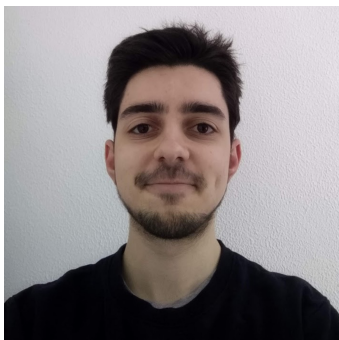


Universidade do Minho
Departamento de Informática

Gerador de Parsers LL(1) Recursivos Descendentes TP2 Grupo 42

15 de Maio, 2021



Alexandre Flores
(a93220)



Mariana Rodrigues
(a93294)



Matilde Bravo
(a93246)

Conteúdo

1	Introdução	3
2	Descrição do Enunciado Proposto	4
3	Gramática Desenvolvida	5
3.1	Ficheiro de Input	5
3.1.1	Exemplo de um ficheiro de Input	6
3.2	Linguagem da Gramática	7
4	Implementação	8
4.1	Parser recursivo descente	8
4.2	Construção do código do parser recursivo descente	9
5	Exemplos de Utilização	10
5.1	Parser que reconhece uma Abin	10
5.2	Parser de uma Calculadora	12
5.3	Tentativa de execução de uma gramática que não é LL(1)	14
6	Conclusão	15

Capítulo 1

Introdução

No âmbito da unidade curricular de *Processamento de linguagens* foram propostos diversos enunciados, de entre os quais selecionamos o terceiro projeto. Este corresponde ao tema *Gerador de Parsers LL(1) Recursivos Descendentes*.

Capítulo 2

Descrição do Enunciado Proposto

Uma vez que a criação de um parser recursivo descendente é um processo bastante repetitivo, é proposta a automatização desse processo. O programa a implementar deverá ser capaz de receber uma gramática independente de contexto, representada numa dada linguagem que consiga descreve-la, e, de seguida, verificar se esta é $LL(1)$. Caso verifique esta condição, deverá gerar código em *python* que implemente um parser recursivo descendente da gramática reconhecida.

Capítulo 3

Gramática Desenvolvida

3.1 Ficheiro de Input

Como já mencionado anteriormente, é necessário reconhecer uma dada gramática independente de contexto. Para tal, começamos por definir como seria a estrutura do ficheiro de input, o qual contém a gramática dada.

Estabelecemos que esta teria que seguir este esquema:

- **Literals:** Aqui estabelecemos quais são os literais válidos da gramática que iremos dar. Para tal, estipulamos que terá que ser dado da seguinte forma:

```
Literals = [ 'itens' ]
```

Corresponde a uma lista de palavras, entre plicas, separadas por espaços. É importante referir que 'itens' poderá não existir, caso a gramática não contenha *literals*.

Seguem-se alguns exemplos de utilização:

```
Literals = [ ]
```

```
Literals = [ '.' ]
```

```
Literals = [ '(' ')' '.' ]
```

- **Tokens:** Da mesma forma que fizemos para os literais, temos que ser capazes de informar quais são os tokens válidos da gramática. Neste caso, a cada token tem que estar associada uma expressão regular. Consequentemente, seria dado desta forma:

```
Tokens = [ 'name' = 'regex' ]
```

Seguem-se alguns exemplos de utilização:

```
Tokens = [ ]
```

```
Tokens = [ 'num' = '\d+' ]
```

```
Tokens = [ 'num' = '\d+' 'name' = '\w+' ]
```

- **Gramática:** Esta é a parte mais crucial, é nesta fase que será definida toda a gramática. A este ponto, será nos dado um conjunto de expressões.

Com isto, tivemos que definir o que seria uma expressão. Esta terá que conter um *id* que a identifique, seguido de pelo menos uma condição. Para informar que a expressão chegou ao fim, terá que ser dada uma ','.

Exemplo de uma expressão apenas com uma condição:

```
Z -> Abin '.';
;
```

Exemplo de duas expressões com diversas condições:

```
Aux -> 'num' Abin Abin;
      | ;
;

Fator -> 'id';
        | 'num';
        | '(' Exp ')';
;
;
```

- **End:** Para ser possível sabermos que não existem mais expressões a serem reconhecidas, o ficheiro de input terá que terminar com um '.' .

3.1.1 Exemplo de um ficheiro de Input

Segue-se um exemplo de um ficheiro de input:

```
Literals = [ '(' ')' '.' ]
Tokens = [ 'num'='\d+' ]

Z -> Abin '.';
;

Abin -> '(' Aux ')';
;

Aux -> 'num' Abin Abin;
      | ;
;
.
```

3.2 Linguagem da Gramática

Após termos definido como seria a estrutura do ficheiro de input, foi necessário desenvolver uma gramática que conseguisse reconhecer a estrutura dada.

