Arthur Andrade Ilario
NUSP: 10432515

Exercício Prático 2

PCS3556
15 de Fevereiro de 2021

Introdução

Essa atividade coloca alguns dos conceitos explorados em sala de aula em prática. O objetivo é implementar um algoritmo que reconheça cadeias a partir de uma gramática recursiva. Isto é, segundo a hierarquia de Chomsky, o algoritmo deve reconhecer gramáticas regulares, livres de contexto e sensíveis ao contexto.

Sobre o projeto

Os arquivos estão organizados em 2 diretórios principais: *lib* e *test*. O código fonte para o programa se encontra no arquivo */lib/parser.ex* e a suíte de testes se encontra no arquivo */test/parser_test.exs*

1 Abordagem e implementação

Para realizar as tarefas computacionais propostas, foi utilizada a linguagem funcional Elixir (1.11.2).

Através de um único módulo, *Parser*, todas as operações necessárias para esse exercício são realizadas. A abordagem utilizada para a construção do sistema foi a sugerida no enunciado da atividade, ou seja, primeiro foi estudada uma função que percorresse uma lista recursivamente, retornando seus elementos. A partir desse conhecimento criou-se uma função capaz de gerar cadeias recursivamente e finalmente foi montado o sistema que usasse essa função para verificar o pertencimento de uma cadeia.

Finalmente a função principal do módulo é responsável por garantir que estamos tratando uma cadeia (w) e não uma sentença, medi-la e invocar a função que vai gerar a linguagem contendo todas sentenças de tamanho $l \leq |w|$ usando a gramática fornecida (explicitada pela tupla no segundo argumento da função). A verificação então é feita testando se w pertence a linguagem gerada.

```
1   def parse(w, {non_terminal, terminal, ruleset, initial}) do
2     # Garante que a cadeia nao eh uma forma sentencial
3     assert(Enum.all?(w, &(&1 in terminal)))
4
5     # Mede a cadeia e entao gera a linguagem (com sentenciais)
6     #conforme a gramatica ate aquele tamanho
7     size = length(w)
8     language = generate([initial], ruleset, size)
9
10    # Verifica se a cadeia pertence a linguagem
11    w in language
```

```
12    end
```

1.1 Estrutura de entrada

Houve duas escolhas principais de estruturas de dados para a realização da atividade. A primeira decisão foi de como representar as cadeias e sentenças no programa. A linguagem Elixir fornece duas estruturas de dados para representar cadeias de caracteres: *Strings*, uma estrutura binária armazenada sequencialmente na memória, e *charlists*, listas ligadas compostas por números inteiros que codificam um caractere válido em Unicode.

A escolha foi de utilizar *charlists* uma vez que as funções precisariam acessar as cadeias elemento a elemento para identificar padrões e por ser uma lista ligada a *charlist* implementa o protocolo Enum, permitindo o uso de uma gama de funções para manipular as cadeias (a estrutura *String* possui muitos métodos mas listas possuem uma interface de acesso à elementos mais prática através das funções hd() e tl() além de possuir um método essencial para o programa, *Enum.chunk_every/4*)

A segunda decisão foi definir uma estrutura para representar a gramática. A linguagem não possui uma solução pronta para esse caso, portanto uma nova estrutura foi criada a partir das existentes. Por representar um conjunto de 4 informações diferentes (símbolos não-terminais, símbolos terminais, regras de derivação e símbolo inicial) foi utilizada uma tupla com 4 elementos:

1. *non_terminal* - Uma lista do tipo *charlist* para representar os símbolos não terminais
2. *terminal* - Uma lista do tipo *charlist* para representar os símbolos terminais
3. *ruleset* - Uma lista de pares contendo duas *charlist*, uma representando a sequência que vai ser substituída e outra representando a substituição (para representar as regras de derivação)
4. *initial* - Um caractere (ou seja, um número codificando um caractere Unicode) para representar o símbolo inicial

Dessa forma foram criados dois novos *types* no programa para definir esse tipo de entrada:

```
1   @type ruleset :: [{charlist, charlist}]
2   @type grammar :: {charlist, charlist, ruleset, char}
```

