Teste: Gramáticas livre de contexto

Compiladores 1, Prof. Fábio M Mendes

04 de Novembro de 2019

Instruções

- Cada teste deve ser feito por uma ou duas pessoas.
- Caso o teste seja feito em dupla, não pode ser uma dupla que já foi utilizada anteriormente.
- As respostas serão entregues em papel, com letra legível, em caneta ou lápis.

Alunos

Nome/Matrícula:

Nome/Matrícula:

1. Matrizes

A linguagem Júlia representa matrizes utilizando a seguinte notação:

```
[ a b c ; d e f ; g h i ]
```

As linhas são separadas por um ponto e vírgula e cada elemento em uma linha é separado por espaços. Crie uma gramática que represente estas matrizes. Assuma que espaços em branco são ignorados (equivalente ao comando `%ignore /\s+/` do Lark) e que *já existe* uma regra definida para representar os elementos atômicos como números e variáveis, que chamamos de `atom`.

Declare a sua gramática utilizando uma notação EBNF (*Extended Backus-Naur Form*), semelhante ao Lark.

```
# Exemplos válidos
[ a b ; d e  ]
[ a ; b ; c ]
[ a b c ]
[]


# Exemplos inválidos
[ [a b] [c d] ]
[ [a b] ; c d ]
[ [a] [b] ; [c] [d] ]
[[]]
```

Lembre-se que sua gramática deve aceitar todas strings válidas da linguagem e recusar todas as strings inválidas.

2. Listas com separadores

Crie uma gramática livre de contexto em notação EBNF, que reconheça uma sub-linguagem do Python que representa listas e listas de listas.

Assim como no caso anterior, assuma que espaços em branco são ignorados (equivalente ao comando `%ignore /\s+/` do Lark) e que *já existe* uma regra definida para representar os elementos atômicos como números e variáveis, que chamamos de `atom`.

Os elementos devem ser separados por vírgulas e a linguagem **não** deve aceitar uma vírgula depois do último elemento da lista.

```
# Exemplos válidos
[a, b, c]
[[a], [b, c]]
[]
[[]]

# Exemplos inválidos
[a,,b]
a
[a b c]
[a, b,]
[,]
[[a]
```

Lembre-se que sua gramática deve aceitar todas strings válidas da linguagem e recusar todas as strings inválidas.

3. Construindo árvores sintáticas

Considere a árvore gramática abaixo e construa a árvore sintática concreta das seguintes expressões: (a) `a + b + c`, (b) `a * b + c`, (c) `a + b * c` e (d) `a * (b + c)`. Você *pode*, se desejar, reduzir expressões de um único filho da mesma forma que o Lark faz quando se declara uma regra com o operador `?<regra>`.

```
expr : term OP_M expr
     | term

term : pow OP_A term
     | pow

pow  : atom "^" pow
     | atom

atom : NAME
     | "(" expr ")"

OP_A : /[+-]/
OP_M : /[*\/]/
NAME : /[a-z]+/
```

(e) A gramática acima implementa corretamenta as regras de precedência e associatividade das operações algébricas fundamentais?