Quarto Exercício Programa

Eric Rodrigues Pires, Mateus Nakajo de Mendonça

Nesse relatório, elaboramos funções em programação funcional para linguagens livres de contexto, para normalização de gramática e identificação de cadeias de gramáticas normalizadas, através de programação dinâmica.

***Palavras-chave*—Elixir, programação funcional, gramática livre de contexto, forma normal de Chomsky, programação dinâmica.**

I. Introdução

NESTE exercício, tivemos como objetivo implementar um algoritmo para converter uma gramática de uma linguagem livre de contexto (LLC) qualquer para a Forma Normal de Chomsky e reconhecer cadeias de LLC a partir da gramática descrita em Forma Normal de Chomsky usando programação dinâmica.

Na Forma Normal de Chomsky, o lado direito de toda regra de uma gramática livre de contexto deve ter, no máximo, comprimento 2. Devido a isso, gramáticas nessa forma não podem produzir cadeias com comprimento menor que 2. Entretanto, com exceção dessa limitação, é possível provar que toda gramática livre de contexto pode ser escrita na Forma Normal de Chomsky [8].

Gramáticas descritas na Forma Normal de Chomsky podem ser reconhecidas facilmente com um algoritmo de programação dinâmica, que será explicado na seção "Algoritmo". No paradigma de programação dinâmica, subinstâncias do problema são resolvidas recursivamente e armazenadas em uma tabela, a fim de evitar que sejam recalculadas na recursão. Essa tabela é a base do algoritmo de programação dinâmica e é essencial para sua eficiência.

II. Desenvolvimento

Lorem ipsum dolor sit amet lorem ipsum dolor sit amet lorem ipsum dolor sit amet lorem ipsum dolor sit amet lorem ipsum dolor sit amet.

1) Modelagem

Lorem ipsum dolor sit amet lorem ipsum dolor sit amet lorem ipsum dolor sit amet lorem ipsum dolor sit amet lorem ipsum dolor sit amet.

2) Algoritmo

O algoritmo desenvolvido para converter uma gramática de LLC para a Forma Normal de Chomsky se baseou em [6] e segue os seguintes passos:

1) START: Introduzir um novo não-terminal inicial S_0, que gera o não-terminal anterior.
2) TERM: Transformar regras com lado direito de comprimento > 1 com pelo menos um terminal em regras apenas com não-terminais. Para cada terminal substituído, adicionamos uma regra tal que o novo não-terminal gere o terminal.
3) BIN: Transformar regras que geram mais de dois não-terminais em várias regras em sequência, com novos não-terminais que indicam esta sequência.
4) DEL: Remover regras que geram cadeia vazia. Para isso, nas regras em que o não-terminal que gera vazioestá do lado direito, além das regras que geram tal não-terminal, deve haver outra regra que não inclui este não-terminal, apagando-o do lado direito em cada cópia de tal regra. Como este processo pode gerar novos não-terminais gerando cadeia vazia, ele é realizado iterativamente até o não-terminal inicial S_0, que é o único permitido para gerar cadeias vazias.
5) UNIT: Todas as regras do último passo que geram apenas um não-terminal devem ter substituição diretamente para o não-terminal do lado esquerdo ter todas as regras em que o não-terminal aparece do lado esquerdo de outras regras. Dessa forma, certos não-terminais que se tornam inúteis terão que ser eliminados iterativamente (i.e. podem gerar mais não-terminais inúteis). A identificação de não-terminais inúteis se dará pela verificação de não-terminais que não podem ser atingidos por nenhuma regra após esta substituição.
6) Adicionalmente, pode ser que S_0 e o não-terminal inicial original gerem exatamente as mesmas regras, por exemplo quando não há geração de cadeias vazias; neste caso, precisamos de um passo a mais para eliminar o não-terminal inicial e substituir suas regras pelo novo início S_0.

Em termos de implementação, foi elaborado um algoritmo independente para cada etapa, que será aplicado nas regras de produção e parâmetros extras necessários para cada uma.

Para a segunda parte deste exercício, utilizamos a referência em [8] para a elaboração de um algoritmo em programação dinâmica, que usa uma tabela N onde os elementos $N[i, i]|1 \leq i \leq n, n = |x|$ são pré-populados com os não-terminais que podem gerar o terminal na posição i da palavra x a ser comparada. O algoritmo determina os seguintes passos dado não-terminal inicial S_0:

```
Para s := [1, n-1]
  Para i := [1, n-s]
    Para k := [i, i+s-1]
      Se existe uma regra (A → BC) ∈ R
      | B ∈ N[i, k], C ∈ N[k+1, i+s]
        Adicione A em N[i, i+s]
Aceite x se S0 ∈ N[1, n]
```

A implementação foi feita aproveitando-se os recursos funcionais de Elixir, como `Enum.reduce`, como alternativa para os loops.

3) Testes

Para este exercício, utilizamos dois testes para cada uma das funções elaboradas. Destes dois testes, utilizamos duas linguagens livres de contexto diferentes, sendo uma delas também uma linguagem regular, e tendo apenas uma delas gerando uma cadeia vazia. Desta forma, podemos encontrar possíveis erros de implementação para cadeias vazias nestas funções.

Para a função `NormalForm.change_to_normal_form/3`, realizamos os testes tais que apenas a estrutura da gramática deveria estar correta, i.e. devem permitir que os não-terminais gerados possuam qualquer nome. Dessa forma, ele varre as regras e verifica as propriedades das gramáticas geradas conforme a forma normal esperada. Para estes testes, utilizamos as linguagens $(aa)^*$ e $a^n b^n | n \geq 1$.

Já nos testes de `NormalForm.normal_form_grammar_generates_word?/2`, dada uma gramática já na forma normal, verificamos diferentes cadeias que devem ou não ser aceitas por aquela gramática, comparando os resultados obtidos. Os testes foram realizados para as linguagens $a(ab)^*$ e de parênteses equilibrados.

III. Resultados

Os testes realizados bateram com o esperado, de acordo com a especificação do exercício. Assim, o algoritmo foi capaz de gerar corretamente gramáticas na Forma Normal de Chomsky. A função de verificação de cadeias para gramáticas na forma normal, através de programação dinâmica, também funcionou corretamente.

IV. Conclusão

Lorem ipsum dolor sit amet lorem ipsum dolor sit amet lorem ipsum dolor sit amet lorem ipsum dolor sit amet lorem ipsum dolor sit amet.

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.

[5] Elixir. *Structs*. Disponível em: https://elixir-lang.org/getting-started/structs.html. Acesso em 22 de março de 2018.

[6] Wikipedia, The Free Encyclopedia. *Chomsky Normal Form*. Disponível em: https://en.wikipedia.org/wiki/Chomsky_normal_form. Acesso em 31 de março de 2018.

[7] Elixir. *List*. Disponível em: https://hexdocs.pm/elixir/List.html. Acesso em 03 de abril de 2018.

[8] LEWIS, H.; PAPADIMITRIOU, C. *Elements of the Theory of Computation*. 2ª edição. Prentice Hall, 1998.