1.2 Geração de linguagem

A parte principal deste EP é a geração de cadeias da linguagem recursivamente. A função que administra essa tarefa é a *generate/3* abaixo:

```elixir
def generate(ti, ruleset, max_length) do
  # Gera o proximo conjunto Ti+1 a partir do conjunto de sentencas anterior (Ti)
  # Para cada sentenca em Ti realiza derivacoes com a funcao transform
  # Joga fora sentencas geradas que sao repetidas (uniq)
  ti_1 = Enum.reduce(ti, [], &transform(&1, ruleset, max_length, &2)) |> Enum.uniq()

  # Para printar os passos de derivacao descomente a linha abaixo
  # IO.inspect(ti_1)

  # Verifica se o conjunto gerado tem os mesmos elementos (eh igual) ao anterior
  # Caso tenha, o conjunto obtido eh o final, caso contrario recursao com o novo conjunto
  if ti_1 -- ti == [] do
    ti
  else
    generate(ti_1, ruleset, max_length)
  end
end
```

A partir de um conjunto contendo, inicialmente, apenas o símbolo inicial, o conjunto de regras de derivação e o tamanho máximo das sentenças a serem geradas, essa função determina recursivamente o restante da linguagem.

A cada nova chamada o conjunto T_i aumenta, sendo que a função *transform/4* é chamada para cada sentença dentro do conjunto T_i afim de gerar o novo conjunto T_{i+1}.

A condição de parada é atingida quando $T_{i+1} = T_i$. Neste ponto $T_i = T_{i+1} = L(w)$, onde L(w) é a linguagem contendo todas as sentenças de tamanho $l \leq |w|$.

1.3 Derivação de sentenças

A derivação de sentenças é a parte mais importante para a geração da linguagem (explicada na seção anterior). Ela é feita em 3 funções diferentes: *transform/4*, *transform/2* e *apply_rule/6*.

```elixir
def transform(w, ruleset, max_length, acc) do
  # Para cada regra de derivacao gera uma lista de derivacoes se possivel
  # Depois filtra as que sao maiores que o tamanho maximo
```

```
4       tw = ruleset |> Enum.flat_map(&transform(w, &1)) |> Enum.filter(&
           length(&1) <= max_length)
5
6       # Adiciona a sentenca inicial, perdida durante as derivacoes
7       # Bem como o acumulador para funcionar no reduce
8       [w] ++ tw ++ acc
9     end
10    def transform(w, {a, b}) do
11      # Mede o tamanho do elemento que estamos buscando para substituir
12      size = length(a)
13
14      # Agrupa a sentenca em blocos do tamanho do elemento para
           facilitar a comparacao
15      # Verifica os grupos recursivamente
16      w |> Enum.chunk_every(size, 1, :discard) |> apply_rule(w, {a,b},
           [], size, 0)
17    end
```

No conjunto de funções acima é criado um conjunto de derivações para cada termo no conjunto T_i e para cada regra de derivação da gramática. Por exemplo, para uma sentença "aaSS"e duas regras de derivação S -> a e S -> b serão gerados dois conjuntos de derivação: {"aaaS", "aaSa"} e {"aabS", "aaSb"}.

É importante explicar o funcionamento de *Enum.chunk_ every/3*. Para regras de derivação que substituem padrões de comprimento $l \geq 2$ a sentença que está sendo analisada é quebrada em blocos de tamanho l. Esses blocos se sobrepõem, de forma que o algoritmo possa analisar todas as tuplas dentro da sentença. Isso torna o processo de identificação e substituição do padrão mais fácil, como visto na próxima função.

Os conjuntos são gerados na função *apply_ rule/5* e então concatenados para criar o conjunto de todas as derivações geradas a partir de uma sentença w.

```
1     defp apply_rule([head | tail], original, {a, b}, t, _rule_size,
         index) do
2       # Se o padrao do primeiro bloco corresponde a regra, substitui
3       if head == a do
4         # Quebra a sentenca em 3 partes:
5         # - prefixo
6         # - o elemento que vai ser substituido
7         # - sufixo
8         {prefix, suffix} = Enum.split(original, index)
9         suffix = suffix -- a
10
11        # Reconstroi a sentenca substituindo o elemento
```

```
12        # Adiciona ele na lista de derivacoes com essa regra
13        t = [prefix ++ b ++ suffix] ++ t
14        apply_rule(tail, original, {a,b}, t, _rule_size, index + 1)
15      else
16        # O padrao nao foi encontrado, entao soh incrementa o indice
17        apply_rule(tail, original, {a,b}, t, _rule_size, index + 1)
18      end
19    end
20    defp apply_rule([], _original, _ruleset, t, _rule_size, index) do
21      t
22    end
```

A essência do reconhecimento e substituição de padrões se encontra nessa função. Ela apenas recebe um bloco de símbolos e se esse bloco corresponder a regra de substituição, uma nova sentença é gerada, com o bloco substituído.