Primeiramente, começamos por definir os *literals* e os *tokens* que teríamos que capturar:

```
EQUALS = '='
ID = r'[A-Za-z]\w*'
LITERAL = r'\ '[^\\']+\''
SETA = r'-'>'
END = r';'
END_PARSER = r'.'
SEP = r'\|'
FCLOSE = r'\s*\]'
ALITERALS = r'Literals\s*=\s*\[s*'
ATOKENS = r'Tokens\s*=\s*\[s*'
```

Tendo isto, passamos à fase seguinte:

```
ValidsLiterals -> 'ALITERALS' ListLiterals 'FCLOSE'      {'ALITERALS'}

ListLiterals -> 'LITERAL' ListLiterals                    {'LITERAL'}
               |                                           {'FCLOSE'}

ValidsTokens -> 'ATOKENS' ListTokens 'FCLOSE'             {'ATOKENS'}

ListTokens -> 'LITERAL' 'EQUAL' 'LITERAL' ListTokens      {'LITERAL'}
               |                                           {'FCLOSE'}

Grammar -> Exp Exps                                       {'ID'}

Exps -> Exp Exps                                         {'ID'}
      | 'END_PARSER'                                    {'END_PARSER'}

Exp -> 'ID' 'SETA' Conduction Conductions                {'ID'}

Tokens -> 'ID' Tokens                                    {'ID'}
        | 'LITERAL' Tokens                              {'LITERAL'}
        | 'END'                                          {'END'}

Conduction -> 'ID' Tokens                                {'ID'}
             | 'LITERAL' Tokens                        {'LITERAL'}
             | 'END'                                    {'END'}

Conductions -> 'SEP' Tokens Conductions                  {'SEP'}
               | 'END'                                    {'END'}

Start -> ValidsLiterals ValidsTokens Grammar             {'ALITERALS'}
```

Capítulo 4

Implementação

Para uma melhor organização deste trabalho, decidimos separa-lo em diferentes ficheiros:

- **LL1.py** : Ficheiro responsável por guardar todas as classes necessárias
- **parserGrammar.py** : Ficheiro responsável pelo reconhecimento e construção da gramática dada
- **buildParser.py** : Ficheiro responsável pela geração do código do parser recursivo descendente de uma dada gramática
- **main.py** : Ficheiro principal que executa e chama os ficheiros anteriores

4.1 Parser recursivo descente

Começamos inicialmente por desenvolver o lexer que ficará responsável por reconhecer o ficheiro de input que será dado.

De seguida, foi desenvolvido um parser que segue a lógica da gramática desenvolvida anteriormente. Uma vez que é necessário armazenar e guardar a informação que formos reconhecendo, foi crucial criarmos as seguintes classes:

```
class Condictio:
    def __init__(self, tokens: list = [], symbols: list = [], last_symbol: str = ''):
        self.rules = tokens          # lista de todos os tokens da condição
        self.symbols = symbols       # lookahead da condição
        self.last_symbol = last_symbol # último símbolo da condição

class Exp:
    def __init__(self, name, condictions):
        self.name = name             # nome da expressão
        self.condictions = condictions # todas as condições dessa expressão
        self.symbols = []           # o lookahead da expressão

class LL1:
    def __init__(self, exps = {}):
        self.exps = exps             # Todas as expressões
        self.rules = []              # Todas as regras
        self.tokens = {}             # Todos os tokens
        self.literals = []           # Todos os literais
```

Após termos lido, reconhecido e convertido a gramática lida para a estrutura de dados definida em cima, falta-nos só verificar se a gramática obtida é LL1.

4.2 Construção do código do parser recursivo descente

Tendo realizado todo o trabalho anterior, nesta fase já temos a estrutura da gramática. Consequentemente, só resta construir o ficheiro de código do parser recursivo descente.

É de salientar que antes desta mesma construção, a gramática que reconhecida é verificada. Isto é, vamos comprovar que a mesma é **LL(1)**, pois caso não o seja, não se desenvolve o código e o programa termina.

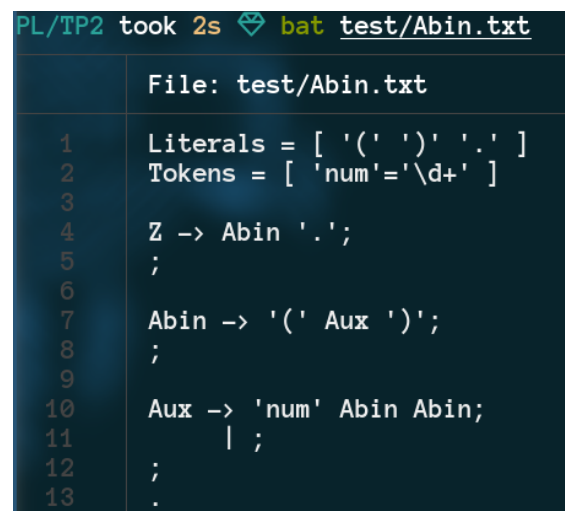
Supondo que de uma gramática **LL(1)** se trata, gera-se o código. Parte desse código faz parte de um *template* já definido por nós enquanto que o restante código irá depender da gramática reconhecida.

