Segundo Exercício Programa

Eric Rodrigues Pires, Mateus Nakajo de Mendonça

Nesse relatório, criamos um algoritmo que gera palavras a partir de uma gramática sensível ao contexto, para reconhecimento de cadeias.

***Palavras-chave*—Elixir, programação funcional, recursão, gramática, hierarquia de Chomsky.**

I. Introdução

NESTE exercício, tivémos como objetivo criar um programa capaz de determinar se uma uma dada cadeia pertence a uma dada linguagem recursiva. A solução implementada para resolver esse problema foi a sugerida na proposta do exercício, e consiste em gerar todas as formas sentenciais da gramática que tenham tamanho menor ou igual à cadeia a ser verificada.

II. Desenvolvimento

Usamos a linguagem de programação Elixir para desenvolver o algoritmo. Por se tratar de uma linguagem funcional, escrevemos os *loops* como recursões com *pattern matching*. Utilizamos também funções algumas da biblioteca padrão Enum, como sort/1, filter/2, all?/2, count/1, dentre outras. Essas funções foram muito úteis para manipular dados em listas e simplificaram bastante o código.

1) Modelagem

A função desejada no nosso programa é a generates_word?/5. Ela recebe como parâmetro uma cadeia (representada por uma lista) e uma gramática. Porém, o principal da lógica será realizado na função generate_all/5, que recebe um comprimento máximo de cadeia e uma gramática. Em ambas, a gramática é construída pelos outros 4 parâmetros, a saber: o não-terminal inicial; as regras (representadas por uma lista de tuplas), os terminais (representados por uma lista) e os não terminais (representados por uma lista). Uma cadeia vazia é representada por uma lista vazia. O tipo dos elementos terminais é indiferente para nosso programa.

2) Algoritmo

Para esta atividade, o desenvolvimento do algoritmo foi realizado em etapas de acordo com o objetivo final e as necessidades de projeto.

A lógica recursiva por trás do programa será a seguinte:

1) Começar a partir do não-terminal inicial.
2) Procurar possíveis substituições de regras de produção, e ramificar a execução da seguinte forma:
   - Substituir a regra de produção e iterar por toda a cadeia novamente.
   - Continuar iterando pela cadeia, sem realizar tal substituição de regra de produção.
3) Realizar as substituições até ultrapassar o comprimento máximo.
4) Executar os itens 2 e 3 até não haver mais regras de substituição em cada palavra.
5) Retornar apenas as palavras formadas por terminais.

Dessa forma, podemos realizar múltiplas substituições de regras de produção em uma cadeia, permitindo a criação de cadeias mais elaboradas.

Em uma primeira etapa, o algoritmo desenvolvido simplesmente varria cada letra da palavra e verificava se era um não-terminal. Encontrado um, buscaria todas as regras correspondentes da lista e realizaria as substituições adequadas. Desta forma, o algoritmo era simples e executava corretamente para gramáticas livres de contexto e regulares. Porém, para o caso de múltiplos não-terminais do lado esquerdo de uma regra, ou terminais e não-terminais mistos, como é o caso de gramáticas sensíveis ao contexto, o algoritmo não funciona.

Dessa forma, na segunda etapa, alteramos a lógica para iterar sobre subcadeias da cadeia atual, e verificar se há regra(s) equivalente(s) para a subcadeia analisada. Assim, o algoritmo funciona para todas as gramáticas esperadas. Porém, agora há muitas verificações sendo feitas, aumentando muito a pilha de recursão do programa, o que causa estouro de pilha para gramáticas complexas com tamanho máximo de cadeia relativamente pequeno.

A solução encontrada na terceira etapa foi iterar sobre as regras de produção, encontrando subcadeias correspondentes na cadeia atual de forma inteligente (por exemplo, buscando subcadeias com o mesmo comprimento do lado esquerdo da regra). Assim, os tempos de execução obtidos são muito melhores, e a maior parte dos estouros de pilha foram resolvidos.

Porém, percebemos que havia um caso nas gramáticas sensíveis ao contexto que poderia gerar loop infinito, no caso de regras circulares (por exemplo, $AB \rightarrow BA$ e $BA \rightarrow AB$). Para isso, numa quarta etapa, implementamos uma variável que acompanha o caminho realizado até então e evita que um estado anterior seja alcançado posteriormente. Isso evita qualquer loop como descrito na construção da cadeia, mas piora um pouco a execução do programa.

3) Testes

Serão realizados dois tipos de testes. O primeiro tipo se preocupa em verificar as cadeias geradas por uma gramática e comprimento fornecidos. O segundo analisa a funcionalidade do método de pertencimento da palavra ao conjunto de cadeias gerado pela gramática.

Os testes desenvolvidos possuirão gramáticas dos três tipos da hierarquia de Chomsky que nos concernem: sensíveis ao contexto, livres de contexto e regulares. Desta forma, poderemos identificar o funcionamento para diferentes estruturas das regras de produção.

III. Resultados

Os testes realizados bateram com o esperado, mostrando a sua funcionalidade com nosso programa. Assim, o algoritmo foi capaz de gerar corretamente as linguagens de gramáticas desde regulares até sensíveis ao contexto, e identificar se uma palavra pertencia à mesma.

1) Problemas identificados

Nos nossos testes com gramáticas mais complexas, a execução pode demorar muito tempo para comprimentos grandes, e que em alguns casos podem causar estouro de pilha, como para a linguagem sensível ao contexto $(x|y|z)^*$ com $n^o x = n^o y = n^o z$ para comprimento de cadeia ≥ 6.

Além disso, como proposto no enunciado, foi realizada uma implementação alternativa – gerar todas as formas setenciais parciais da gramática recursivamente e analisar a alteração –, mas foi detectada degradação tanto na velocidade do programa quanto na quantidade de memória para os mesmos testes, já que uma mesma regra seria aplicada múltiplas vezes a cada iteração. Por isso, optamos pela solução desenvolvida anteriormente.

IV. Conclusão

Esse exercício programa nos permitiu ampliar e desenvolver nossos conceitos em matemática discreta sobre gramáticas, e pela linguagem Elixir, melhorar nossoso conhecimentos de programação funcional.

Quanto ao programa desenvolvido, também avaliamos que, para melhorar a sua execução, o próximo passo seria evitar redundância de aplicação de regras de produção em uma cadeia (i.e. somente ignorar uma regra $A \rightarrow ...$ se houver alguma regra $AB \rightarrow ...$ na construção da gramática; caso contrário, parar a iteração). Porém, tal alternativa exige refatoração de todo o código para armazenar um estado global, e deve considerar muitos casos de borda como a ordem de produção de regras diferentes, que nem sempre é a mesma ordem de leitura da cadeia.

Referências

[1] ALMEIDA, Ulisses. *Learn Functional Programming with Elixir*. The Pragmatic Bookshelf, 2018.

[2] FORD, Neal. *Functional Thinking*. Disponível em: https://www.infoq.com/presentations/Functional-Thinking. Acesso em 7 de fevereiro de 2018.

[3] Elixir. *Introduction*. Disponível em: https://elixir-lang.org/getting-started/introduction.html. Acesso em 7 de fevereiro de 2018.

[4] Elixir. *Enum*. Disponível em: https://hexdocs.pm/elixir/Enum.html Acesso em 24 de fevereiro de 2018.