1.4 Exemplo de funcionamento

Descomentando a linha 22 na função *generate/3* é possível ver o algoritmo em funcionamento, isto é, ver o valor de cada conjunto T_i a cada chamada recursiva.

Como exemplo, usaremos de entrada a sentença "S"e regras de derivação $P = \{S \longrightarrow aBC, S \longrightarrow aSBC, CB \longrightarrow BC, aB \longrightarrow ab, bB \longrightarrow bb, bC \longrightarrow bc, cC \longrightarrow cc\}$. Essas regras de derivação fazem parte da gramática que forma a linguagem $a^n b^n c^n$. Essa é saída da função para cadeias de tamanho $l \leq 6$:

```
1  ['S', 'aBC', 'aSBC']
2  ['aSBC', 'aaBCBC', 'aBC', 'abC', 'S']
3  ['S', 'aBC', 'aSBC', 'abC', 'abc', 'aaBCBC', 'aaBBCC', 'aabCBC']
4  ['aabCBC', 'aabBCC', 'aabcBC', 'aaBBCC', 'aaBCBC', 'abc', 'abC', 'aSBC', 'aBC', 'S']
5  ['S', 'aBC', 'aSBC', 'abC', 'aaBCBC', 'abc', 'aaBBCC', 'aabCBC', 'aabBCC', 'aabcBC', 'aabbCC']
6  ['aabbCC', 'aabbcC', 'aabcBC', 'aabBCC', 'aabCBC', 'aaBBCC', 'abc', 'aaBCBC', 'abC', 'aSBC',
7  'aBC', 'S']
8  ['S', 'aBC', 'aSBC', 'abC', 'aaBCBC', 'abc', 'aaBBCC', 'aabCBC', 'aabBCC', 'aabcBC', 'aabbCC',
9  'aabbcC', 'aabbcc']
10 ['aabbcc', 'aabbcC', 'aabbCC', 'aabcBC', 'aabBCC', 'aabCBC', 'aaBBCC', 'abc', 'aaBCBC', 'abC',
11 'aSBC', 'aBC', 'S']
```

É possível observar que o algoritmo para quando as duas últimas listas são iguais (embora estejam em ordem reversa).

2 Testes

Existem ao todo 5 testes a serem realizados que podem ser divididos em 3 categorias: teste da função *transform/4*, teste da função *parser/2* e teste de desempenho.

Para rodar a suíte de teste basta executar o comando *mix test* na pasta principal do projeto. Para adicionar novos testes, basta copiar o modelo dos testes já presentes no arquivo */test/parser_ test.exs*

2.1 Teste da função *transform/4*

O teste consiste em verificar se a função *transform/4* produz as derivações esperadas para uma dada sentença e conjunto de regras (lembrando que essa função só executa "um passo"de derivação por regra). Os testes dessa categoria são todos realizados com as regras de derivação: $P = \{S \longrightarrow a, S \longrightarrow b\}$

Testes realizados:

1. Cadeia a ser derivada: 'abSabS'

   Saída esperada: ['abSabS', 'abaabS', 'abSabb', 'abSaba', 'abbabS']

2. Cadeia a ser derivada: 'ab'

   Saída esperada: ['ab']

O primeiro teste verifica se todas as possíveis cadeias geradas após um passo estão presentes na saída e o segundo teste verifica se sentenças não-deriváveis permanecem inalteradas.

2.2 Teste da função *parser/2*

O teste consiste em verificar se o sistema está funcionando corretamente. Primeiro executamos um teste com uma gramática trivial e depois com uma gramática um pouco mais complexa.