Capítulo 5

Exemplos de Utilização

As seguintes *screenshots* ilustram a utilização do programa.

5.1 Parser que reconhece uma Abin



```
PL/TP2 took 2s bat test/Abin.txt
File: test/Abin.txt
1 Literals = [ '(' ')' '.' ]
2 Tokens = [ 'num'='\d+' ]
3
4 Z -> Abin '.';
5 ;
6
7 Abin -> '(' Aux ')';
8 ;
9
10 Aux -> 'num' Abin Abin;
11 | ;
12 ;
13 .
```

Figura 5.1: Ficheiro de Input

Com isto basta-nos correr o comando:

```
cat test/Abin.txt | python main.py > Abin.py
```

Gerando assim um ficheiro chamado Abin, com o parser recursivo descendente pretendido.

```
File: Abin.py

#!/usr/bin/env python3

import ply.lex as lex
import ply.yacc as yacc

literals = ['(', ')', '.', '']
tokens = ['num']

def t_num(t):
    r'\d+'
    return t

def t_new_line(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

t_ignore = "\t "

def t_error(t):
    print("Erro léxico no token '%s' % t.value[0])
    t.lexer.skip(1)

lexer = lex.lex()

def p_Z_0(p):
    "Z : Abin '.'"

def p_Abin_0(p):
    "Abin : '(' Aux ')"

def p_Aux_0(p):
    "Aux : num Abin Abin"

def p_Aux_1(p):
    "Aux : "

def p_error(p):
    print("Erro sintático: ", p)
    parser.success = False

# Build the parser
parser = yacc.yacc()
parser.success = True

# Read line from input and parse it
import sys
for linha in sys.stdin:
    parser.success = True
    parser.parse(linha)
    if parser.success:
        print("Success")
```

Figura 5.2: Ficheiro de output

```

PL/TP2 python Abin.py
Generating LALR tables
().
Success
(112 (12 (12 () ()) ()) ( 12 () ())).
Success
(112 (12 (12 () ()) ()) ( 12 () ( 12 () () ))).
Success
(112 ((((((((((.
Erro sintatico:, LexToken((, '(', 4, 6)

```

Figura 5.3: Parser Abin

5.2 Parser de uma Calculadora

```

File: test/Calc.txt

Literals = [ '.', '+', '*', '/', '(', ')', '-' ]
Tokens = [ 'num'='\d+', 'id'='\w+' ]

Z -> Exp '.';
;

Exp -> Termo Exp2;
;

Exp2 -> '+' Exp;
      | '-' Exp;
      | ;
;

Termo -> Fator Termo2;
;

Termo2 -> '*' Termo;
        | '/' Termo;
        | ;
;

Fator -> 'id';
        | 'num';
        | '(' Exp ')';
;
.

```

Figura 5.4: Ficheiro de Input

Correndo:

```
cat test/Calc.txt | python main.py > calculadora.py
```

Obtemos o ficheiro de output:

```
literals = ['.', '+', '*', '/', '(', ')', '-']
tokens = ['num', 'id']

def t_num(t):
    r'\d+'
    return t
def t_id(t):
    r'\w+'
    return t

def t_new_line(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

t_ignore = "\t "

def t_error(t):
    print("Erro léxico no token '%s' % t.value[0])
    t.lexer.skip(1)

lexer = lex.lex()

def p_Z_0(p):
    "Z : Exp '.'"

def p_Exp_0(p):
    "Exp : Termo Exp2"

def p_Exp2_0(p):
    "Exp2 : '+' Exp"
```

Figura 5.5: Excerto do Ficheiro de output

```
PL/TP2 took 33s python calculadora.py
Generating LALR tables
2+3.
Success
2.
Success
numero + 2.
Success
numero + ( (2*3) / 8) * 2.
Success
().
Erro sintatico:, LexToken(),')',5,1)
numero + -
Erro sintatico:, LexToken(-,'-',6,9)
```

Figura 5.6: Parser da Calculadora

5.3 Tentativa de execução de uma gramática que não é LL(1)

```
PL/TP2 bat buganco.txt
File: buganco.txt
1  Literals = ['.' ]
2  Tokens = [ ]
3
4  Z -> Exp '.' ;
5  ;
6
7  Exp -> Exp2;
8  ;
9
10 Exp2 -> Exp;
11 ;
12 .

on main [🤖]
PL/TP2 cat buganco.txt | python main.py
The grammar isn't a LL1 parser
```

Figura 5.7: Tentativa de execução de uma gramática que não é LL(1)

Capítulo 6

Conclusão

Este projeto permitiu aprofundar os conhecimentos adquiridos nas aulas de Processamento de Linguagens. Podendo assim, compreender melhor o funcionamento dos *parsers* recursivos descendentes, bem como a forma como estes podem ser gerados.

Em conclusão, estamos satisfeitos com o trabalho realizado e achamos que este cumpre os objetivos estabelecidos no enunciado.