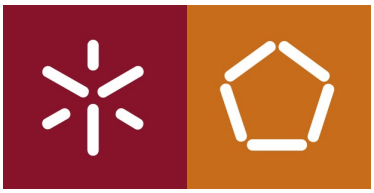

Universidade do Minho
Licenciatura em Engenharia Informática
3ºano - 2º Semestre

Processamento de Linguagens

Projeto final - Extensão funcional para Python

A85635 - André Nunes

20 de maio de 2023

Conteúdo

Breve descrição do enunciado

No âmbito da unidade curricular de Processamento de Linguagens foram-nos apresentados vários enunciados entre os quais teriamos que escolher apenas um. Depois de alguma deliberação o nosso grupo decidiu escolher o enunciado "Extensão funcional para Python". O objetivo deste projeto é a criação de uma extensão funcional para a linguagem Python. Por outras palavras, devemos 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.

Semântica e sintaxe

Nesse sentido o grupo começou por definir as regras de sintaxe e semântica da linguagem a construir. Como pedido no enunciado podemos intercalar Python puro com os nossos blocos de FPYTHON. Cada bloco deve seguir a seguinte estrutura:

"""FPYTHON <lines-of-code>"""

Dentro de cada um desses blocos o utilizador pode fazer comentários, escrevendo qualquer frase precedida dos caracteres '–', pode fazer qualquer tipo de declaração usando o caracter '=', escrever uma frase condicional com a sintaxe "if...then...else...", e declarar funcões. De notar que esta extensão aceita qualquer tipo que o Python aceite, uma vez que funciona como um tradutor para Python. No contexto da declaração de funções a sintaxe a seguir deve ser:

```
def <name-of-the-function>():
    <instruction>;
    ...
end
```

Cada instrução declarada dentro das funções tem também uma sintaxe especial, muito parecida com a linguagem Haskell:

<pattern> = <operation>;

Uma instrução é composta por um padrão e a respetiva operação que deve ser executada sobre esse padrão. Essas operações podem ser expressões aritméticas, lógicas, condicionais ou recursivas. Todas estas operações seguem uma sintaxe muito similar com a linguagem Haskell.

Temos portanto como exemplo de um bloco de FPYTHON:

```
"""FPYTHON
--comment
y = [x*2 | x <- [1..20], x>12]

def sum:
    [] = 0;
    [a] = a;
    (h:t) = h + sum(t);
end
"""
```

Gramática

De seguida o grupo definiu uma gramática de acordo com a semântica e sintaxe estabelecidas. Esta gramática cobre todas as ações definidas para os blocos FPYTHON.

```
program : INIT frases END

comments : COMMENT conteudo

frases : comments
       | decl
       | cond
       | func
       | frases func
       | frases cond
       | frases comments
       | frases decl

func : DEF insts EDEF

insts : inst
      | insts inst

inst : INST exp EINST
     | INST logic EINST
     | INST cond EINST
     | INST rec EINST

rec : exp REC exp

decl : exp DECL exp
     | exp DECL logic
```

```
exp : exp PLUS exp
    | exp MINUS exp
    | exp TIMES exp
    | exp DIVIDE exp
    | exp MOD exp
    | conteudo

logic : exp EQ exp
      | exp GT exp
      | exp LT exp
      | exp GTE exp
      | exp LTE exp
      | exp AND exp
      | exp OR exp
      | NOT logic

cond : IF logic THEN exp ELSE exp

conteudo : NUM
         | STRING
         | ID
         | BOOL
         | TUPLE
         | lista
         | call

call : ID LP exp RP

lista : LISTA
```

Como podemos perceber pela gramática definida, cada bloco de FPYTHON é definido por um Token INIT, uma ou mais frases e um token END. Estas frases podem ser comentários, declarações, condicionais ou funções. Cada comentário é definido com um token COMMENT e um qualquer conteúdo. Cada declaração é definida por uma expressão, um token DECL e outra expressão ou por uma expressão, um token DECL e uma expressão lógica. Cada condicional é definida por um token IF, uma expressão lógica, um token THEN, uma expressão, um token ELSE e outra expressão. Relativamente à declaração de funções temos que cada funçao é definida por um Token DEF, uma ou mais instruções e um token EDEF. Estas intruções podem ser expressões, expressões lógicas, condicionais ou recursividas inseridas entre os tokens INST e EINST. O token INST indica o padrão que vai ser manipulado pela instrução e o EINST indica o final de uma instrução. As expressões são apenas interações aritméticas entre conteúdos enquanto que as lógicas são apenas interações lógicas entre conteúdos. Como conteúdo temos os tipos unitários aceites pela liguagem, no entanto chamadas

de funçoes e listas têm uma sintaxe expecial. As chamadas de funções são constituídas pelo nome da função e uma expressão dentro de parênteses. A listas são apenas listas normais ou listas por compreensão definidas exatamente do mesmo modo que na linguagem Haskell.

