gramáticas, computadores e expressões matemáticas

@o-santi

<2021-09-21 Tue>

Contents

1 computadores e linguagens

computadores para a maior parte das pessoas parecem máquinas sobrenaturais, que de alguma maneira misturam zeros e uns para produzir através de métodos misteriosos aplicativos extremamente avançados como youtube e whatsapp. a relação entre esses aplicativos e as sequências em binários, por mais obtusa que seja, é dada através das linguagens de programação; isto é, essa linguagens fazem a ponte entre o código diretamente entendido pela máquina e o que nós seres humanos conseguimos compreender, abstraindo diversos conceitos para facilitar a vida do programador.

entretanto, ainda resta a questão: como o computador consegue entender essas linguagens de programação de modo a transformá-la em um código de máquina correto? a resposta é magia! brincadeira, *quase*, compiladores. porém, explicar o funcionamento um compilador como o *gcc* com suas 7.3 milhões de linhas de código levaria alguns séculos (e não sei se existe algum ser humano que entende *completamente* todo o gcc), então me satisfarei em explicar aqui uma linguagem mais simples: expressões matemáticas.

de fato, se você abrir o google agora e digitar $9*(4+4)$ verá que ele te dará a resposta 72. num primeiro momento, podemos pensar que ele executa as instruções sequencialmente, na ordem em que elas aparecem na expressão, mas rapidamente percebemos que isso não funciona para todos os casos. na verdade, nem para o caso dado essa estratégia funciona, pois teríamos $9*4 = 36+4 = 40$ que até hoje não é igual a 72, e portanto fica claro que ele precisa de alguma maneira de entender a estrutura, a hierarquia das operações para calcular corretamente.

2 terminologia e definições

considerarei aqui que o leitor possui o míminimo conhecimento sobre a terminologia das linguagens regulares. caso não conheça, todas as definições podem ser encontradas nos meus posts anteriores, em especial: expressões regulares, gramáticas regulares e gramáticas livres de contexto.

3 expressões matemáticas

vimos anteriormente que podemos definir uma gramática G_{exp} muito simples que define todas as possíveis expressões que possuem soma e multiplicação. para simplificar, podemos transformar uma expressão do tipo

$$(a*b)+c$$

em

$$(id*id)+id$$

visto que o nome das variáveis nas estruturas é arbitrário, e é muito mais fácil nomeá-las todas pelo mesmo símbolo e guardar os valores em outro lugar (pense que cada símbolo na verdade é um token com várias características, uma delas sendo o valor guardado na variável). portanto, podemos definir o conjunto de terminais como $T = \{id, +, *, (,)\}$, e as regras como:

$$S \to E$$
$$E \to E+E$$
$$E \to E*E$$
$$E \to (E)$$
$$E \to id$$

onde S é o símbolo inicial da gramática. entretanto essa gramática tem um problema: ela é ambígua. isto é, ela possui palavras que tem mais de uma derivação possível. por exemplo, a expressão:

$$id+id*id$$

pode ser derivada das seguintes maneiras:

$$S \to E+E \to E+E*E \to id+id*id = id+(id*id)$$
$$S \to E*E \to E+E*E \to id+id*id = (id+id)*id$$

apesar de parecer ínfima, essa ambiguidade significa que um possível compilador não conseguiria decidir qual das duas operações deve ser executadas primeiro (sem definir por exemplo uma hierarquia de operações). enquanto existem linguagens que são inerentemente ambíguas, isto é, independente da gramática sempre existirá ao menos uma palavra com mais de uma derivação, esse não é o caso da nossa linguagem acima. adicionando novos símbolos e novas regras, podemos eliminar essa ambiguidade:

$$E \to E + T$$
$$E \to T$$
$$T \to T * F$$
$$T \to F$$
$$F \to (E)$$
$$F \to id$$

embora provar que essa gramática não é ambígua seja difícil, a intuição é simples: só podemos multiplicar uma soma utilizando os parenteses, e portanto a única maneira de derivar a palavra $id + id * id$ seria primeiro fazendo a multiplicação e depois a soma.

