Processamento de Linguagens
Engenharia Informática (3º ano)
Projeto Final

23 de Março de 2023

Dispõe de **7 semanas** para desenvolver este trabalho, a entrega deverá ser feita até à meia noite de **14 de Maio**.

1 Objectivos e Organização

Este trabalho prático tem como principais objectivos:

- aumentar a experiência em engenharia de linguagens e em programação generativa (gramatical), reforçando a capacidade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT);
- desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora;
- desenvolver um compilador gerando código para um objetivo específico;
- utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o Yacc, versão PLY do Python, completado pelo gerador de analisadores léxicos Lex, também versão PLY do Python.

Na resolução dos trabalhos práticos desta UC, aprecia-se a imaginação/criatividade dos grupos em todo o processo de desenvolvimento!

Deve entregar a sua solução até Domingo dia 14 de Maio. O ficheiro com o relatório e a solução deve ter o nome 'pl2023-projeto-grNN', em que NN corresponderá ao número de grupo. O número de grupo será ou foi atribuído no registo das equipes do projeto.

Cada grupo, **é livre** para escolher qual o enunciado que pretende desenvolver.

A submissão deverá ser feita por email com os seguintes dados:

to: jcr@di.uminho.pt

subject: PL2023::grNN::Projeto::Enunciado

body: Colocar um ZIP com os ficheiros do Projeto: relatório, código desenvolvido e datasets de teste.

Em cima, "Enunciado"é um dos valores: Pandoc, Ply-Simple, RecDesc, TDTabela.

Na defesa, a realizar na semana de 16 a 20 de Maio, o programa desenvolvido será apresentado aos membros da equipa docente, totalmente pronto e a funcionar (acompanhado do respectivo relatório de desenvolvimento) e será defendido por todos os elementos do grupo, em data e hora a marcar. O relatório a elaborar, deve ser claro e, além do respectivo enunciado, da descrição do problema, das decisões que lideraram o desenho da solução e sua implementação, deverá conter exemplos de utilização (textos fontes diversos e respectivo resultado produzido). Como é de tradição, o relatório será escrito em LaTeX.

2 Enunciados

2.1 Linguagem de templates (inspirada nos templates Pandoc)

Considere a linguagem de definição de templates do Pandoc:
(ver exemplos e detalhes em `https://pandoc.org/MANUAL.html#templates`)

- Escrever a gramática (comece por um subconjunto e vá sucessivamento enriquecendo).
- Construir o analisador léxico.
- Construir o parser que:
- hipótese 1: dado um template T1 gerar uma função python `expand_T1` que recebendo um dicionário dê um "texto final"
- hipótese 2: escreve um programa que dado um template T1 e um dicionário produza o "texto final"
- hipótese 3: escreve um programa que dado um ficheiro template (T1) e um ficheiro YAML (dicionário) produza o "texto final"

2.2 Gerador de Parsers LL(1) Recursivos Descendentes

Apesar de algumas limitações das gramáticas LL(1) um parser recursivo descendente tem a vantagem de poder ser implementado em qualquer linguagem de programação que suporte recursividade.

A tarefa de escrever um parser recursivo descendente é bastante repetitiva e segue um padrão algorítmico, padrão esse que se quer captar, nesta proposta, na geração de código.

Assim, neste projeto deverás realizar as seguintes tarefas:

- Criar uma linguagem para descrição de gramáticas independentes de contexto;
- Criar um reconhecedor para essa linguagem que:
  1. Verifique se a linguagem é LL(1);
  2. Caso seja, gere o código Python que implementa o parser recursivo descendente;
  3. Como extra, poderás pensar em arranjar uma maneira de juntar ações semânticas ao parser gerado.

2.3 Gerador de Parsers LL(1) Dirigidos por Tabela

A tarefa de escrever um parser Top Down dirigido por tabela é bastante repetitiva e segue um padrão algorítmico, padrão esse que se quer captar, nesta proposta, na geração de código.

Assim, neste projeto deverás realizar as seguintes tarefas:

- Criar uma linguagem para descrição de gramáticas independentes de contexto;
- Criar um reconhecedor para essa linguagem que:
  1. Verifique se a linguagem é LL(1);
  2. Caso seja, constrói o parser Top-Down dirigido por tabela;
  3. Como extra, poderás pensar em arranjar uma maneira de juntar ações semânticas ao parser gerado.

