Processamento de Linguagens

Trabalho Prático 2

Relatório de Desenvolvimento

(A89518-Ema Dias) (A93253-David Duarte) (A93281-Leonardo Freitas)

15 de maio de 2022

Resumo

Ao longo deste documento, será especificado todo o raciocínio utilizado na realização do projeto, assim como consequentes resultados do mesmo. Este projeto refere-se ao enunciado número 3, dos enunciados propostos pela equipa docente: Geradores de Parsers LL(1) Recursivos Descendentes.

Desta forma, o relatório tem como objetivo identificar o problema e objetivos fornecidos pelo corpo docente e respetivas decisões tomadas para atingir os mesmos, por parte da equipa de trabalho.

Conteúdo

Capítulo 1

Introdução

No segundo projeto da Unidade Curricular Processamento de Linguagens, tem-se como principal objetivo aumentar o conhecimento em engenharias de linguagens e programação gramatical, promovendo, assim, a escrita de gramáticas. Além disso, propõem-se a escrita e desenvolvimento de processadores de linguagens, assim como, um gerador de código para um determinado objetivo. Com isto, pretende-se familiarizar os alunos com o gerador de analisadores sintáticos *yacc* e o gerador de analisadores léxicos *lex*.
Para isso, a equipa docente forneceu quatro enunciados, dos quais o grupo de trabalho teve a liberdade de selecionar um.

1.1 Gerador de Parsers LL(1) Recursivos Descendentes

Área: Processamento de Linguagens

1.2 Enquadramento

Uma gramática é considerada LL(1) se for possível saber-se antecipadamente para qual das produções um símbolo irá derivar, com base na análise do token seguinte. A sigla LL(1) traduz-se em leitura da frase da esquerda para a direita, derivando-se pela esquerda, com um símbolo antecipável. Compreende-se, deste modo, que o conjunto dos símbolos que iniciam frases derivadas de dois lados direitos de produções do mesmo lado esquerdo devem ser disjuntos. Além disso, no caso da presença de símbolos anuláveis, por exemplo na derivação de A, o conjunto de símbolos que se iniciam a partir de A, deve ser disjunto do conjunto de símbolos que se possam seguir a qualquer sequência derivada de A. Uma gramática livre de contexto é tal que as produções são descritas da seguinte forma $A \rightarrow \alpha$, onde A é um símbolo não terminal e α um conjunto de símbolos terminais e/ou não terminais, podendo ser vazio. Para além disso, um gramática livre de contexto pressopõem que qualquer símbolo não terminal do lado esquerdo, pode ser substítuido pela cadeia descrita no lado direito.
Além demais, está é definida por um conjunto de símbolos não terminais, representados por letras maiúsculas, um conjunto de símbolos terminais, representados por letras minúsculas, um conjunto de produções e um símbolo inicial.

1.3 Contexto

Em contexto do problema explicitado de seguida , é necessário perceber o que é o parsing. Sendo que o mesmo, é o processo para determinar se o simbolo que inicia a gramática pode derivar o programa ou não. Se o Parsing for bem sucedido o programa é válido, noutro caso isso não se verifica.

Para entender o conceito de um Parser Recursivo Descendente é necessário perceber o que é um Top-Down Parser, uma vez que o o primeiro é um tipo do segundo. Top-Down Parsers usam a técnica de Parsing em que o símbolo inicial é expandido para todo o programa para verificar a validade do mesmo.

No caso do Recursive Descent Parser, a árvore de parsing é construida do topo para baixo começando com símbolo não-terminal inicial.

1.4 Problema e objetivos

No enunciado selecionado pela equipa, tem-se como objetivo realizar um gerador de parsers recursivos descendentes, para isso é necessário inicialmente criar uma linguagem para definir as gramáticas que devem ser independentes do contexto, e ainda um reconhecedor que seja capaz de verificar se a gramática fornecida se trata de uma construção LL(1) e, após isso, realizar o gerador o parser recursivo que deverá ser possível de realizar para qualquer GIC.