4 autômatos de pilha

> tudo bem, eu tenho a gramática que descreve as expressões matemáticas e define uma hierarquia. e agora? como eu enfio ela no meu computador?

acalme-se senhor leitor, eu ia explicar agora. assim como no caso das expressões regulares, onde podíamos construir um autômato que as aceitasse, podemos algoritmicamente construir um autômato um pouco mais *sofisticado* que aceite as linguagens livres de contexto.

a ideia principal desses novos autômatos é introduzir uma memória, mais especificamente uma estrutura de pilha, que simula as transições de uma gramática. isto é, colocamos os símbolos que são transformados diretamente na pilha, e o topo da pilha (o caracter mais à esquerda) será utilizado como entrada para a função de transição também.

quando o topo da pilha e o símbolo mais à esquerda da palavra forem o mesmo, podemos retirar ambos da pilha, visto que um terminal não pode ser transformado pela definição de gramática livre de contexto. essa função de transição, entretanto, ainda é uma função não-determinística, então a emulação desse autômato deve levar isso em conta

5 implementando . . .

existem inúmeras maneiras de implementar autômatos de pilha, entretanto num primeiro momento escolhi a maneira que me pareceu mais intuitiva e simples: classes. o estado, o input e a pilha serão

propriedades da classe `Automato` e a função de transição um método. além disso, escolhi python simplesmente porque é a linguagem mais legível e simples para iniciantes.

logo, podemos implementar o autômato com o seguinte excerto:

```
  class Automato:

      """
      Classe que representa um estado de um autômato
      """

      def __init__(self, input_string: str, pilha: str, regras:
       ↪  dict[str, list[str]]):
          self.input_string = input_string
          self.pilha = pilha
          self.regras = regras
```

aqui, nós assumimos que o dicionário das regras é da forma

```
regras = { variavel : transicoes }
# no caso das expressões matemáticas, podemos definí-lo como
regras = {"S":["E"], "E" : ["E+T", "T"], "T": ["T*F", "F"],
 ↪  "F":["(E)", "id"]}
```

pois facilita na hora de adicionar os novos símbolos na pilha. note também que não precisamos explicitamente guardar o estado (inicial ou final) do autômato, visto que todas as transições fora a primeira (que coloca o símbolo inicial S na pilha) estarão no estado final. portanto, se iniciarmos o nosso cálculo direto com S na pilha, não precisamos checar se o autômato está no estado final.

entretanto, temos um problema claro com essa implementação: a função de transição **não** é determinística. isso significa que existirão casos (e eles serão frequentes) onde temos mais de uma possível escolha para a saída da função. e um autômato desse estilo não pode simplesmente possuir múltiplos estados ao mesmo tempo.

a solução portanto é, ao invés de alterar o estado do autômato, simplesmente fazer com que a função de transição retorne todos os possíveis estados que ele pode tomar em forma de novos objetos. traduzindo isso em python:

```
class Automato:

    (...)

    def funcao_de_transicao(self) -> list[Automato]:
```

```
        topo_da_pilha = self.pilha[0] if len(self.pilha) > 0 else
         ↪ None
        if topo_da_pilha in self.regras:
            return [
                Automato(self.input_string, transicao +
                 ↪ self.pilha[1:], self.regras)
                for transicao in self.regras[topo_da_pilha]
            ]
        elif topo_da_pilha == self.input_string[0]:
            return [
                Automato(self.input_string[1:], self.pilha[1:],
                 ↪ self.regras)
            ]
```

ou seja, colocamos os novos símbolos da lista no topo (considerando que o topo seja o índex 0) fazendo `transicoes + self.pilha[1:]` (o que já retira o símbolo que está sendo trocado), e retornamos o objeto resultante para cada possível transição descrita nas regras.

por fim, podemos criar uma função do automato que nos diz se ele aceitou o input ou não:

```
class Automato:

    (...)

    def palavra_aceita(self):
        return len(self.input_string) == len(self.pilha) == 0
```

lembrando que não precisamos checar o estado porque pela construção já será final.