2.4 Reverse engineering de um dicionário

Em `natura.di.uminho.pt/~jj/pl-20/TP2` está o texto de um dicionário EN-PT. Considere o seguinte extrato de um dicionário de negócio e finanças EN-PT:

```
dispute:
 labour -                conflito (m) trabalhista
dissolution              dissolução (f)
distribution:
 - costs                 custos de distribuição
 channels of -           canais (mpl) de distribuição
 physical - management   controle (m) da distribuição física
distributor              distribuidor (m), atacadista (m)
diversification:         diversificação (f)
 - strategy              estratégia (f) de diversificação
 product -               diversificação (f) de produtos
```

Este ficheiro de partida foi resultado da conversão automática PDF-TXT e pretendemos fazer *reverse engineering* de modo a tirar o máximo partido da informação.

- Escreva uma gramática que cubra tanto quanto possivel as entradas do dicionário.
- Construa um processador (flex, yacc) que converta o ficheiro num formato mais tratável. Ver abaixo um exemplo de saída pretendida.
- Deve haver casos de erros de formato, incoerências: se for possível, avise da linha do erro, e continue o processamento.

```
EN labour dispute
+base dispute
PT conflito (m) trabalhista

EN dissolution
PT dissolução (f)
```

```
EN distribution costs
+base distribution:
PT custos de distribuição

EN channels of distribution
+base distribution:
PT canais (mpl) de distribuição

EN physical distribution management
+base distribution:
PT controle (m) da distribuição física

EN distributor
PT distribuidor (m)
PT atacadista (m)

EN diversification
PT diversificação (f)

EN diversification strategy
+base diversification
PT estratégia (f) de diversificação

EN product diversification
+base diversification
PT diversificação (f) de produtos
```

2.5 conversor de Pug- - para HTML

Construa um conversor que aceite um subconjunto da linguagem Pug e gere o HTML correspondente (defina claramente a parte coberta).

Exemplo da notação Pug:

```
html(lang="en")
  head
    title= pageTitle
    script(type='text/javascript').
      if (foo) bar(1 + 5)
  body
    h1 Pug - node template engine
    #container.col
      if youAreUsingPug
        p You are amazing
      else
        p Get on it!
      p.
        Pug is a terse and simple templating language with a
        strong focus on performance and powerful features
```

HTML correspondente:

```
<html lang="en">
  <head><title></title>
      <script type="text/javascript">
        if (foo) bar(1 + 5)</script>
  </head>
  <body>
     <h1>Pug - node template engine</h1>
     <div class="col" id="container">
         <p>Get on it!</p>
         <p>Pug is a terse and simple templating language with a
strong focus on performance and powerful features</p>
```

```
    </div>
  </body>
</html>
```

Ver também a documentação

```
https://pugjs.org/language/tags.html
https://pugjs.org/language/plain-text.html
https://pugjs.org/language/attributes.html
```

2.6 Conversor toml-json

Ver detalhes em: `https://github.com/toml-lang/toml`

A linguagem *toml* permite uma fácil definição de estruturas complexas (dicionários generalizados) frequentmente usados em ficheiros de configuração e em vários outros domínios de modo análogo ao JSON e ao YAML

Considere o sequinte exemplo:

```
title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
date = 2010-04-23
time = 21:30:00

[database]
server = "192.168.1.1"
ports = [ 8001, 8001, 8002 ]
connection_max = 5000
enabled = true

[servers]
[servers.alpha]
  ip = "10.0.0.1"
  dc = "eqdc10"

[servers.beta]
  ip = "10.0.0.2"
  dc = "eqdc10"

# Line breaks are OK when inside arrays
hosts = [
  "alpha",
  "omega"
]
```

Pretende-se construir uma ferramenta (Flex,Yacc) que converta um subconjunto de Toml para JSON.

2.7 Conversor de GEDCOM

A Genealogia tem desde sempre atraído muitas pessoas. Investigadores, historiadores, curiosas, pessoas comuns que por um motivo ou outro precisam de construir a sua árvore genealógica.

Ao longo dos últimos anos, o formato GEDCOM ganhou popularidade e uma comunidade de utilizadores que faz com que seja o formato de eleição para a transmissão e intercâmbio de informação nesta área.

Na internet é possível encontrar muitos recursos e informação sobre este formato.