Capítulo 2

Desenvolvimento

2.1 Concessões estabelecidas

De maneira a diferenciar o tipo de produções, e até mesmo as regexs que eram necessárias incluir para gerar o ficheiro python, a equipa declarou algumas chaves para os mesmos.
Na Figura 2.1 podemos analisá-los.

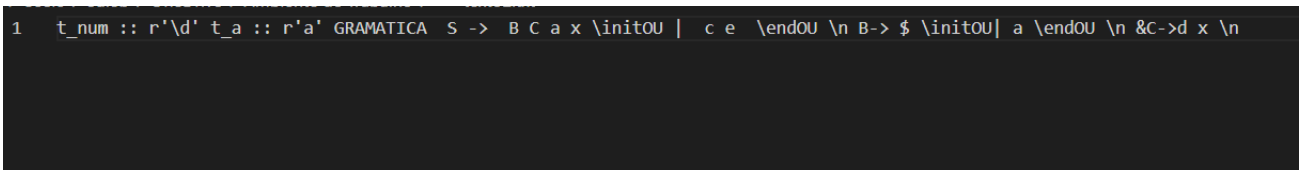

Figura 2.1: Exemplo de linguagem

Como é possível verificar, as regexs foram escritas da forma: t_ nome :: regex .
Além disso, as regexs são separadas da linguagem através da palavra GRAMATICA, assim como as produções que não contêm *"ou's"*, ou seja em que apenas existe uma produção para aquele símbolo não terminal, iniciam-se com &.
Por fim, as produções que representam um *ou* iniciam-se com \ initOU seguidos de "|"e terminam com \endOU. Todas as produçõoes finalizam-se com \n.
Esta foi a forma, que a equipa escolheu para poder distinguir em que situação se encontrava aquando do parser. Também se tornou convenção que não existem várias produções a começar pelo mesmo símbolo inicial, mas como referido para representar esses casos, utiliza-se o "ou".

2.2 Tokens

Os tokens que a equipa considerou essencial são:

- Um token para representar o separador entre regexs e a linguagem.
- Três tokens para representar as regexs.
- Um token para a nova linha.
- Um token para representar o initOU e outro para o endOU.

- Um token para representar a seta das produções.
- E por fim, os mais utilizados um para representar os símbolos terminais e outro para os símbolos não terminais. Sabendo-se que os símbolos terminais inicial por minúscula e os terminais por maiúscula.

Além disso, descreveu-se os literals para o vazio, representado por $, a barra vertical | para as produções *ou* e o & para as produções únicas.

```
def t_NEWLINE(t):
    r' \\n'
    return t

def t_END(t):
    r'\\endOU'
    return t

def t_OU(t):
    r'\\initOU'
    return t

def t_REGEX(t):
    r't_\w+'
    return t

def t_REGEX2(t):
    r'::'
    return t

def t_REGEX3(t):
    r'r\'(\\)?\w+\''
    return t

def t_GRAM(t):
    r'GRAMATICA'
    return t

def t_SIMBTERMINAIS(t):
    r'[a-z]\w* '
    return t

def t_SIMBNAOTERMINAIS(t):
    r'[A-Z]\w* '
    return t

def t_ARROW(t):
    r'\-\>'
    return t
```

Figura 2.2: Exemplo de linguagem

2.3 Parser

Para desenvolver um parser de uma gramática independente de contexto, foi necessário percorrer todas as possibilidades de construção das produções, assim como, a inserção de regexs no início do ficheiro. Permitindo existir recursividade à esquerda, assim como a possibilidade de existir mais do que um símbolo terminal na produção e/ou não terminal, para ir ao encontro da definição de gramática independente de contexto.