para emular a derivação de uma palavra podemos então utilizar um algoritmo de busca em largura (*breadth first search*), isto é, começamos com o autômato inicial (que já possui S em sua pilha) e calculamos todas as possíveis transições até que uma delas aceite a palavra.

```
def simular_automato(input_string: str, regras: dict[str,
 ↪ list[str]]):
    automato_inicial = Automato(input_string, "S", regras)
    lista_de_automatos = [automato_inicial]
    automatos = {0: lista_de_automatos}
    i = 0
    while not any(automato.palavra_aceita() for automato in
     ↪ lista_de_automatos):
        lista_de_automatos = []
```

```
        for aut in automatos[i]:
            if transicoes := aut.funcao_de_transicao():
                lista_de_automatos.extend(transicoes)
        if lista_de_automatos is None:
            break
        i += 1
        automatos.update({i: lista_de_automatos})
    if len(lista_de_automatos) > 0:
        print(f"A palavra {input_string} está na gramática")
    else:
        print(f"A palavra {input_string} não está na gramática")
```

basicamente, criamos um dicionário `automatos` que guarda cada geração da iteração e, para cada possível estado dessa geração, calculamos todos os estados que podemos chegar, e colocamos ele na próxima geração.

quando qualquer um desses autômatos aceitar a palavra (tanto a `input_string` quanto a pilha estão vazias), podemos quebrar o loop e ter certeza que ela foi aceita. exemplificando, se rodarmos:

```
regras = {"S":["E"], "E" : ["E+T", "T"], "T": ["T*F"], "F":["(E)",
↪  "id"]}
simular_automato("id+(id*id+id)",regras)
```

vemos que ele corretamente printará `A palavra id+(id*id+id) está na gramática`.

para achar a derivação exata (e montar por exemplo uma árvore sintática), precisaríamos adicionar alguma maneira de descobrir qual foi a derivação *anterior* a um estado. assim, uma solução seria utilizar a noção de árvores como estrutura de dados, isto é, cada autômato teria um `pai`, que representa o estado que o gerou, e o automato inicial poderia simplesmente ser iniciado com `pai = None`.

6 os problemas

qualquer programador que se preze deve agora estar espumando de raiva com as linhas acima. sim. eu sei. infelizmente, se tentarmos rodar essa função em uma palavra que certamente não está na linguagem, na maior parte dos casos ele não irá terminar (e isso inclui a nossa linguagem de expressões matemáticas $L(G_{exp})$).

a função não irá terminar pois existem linguagens infinitas que sempre podem gerar pelo menos um novo estado simplesmente por recursão. por exemplo, a regra $E \to E + T$ permite que o automato crie uma cadeia infinita $T + T + T \cdots$ e, por mais que a palavra não esteja na linguagem, o autômato não consegue ter certeza, através do que foi feito acima, que de alguma maneira mágica a expressão $T + T + T \cdots$ não se tornará por exemplo $id * + id$.

uma possível solução seria contar o tamanho da pilha e comparar com o tamanho da palavra dada, entretanto ela também tem sérios problemas. primeiramente, existem as transições do tipo $X \to \varepsilon$, então não podemos simplesmente contar as variáveis. portanto, poderíamos contar apenas os símbolos terminais, mas cairíamos em outro problema: existem também transições do tipo $X \to XY$, e isso geraria uma cadeia infinita de Y's sem nenhum terminal, e não há nenhuma maneira clara de ter certeza se essa cadeia infinita de Y's se torne depois alguma coisa, especialmente se existir a regra $Y \to \varepsilon$.