Neste projeto, pretende-se que o grupo de trabalho desenvolva um conversor de GEDCOM 5.5 para XML/HTML.

Sugerem-se as seguintes etapas para o desenvolvimento deste projeto:

1. Estudar o formato, alguns casos de estudo e a sua estrutura. Sugerem-se os seguintes links:

   `http://homepages.rootsweb.com/~pmcbride/gedcom/55gcch1.htm` – Contem uma descrição do formato juntamente com um rascunho da gramática;

`http://www4.di.uminho.pt/~jcr/AULAS/didac/ontologias/BibleFamilyTree.ged.txt` – Dataset com as famílias da Bíblia;

`http://www4.di.uminho.pt/~jcr/AULAS/didac/ontologias/GreekGods.ged.txt` – Dataset com as famílias da mitologia grega;

`http://www4.di.uminho.pt/~jcr/AULAS/didac/ontologias/RomanGods.ged.txt` – Dataset com as famílias da mitologia romana;

`http://www4.di.uminho.pt/~jcr/AULAS/didac/ontologias/Royal92.ged.txt` – Dataset com as famílias da realeza europeia.

2. Escrever uma gramática para o GEDCOM 5.5;
3. Implementar a gramática em Lex e Yacc;
4. Definir o formato de saída (pedir orientação ao professor), apresenta-se um exemplo dum possível formato de saída: `http://www4.di.uminho.pt/~jcr/AULAS/didac/ontologias/royalFamilies.xml`

```
<genoa>
  <pessoa>
    <id>I1</id>
    <nome>Victoria Hanover</nome>
    <título>Queen of England</título>
    <sexo>F</sexo>
    <dataNasc>1819-05-24</dataNasc>
    <localNasc>Kensington,Palace,London,England</localNasc>
    <dataÓbito>1901-01-22</dataÓbito>
    <pai>I133</pai>
    <mae>I138</mae>
  </pessoa>
  ...
</genoa>
```

5. Acrescentar ao parser as ações geradoras do formato final;
6. Ir discutindo com o docente as opções tomadas...

2.8 Extensão funcional para Python

A programação funcional é um paradigma de programação que se foca no uso de funções puras, dados imutáveis e funções de ordem superior. Apesar de já existir há várias décadas, tem ganhado popularidade recentemente graças à sua expressividade, simplicidade e utilidade em computação paralela e sistemas distribuídos. As linguagens imperativas e/ou orientadas a objetos, como Python ou Java, têm adotado alguns princípios da programação funcional, estando mesmo assim bastante aquém daquilo que é possível fazer numa linguagem funcional.

O objetivo deste projeto é a criação de uma extensão funcional para a linguagem Python. Por outras palavras, devem criar uma linguagem de programação nova, exatamente igual a Python, mas onde também é possível definir funções "funcionais". Estas funções podem estar definidas num comentário *multi-line*, para estarem bem identificadas, ou soltas algures pelo ficheiro, fora de comentários. Cabe a cada grupo decidir que método prefere.

Um exemplo de um programa escrito com esta extensão poderá assemelhar-se ao seguinte:

```
"""FPYTHON

deff mais_um(x):
    x + 1

deff sum([]):
    0
deff sum([h | t]):
    h + sum(t)

"""

x = 4
y = f_mais_um_(x)
```

```
print(y)

l = [1,2,3,4,5]
sum_l = f_sum_(l)
print(sum_l)
```

Temos então um comentário, identificado com o nome FPYTHON (podem substituir pelo nome que derem à vossa extensão/linguagem, ou outra coisa qualquer), no qual estão definidas as nossas funções funcionais. Estas podem ser identificadas também pela *keyword* "deff". Quando queremos chamar uma destas funções no nosso programa principal Python, usamos o prefixo "f_" e o sufixo "_" para distinguir uma função funcional de uma função "normal". Esta sintaxe pode ser alterada pelo grupo, se pertinente, tal como a sintaxe das funções. Apenas precisam de se certificar que a vossa extensão permite fazer o seguinte:

- Definir funções;
- Realizar operações aritméticas, como adição, subtração, etc., dentro das mesmas;
- Usar tipos de dados inteiros, reais, booleanos e listas, pelo menos;
- Garantir a imutabilidade dos dados, por outras palavras, nunca alterar as variáveis que uma função recebe;
- Ser pura, isto é, para o mesmo input devolver sempre o mesmo output (isto significa que não pode suportar funções de input/output, devem fazer estas operações sempre no código Python);
- Permitir instruções condicionais do tipo `if ... then ... else ...`, com esta ou outra sintaxe (o bloco `else` é obrigatório pois as funções funcionais devem sempre produzir um valor);
- Permitir recursividade;
- Permitir fazer *pattern matching* nos argumentos das funções, tal como é demonstrado no exemplo acima, na função 'sum'. Deve ser pelo menos possível definir uma lista como um par cabeça-cauda ou uma lista vazia e verificar se um inteiro possui um certo valor, para poder usar funções recursivas com estes dois tipos.