Não obstante, percebeu-se que era necessário de alguma forma guardar qual a sequência de ordem das produções para que mais tarde se validasse se era LL1. Chegou-se à conclusão que uma forma de fazê-lo seria incluir no projeto um dicionário onde cada key representa o lado esquerdo e o value a cadeia de símbolos do lado direito. Como permitimos a possibilidade de existir produções para o mesmo símbolo não terminal do lado esquerdo, foi necessário adicionar um iterador que quando se depara com a produção correspondente

incrementa. Ou seja, caso surja uma gramática S → A a | b, no dicionário irá constar { S: [A a], S_0: [b] }, pois esta informação é crucial para calcular o lookahead.
Posto isto, entende-se que para cada produção os símbolos lidos são inseridos numa lista, sendo que existe uma lista para a primeira produção e outra para as produções adicionais, e para cada key lida (símbolo do lado esquerdo) a mesma é adicionada ao dicionário com a lista correspondete lida até ao momento.
No exemplo da figura 2.1, o dicionário correspondente seria:

```
{'S': ['B', 'C', 'a', 'x'], 'S_0': ['c', 'e'], 'B': ['$'], 'B_0': ['a'], 'C': ['d', 'x']}
```

Figura 2.3: Exemplo do dicionário da gramática

2.4 Validação da linguagem

Tal como foi solicitado no enunciado, desenvolveu-se um algoritmo com o objetivo de perceber se a linguagem era de facto do tipo LL(1). Mas antes de realizar este raciocínio, foi necessário relembrar as condições que a tornam ou não LL(1). Através do livro de apoio bibliográfico percebemos que:
Condição 1:

$$\forall_{1 \leq i,j \leq n} : i \neq j \qquad \Rightarrow First(\alpha_i) \cap First(\alpha_j) = \emptyset$$

Figura 2.4: Condição 1

Condição 2:

$$A \Rightarrow^* \epsilon \qquad \Rightarrow First_N(A) \cap Follow(A) = \emptyset$$

Figura 2.5: Condição 2

Com base nestas definições de condições, percebeu-se que seria necessário criar três funções. A primeira é responsável pela iteração de todas as keys que estão presentes no dicionário referido anteriormente, onde verifica se o primeiro elemento da lista correspondente é um símbolo terminal ou não terminal.
Caso seja terminal, esse valor é guardado numa lista que será sempre usada para validar se o resultado das produções com essa mesma key já foi inserido ou não. Caso o resultado já esteja presente na lista, então a gramática não é LL(1), pois a primeira condição é violada.
Caso seja não terminal, invoca-se a segunda função. Função esta que recebe como input a lista a intersetar e a key que deverá procurar nas restantes produções. Nesta mesma função, também se verifica se o first dessa key é ou não um símbolo terminal, pois caso o first seja não terminal então dever-se-á invocar novamente a função *follow* para esse mesmo first.
Uma terceira validação é realizada na primeira função e na segunda, no caso de estarmos perante um $, então deve-se invocar a função que verifica a condição 2 e portanto, realiza o follow do símbolo que se segue do que deu o resultado vazio e compara-se com os firsts do último.
Caso em algum momento as condições não se verifiquem é realizado um exit do programa e imprime-se no terminal que não se está perante uma LL(1).

De seguida, serão expostos exemplos:

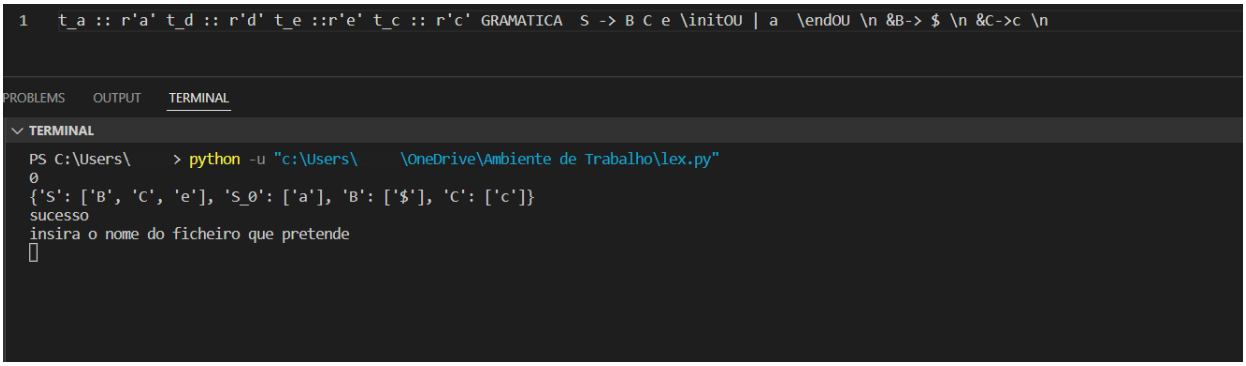

Figura 2.6: Sucesso da validação

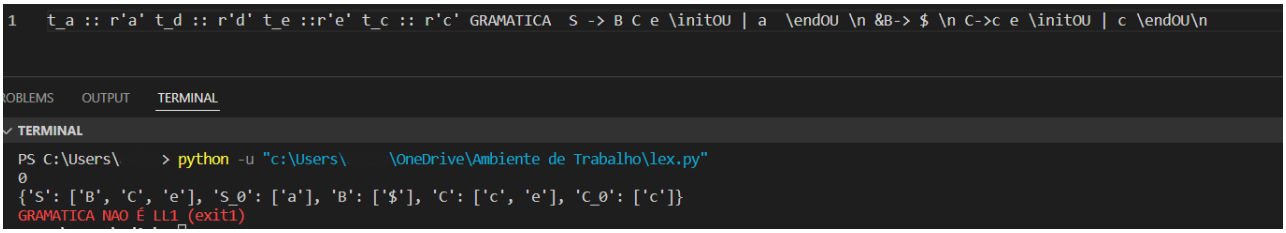

Figura 2.7: Insucesso da validação por violação da Condição 1

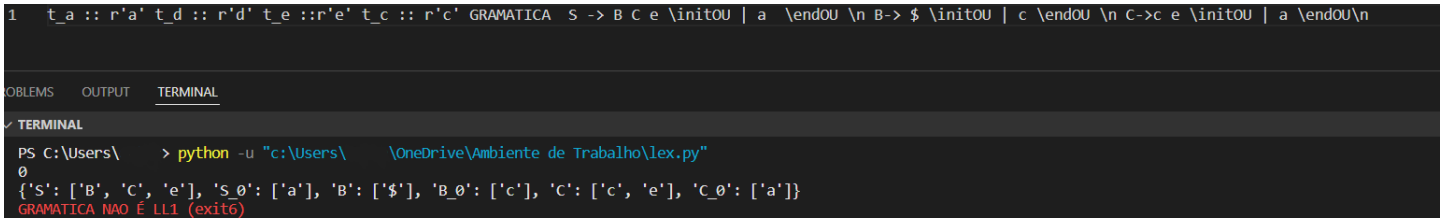

Figura 2.8: Insucesso da validação por violação da Condição 2

2.5 Gerador de Parser Recursivo descendente

Por fim, o último objetivo do projeto foi criar um ficheiro python com o código correspondente ao parser recursivo descendente da linguagem. Naturalmente, a possibilidade de criar o ficheiro só se aplica às gramáticas que foram validadas como sendo LL(1).

Inicialmente é pedido que se insira o nome do ficheiro, que deverá ser colocado sem a extensão .py . De seguida, esse ficheiro será aberto no modo de escrita e irá ser feita a leitura tanto do dicionário onde estão guardadas as produções, como a lista de tokens inseridos e a lista de regexs adicionadas.

Tanto as regexs, como os tokens são escritos no ficheiro e como convenção dever-se-á ignorar tanto os *tabs* como as novas linhas.

Posteriormente, realiza-se um ciclo que de forma inicial verifica se estamos perante uma produção inicial, ou seja a primeira produção com o dado símbolo não terminal do lado esquerdo. Caso isso se verifique, procura-se o *first* da mesma e, se for um símbolo terminal considera-se que o próximo símbolo da produção

tem que ser o mesmo, caso contrário terá que ser o *first* do símbolo não terminal. De realçar que num parser recursivo descendente, é através do primeiro símbolo do lado direito que se verifica em que regra se está perante. Por sua vez, conforme essa mesma regra, invoca-se tanto o recursivo descendente para os símbolos não terminais do lado direito, como a função recursivo termo para os símbolos terminais.

É importante referir, para cada símbolo não terminal existe uma função de recursividade que realiza o raciocínio apresentado, como uma função recursivo termo que apenas atualiza o próximo símbolo através da expressão *prox_symbol = lexer.token()*.