Arquitetura da solução

De modo a atender às necessidades do projeto o grupo decidiu dividir o código em dois ficheiros: "lexer.py"e "parser_FP.py". O primeiro é responsável por ler o ficheiro de input e identificar todos os tokens lidos de modo a que o segundo possa traduzir todos os blocos de FPYTHON e produzir um novo ficheiro de Python puro.

Lexer

O nosso Lexer compreendido no ficheiro "lexer.py"garante que todos os blocos de FPYTHON são tokenized.

Tokens

Os tokens necessários para abranger todas as funcionalidades do projeto são os seguintes:

```
tokens = ('NUM', 'STRING', 'DECL', 'INIT', 'END',
          'COMMENT', 'LISTA', 'BOOL', 'TUPLE',
          'PLUS', 'MINUS', 'TIMES', 'DIVIDE',
          'MOD', 'ID', 'DEF', 'EDEF', 'INST',
          'EINST', 'REC', 'EQ', 'GT', 'LT',
          'GTE', 'LTE', 'AND', 'OR', 'NOT',
          'LP', 'RP')
```

Estes tokens garantem que todos os padrões de sintaxe dos blocos FPYTHON são identificados. No entanto surgem alguns conflitos como por exemplo em declarações e instruções uma vez que ambas usam o carater "="onde este tem significados diferentes. Para isso o grupo teve que definir estados do lexer, para este poder navegar pelo código e saber distinguir estas nuances.

Palavras reservadas

Como palavras reservadas o grupo apenas definiu as relacionadas com as frases condicionais.

```
reserved = {
    'if'     : 'IF',
    'else'   : 'ELSE',
    'then'   : 'THEN'
}
```

Estados

Relativamente aos estados do Lexer o grupo definiu os seguintes:

```
states = (
    ('lang','exclusive'),
    ('func','inclusive'),
    ('inst','inclusive'),
)
```

Estes estados permitem criar regras diferentes para tokens com conteúdo igual mas com significados diferentes dependendo dos estados. O primeiro estado "lang"é relativo a todas as operações fora de funções. Este é acionado quando um token INIT é encontrado, e termina quando um token END é encontrado. O segundo é ativado quando uma declaração de função surge e termina quando esta termina. No entanto, dentro dessas mesmas funções temos o estado "inst"que define regras para a sintaxe de cada instrução. O qual inicia quando um token INST aparece e termina quando EINST aparece. Tal como foi referido estes estados foram criados para colmatar os conflitos de semântica existentes no carater "=".

Parser

O nosso Parser compreendido no ficheiro "parser_FP.py" garante que todos os blocos de FPYTHON tokenized pelo lexer são parsed e convertidos em Python puro.

Precedências

De modo a definir as prioridades das operações aritméticas foram criadas precedências no parser.

```
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES'),
    ('left', 'DIVIDE', 'MOD'),
)
```

Estas precedências garantem que as regras de aritmética são cumpridas, uma vez que o parser põe dentro de parenteses as expressões de acordo com as suas prioridades.

Parse

Na generalidade dos casos do parser este apenas copia e cola pedaços de texto e troca apenas os operadores, uma vez que grande parte da sintaxe é semelhante com Python, no entanto no caso das funções, das instruções e das listas por compreensão alguns procedimentos são necessários para conseguir fazer a tradução do código. De modo a permitir o pattern matching nos argumentos das funções e permitir recursividade o grupo adotou as dicas descritas no enunciado. Apartir das quais conseguimos fazer o parse e traduzir todos os casos, podendo tratar listas tal como no Haskell "(<head>:<tail>)"com qualquer padrão na head e qualquer string na tail ou o operador ":"no corpo das instruções que permite o mesmo comportamento como em haskell. De realçar que todas as funções definidas em blocos FPYTHON quando convertidas passam a ter o prefixo "f_"para identificar que foram escritas com a extensão.

Como usar

De modo a usar esta ferramenta o utilizador apenas tem que colocar o ficheiro que pretende traduzir na diretoria do programa e correr o comando:

```
python3 parser_FP.py <input-file> <output-file>
```

Após o qual o programa gera um novo ficheiro com o nome específicado no comando de Python puro e completamente funcional. Na diretoria da ferramenta estão 3 ficheiros com exemplos, os quais o utilizador pode experimentar a ferramenta e perceber como esta funciona e qual a sintaxe a usar.

Conclusão

Em síntese, o grupo acredita que conseguiu resolver todos os objetivos a que se propôs e que o enunciado indicava, no entanto reconhece que existem aspetos que poderiam ser melhorados e funcionalidades adicionais que poderiam ser implementadas tais como pattern matching no corpo das funções, composição de funções ou funções de ordem superior. No entanto, percebemos que este enunciado tem um teto de escalabilidade extremamente elevado e que atingir a perfeição seria algo muito trabalhoso e de todo não o objetivo da elaboração deste trabalho. Na generalidade o grupo aprendeu a criar gramáticas e parsers que respondem às necessidades das mesmas com a biblioteca "ply.lex"e desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora.