instructions organized by type:

Type	Number
attr	9
read	0
write	3
cond	3
cycle	1

Figura 5.2: Tabela com as funções

A árvore de contexto está apresentada na figura 5.3, também corretamente descrita.

Instruction Counter based on Context

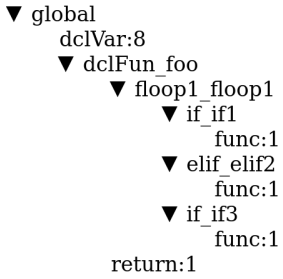

Figura 5.3: Hierarquia de contextos

E por fim o código anotado, que assinala que considera possuir erros ou avisos. Na figura 5.9 conseguimos verificar que o analisador encontrou os erros que foram descritos anteriormente, os quais são descritos de acordo com o erro provocado, com uma mensagem pop-up, que surge quando um utilizador realiza um ”hover”sobre o elemento assinalado com um erro.

De notar que o texto colorido a cor amarela/alaranjada representa um aviso, que poderá ser originário da não utilização de uma variável, ou variáveis cujo tipo não pode ser identificado, e o texto colorido vermelho sinaliza erros, mais graves, como acessos a array fora dos indíces possíveis, a redeclaração de variáveis etc...

Annotated Code

Figura 5.4: Código anotado

Annotated Code

```
int global_i = 10 + 30;
string frase = 3;


string global_i= "oi";



global_i = myFunc(1,2);


bool[3] arrayBool = [true,false,true];
x = 13;

bool k = arrayBool[4];



def foo(int l){
    for i in [1 -> 10] {
        if (i < global_i) {
            write("foo","STDOUT");
        }
        elif (i > global_i) {
            write("bar","STDOUT");
        }
        if (i == l){
            write("Found it!","STDOUT");
        }
    }
    return l;
}
int x = foo(10);
```

Figura 5.5: Código anotado na sua totalidade

5.2 Ifs aninhados

Nesta secção serão apresentados os casos onde ifs podem ou não agregados a condições parentes:

```
 if (i/10 == 0) {
      y=20;
      if (y * 10 ==10) {
          write("FizzBuzz\t","STDOUT");
      }
```

Figura 5.6: Junção não sugerida devido à alteração da variável y

```
if (i/10 == 0) {
     x = "Hello";
     if (i * 10 ==10) {
         write("FizzBuzz\t","STDOUT");
     }
```

Figura 5.7: Junção não sugerida devido à utilização da mesma variável em ambas as condições

```
if (i/10 == 0) {
     x = "Hello";
     if (y * 10 ==10) {
         write("FizzBuzz\t","STDOUT");
     }
     elif (x == "Hello"){
         write("BuzzFizz\t","STDOUT");
     }
```

Figura 5.8: Junção não sugerida devido à existência de "elif"associado a condição aninhada

```
if (i/10 == 0) {
     x = "Hello";
     if (y * 10 ==10) {

         write("FizzBuzz\t","STDOUT");
     }
```

Figura 5.9: Junção possível e sugerida

Capítulo 6

Conclusão

Este relatório detalhou o processo de criação de uma LPI, as especificidades associadas a uma gramática do género, e a edificação de um analisador de código estático usando o módulo para geração de processadores *Lark.Interpreter*, capaz de a complementar, oferecendo relatórios de erros e/ou avisos sobre má conduta de escrita usando a LPI criada ou, simplesmente, oferecer diversas estatísticas sobre o eventual funcionamento do programa contendo informações sobre as variáveis e as instruções usadas. Por fim, foi também detalhado todo o raciocínio por detrás do processamento da AST que permitiu não só, fazer a análise do código mas também, gerar a página de HTML desejada.

O grupo considera ter atingido todos os objetivos definidos anteriormente, apresentando uma linguagem versátil e capaz de gerar vários tipos de programas, e um analisador de código, capaz de reportar uma grande variedade de erros, calcular expressões simples e fazer verificação de tipos (que foi talvez dos objetivos mais complexos de alcançar). O grupo faz assim um balanço positivo do projeto desenvolvido, que considera ter sido essencial na ampliação do seu conhecimento no desenvolvimento de linguagens de programação e respetivos analisadores de código.