Por fim, as produções do mesmo símbolo não terminal do lado esquerdo, permitem adicionar mais condições possíveis para os próximos símbolos, por exemplo para A →a | b, para a função recursivo descendente *recA()* dever-se-á encontrar uma primeira condição para o próximo símbolo *a* que deverá invocar a função *recT(a)*, caso contrário o próximo símbolo será o *b*, invocando *recT(b)*, senão estamos perante um erro. De notar, que recT é correspondente ao recursivo Termo.

De seguida irá ser apresentado um exemplo.

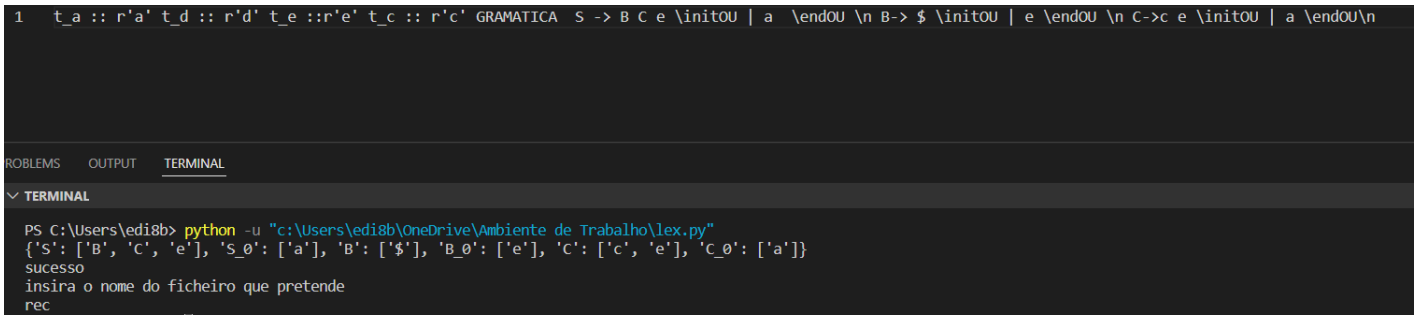

Figura 2.9: Linguagem

```
import ply.lex as lex

tokens = ["e","a","$","e","e","a"]
t_a=r'a'
t_d=r'd'
t_e=r'e'
t_c=r'c'


t_ignore = ' \t\n'

def t_error(t):
 print ( "Illegal Character:"+ t.value[0])
 t.lexer.skip(1)

lexer = lex.lex()
prox_symbol = lexer.token()


def recS():
    if prox_symbol.type() == '$':
        recB()
        recC()
        recT('e')

    elif prox_symbol=='a':
        recT('a')

    else:
        print('erro')
        exit(2)

def recB():
    if prox_symbol=='$':
        recT('$')

    elif prox_symbol=='e':
        recT('e')

    else:
        print('erro')
        exit(2)

def recC():
    if prox_symbol=='c':
        recT('c')

    elif prox_symbol=='a':
        recT('a')

    else:
        print('erro')
        exit(2)


def recT(t):
    global prox_symbol
    if prox_symbol=='t':
        prox_symbol==lexer.token()
    elif t=='$':
        print('Sucesso')
    else:
        print(' erro a reconhecer t')
        exit(1)



recS()
```

Figura 2.10: Ficheiro rec.py

Capítulo 3

Conclusão

Em síntese, a equipa acredita ter resolvido todos os objetivos a que se propôs, sendo que reconhece que existem aspetos que deveriam, num trabalho futuro, ser melhorados.

Dentro destes aspetos a progredir, tem-se a reutilização de código, pois apesar de ter sido algo que foi tido em conta, acredita-se que, no futuro, deveria ser aperfeiçoado. Além disso, algo que a equipa desejaria ter executado seria aumentar a flexibilidade do parser, permitindo retirar os indicadores adicionados às produções. Na medida em que se pretendia realizar uma gramática mais genérica e mais próxima à realidade, sem a inserção dos símbolos adicionados pela equipa.