Podem ainda implementar outras funcionalidades extra, como *pattern matching* no corpo das funções, composição de funções (como `(h . g . f)(x)` em Haskell ou `x |> f() |> g() |> h()` em Elixir), funções de ordem superior, etc. O céu é o limite!

Podem-se basear em linguagens de programação funcionais existentes. Seguem-se dois exemplos de funções que calculam a soma dos elementos ímpares de uma lista, escritas em Haskell e em Elixir.

```
soma_impares [] = 0
soma_impares (h:t) = if not e_par then soma t else h + soma t
    where e_par = h `mod` 2 == 0
```

```
def soma_impares([]) do
    0
end
def soma_impares([ h | t ]) do
    e_par = rem(h, 2) == 0
    if not e_par do
        soma t
    else
        h + soma t
    end
end
```

Quanto à estrutura das funções, há duas opções. As funções poderão ter apenas uma instrução ou bloco no seu corpo, sendo que variáveis auxiliares poderão ser declaradas num bloco separado do corpo da função. Alternativamente, as funções podem ter várias instruções no seu corpo, sendo que o valor produzido pela função corresponde ao valor da última instrução ou bloco. Os exemplos acima demonstram estas duas implementações.

O compilador deverá converter um programa escrito com esta extensão para um programa Python puro (sem apagar o ficheiro original), que possa ser executado diretamente com o comando `python`. O código que já está em Python pode ficar intacto, não precisam de definir um parser para Python, apenas para a extensão. Como Python não é uma linguagem funcional, ao fazer a conversão, devem preocupar-se com o *pattern matching*, pois não é possível fazê-lo nativamente em Python. Um exemplo de como podem lidar com isso é o seguinte:

```
def f_sum_(arg0):
    if len(arg0) == 0:
        return 0
    if len(arg0) >= 1:
        h = arg0[0]
        t = arg0[1:]
        return h + f_sum_(t)
    else:
        raise ValueError
```

Esta é uma definição em Python puro da função funcional `sum` definida no exemplo acima. Substituímos o *pattern matching* por condições e atribuições. Caso nenhuma das condições seja verdadeira, "levantamos"um erro durante a execução. Tal como anteriormente, podem arranjar outras formas de lidar com estes erros, ou usar outra sintaxe.

Segue-se outro exemplo de um programa escrito com a extensão "funcional"e da sua possível conversão para Python:

```
"""FPYTHON

deff fib(0):
    0
deff fib(1):
    1
deff fib(n):
    fib(n-1) + fib(n-2)
"""

print([f_fib_(n) for n in range(12)])
```

```
def f_fib_(arg0):
    if arg0 == 0:
        return 0
    if arg0 == 1:
        return 1
    n = arg0
    return f_fib_(n - 1) + f_fib_(n - 2)

print([f_fib_(n) for n in range(12)])
```

É possível que uma função Python seja menos eficiente do que a sua função funcional equivalente. Não há problema nenhum com isso, aqui os objetivos são que a extensão funcional facilite a escrita de código e que este produza o resultado esperado, a *performance* não é importante.

Uma última observação: lidar com a indentação da mesma forma que o Python é extremamente difícil, e vai bastante para lá daquilo que é exigido deste trabalho prático. Não é recomendável que tentem emular essa funcionalidade. O conselho da equipa docente é que definam uma indentação constante para cada função (4 espaços, por exemplo, e qualquer coisa diferente é considerado inválido) ou simplifiquem ainda mais e usem chavetas ou uma sintaxe semelhante à de Elixir, com uma *keyword* `end` no fim de cada bloco.

Bom trabalho e boa sorte

A equipe docente:

José Carlos Ramalho
José João Almeida
Pedro Rangel Henriques
Tiago Baptista
Sofia Santos