Testes realizados:

1. $G = (\{S, A, B\}, \{a, b\}, P, S)$

   $P = \{S \longrightarrow ab, S \longrightarrow aA, A \longrightarrow Bb, A \longrightarrow b, B \longrightarrow aA\}$

   Cadeia a ser verificada: 'aabb'

   Saída esperada: *true* (ou seja, a cadeia pertence a gramática)

2. $G = (\{S, A, B\}, \{a, b\}, P, S)$

   $P = \{S \longrightarrow ab, S \longrightarrow aA, A \longrightarrow Bb, A \longrightarrow b, B \longrightarrow aA\}$

   Cadeia a ser verificada: 'aabbb'

   Saída esperada: *false* (ou seja, a cadeia não pertence a gramática)

3. $G = (\{S, B, C\}, \{a, b, c\}, P, S)$
   $P = \{S \longrightarrow aBC, S \longrightarrow aSBC, CB \longrightarrow BC, aB \longrightarrow ab, bB \longrightarrow bb, bC \longrightarrow bc, cC \longrightarrow cc\}$
   Cadeia a ser verificada: 'aabbcc'
   Saída esperada: *true*
4. $G = (\{S, B, C\}, \{a, b, c\}, P, S)$
   $P = \{S \longrightarrow aBC, S \longrightarrow aSBC, CB \longrightarrow BC, aB \longrightarrow ab, bB \longrightarrow bb, bC \longrightarrow bc, cC \longrightarrow cc\}$
   Cadeia a ser verificada: 'aaabbbcc'
   Saída esperada: *false*

Para cada gramática foram feitos dois testes, um para verificar o caso positivo e outro para verificar o caso negativo.

2.3 Teste de desempenho

Esses dois últimos testes foram feitos apenas no intuito de verificar o desempenho do programa para entradas grandes (cadeias longas e regras longas). A métrica usada para medir esse desempenho é meramente o tempo gasto na execução do teste.

Testes realizados:

1. $G = (\{S, B, C\}, \{a, b, c\}, P, S)$
   $P = \{S \longrightarrow aBC, S \longrightarrow aSBC, CB \longrightarrow BC, aB \longrightarrow ab, bB \longrightarrow bb, bC \longrightarrow bc, cC \longrightarrow cc\}$
   Cadeia a ser verificada: 'aaaaaaaabbbbbbbbcccccccc'
   Saída esperada: *true*
2. $G = (\{S, A, B\}, \{a, b, c\}, P, S)$
   $P = \{S \longrightarrow AAAAAAABS, AAAAAAA \longrightarrow aaaaaaaS, B \longrightarrow b, S \longrightarrow c, A \longrightarrow BS\}$
   Cadeia a ser verificada: 'aaaaaaacbc'
   Saída esperada: *true*

Para o primeiro teste de desempenho (em relação ao tamanho das cadeias) observou-se que o tempo de execução para cadeias maiores aumentava significativamente apenas quando as cadeias eram aceitas. Por exemplo, no caso teste apresentado a linguagem da gramática é $a^n b^n c^n$. Sendo assim a próxima cadeia aceita após $a^8 b^8 c^8$ seria $a^9 b^9 c^9$ todas as cadeias com tamanho entre as duas cadeias ($l = 25, 26$) possuem tempo de execução muito próximo a cadeia $a^8 b^8 c^8$.

Tempo de execução para cadeias:

1. $a^7b^7c^7$ ($l = 21$): 1966.9ms
2. $a^7b^7c^8$ ($l = 22$): 2336.6ms
3. $a^7b^8c^8$ ($l = 23$): 2274.4ms
4. $a^8b^8c^8$ ($l = 24$): 9783.3ms
5. $a^8b^8c^9$ ($l = 25$): 11215.8ms

Observe que o salto no tempo de execução ocorre do terceiro para o quarto caso, bem quando a cadeia passa a ser válida novamente.

Já para o teste de desempenho de regras longas, notou-se que o tempo de execução não é tão significativamente afetado por esta variável como é para o tamanho da cadeia de entrada.

3 Próximos passos

Existem diversas melhorias que podem ser avaliadas para otimizar o funcionamento do algoritmo. O programa não tira proveito de uma das principais características do Elixir, isto é, o paralelismo. Existem pelo menos dois trechos do programa que poderiam ser paralelizados, a execução da função *transform* para cada elemento de T_i (cada elemento poderia ser executado em um processo isolado) e a execução da função *apply_rule* para cada regra ou para cada bloco de símbolos (a aplicação de cada regra poderia ser realizada em paralelo ou a analise de cada bloco poderia ser realizada em paralelo).

Além da paralelização, poderiam ser feitas otimizações para cada tipo de gramática identificada. Isto é, uma gramática regular, por exemplo, possui propriedades que permitem que um algoritmo de reconhecimento seja otimizado para ela.

Outras análises também podem ser feitas em cima do código no geral afim de encontrar possíveis bottlenecks e tarefas que são executadas desnecessariamente mais de uma vez.