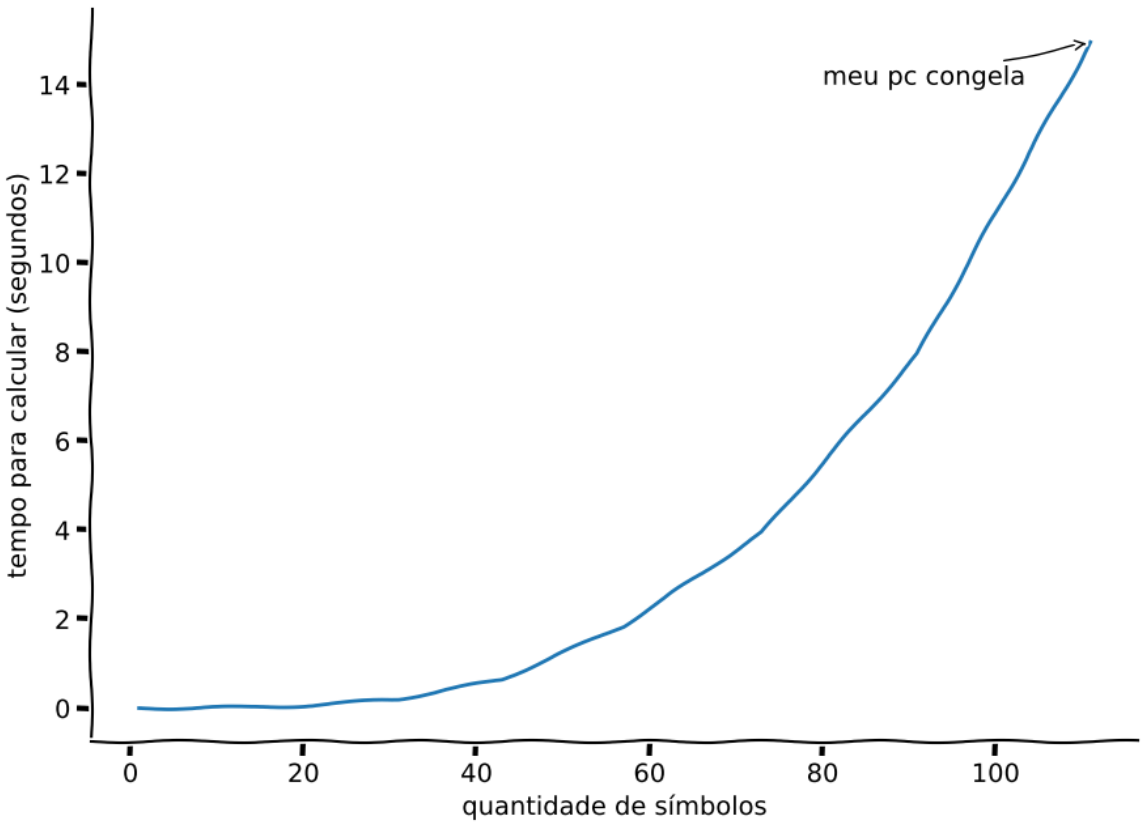
para nossa gramática G_{exp}, limitar a quantidade de computações pela quantidade de terminais é suficiente, mas, de modo geral, essa solução não funcionará pra todas as gramáticas livres de contexto. além disso, temos um inimigo ainda maior: a complexidade temporal.

este algoritmo é terrivelmente ineficiente. devido à natureza explosiva do problema, a quantidade de estados necessários cresce exponencialmente. na verdade, não é difícil mostrar que a quantidade de possíveis árvores de *parsing* é proporcional aos números de Catalan, e se o nosso programa precisa checar caso por caso todas as possíveis a árvores, então uma palavra de tamanho n leva tempo proporcional à:

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

antes que tentem testar, qualquer palavra com mais de 120 caracteres que seja dada como entrada para a função faz com que não só o meu computador trave mas também que o processo seja inevitavelmente terminado pelo sistema operacional.

fazendo um rápido *benchmarking*, grafei o tempo levado para calcular baseado no input:



eu não sei vocês, mas os meus olhos doem só de ver isto.

7 as soluções

primeiro algumas notas e considerações: diferentemente das linguagens regulares, nem todas as linguagens livres de contexto podem ser descritas sem ambiguidade. isso significa que as linguagens livres de contexto *não-ambíguas* são um subconjunto próprio de todas as linguagens livres de contexto. para elas, existem algoritmos que realizam o trabalho em tempo quadrático ou até linear (como por exemplo o algoritmo para analisar gramáticas $LR(k)$ de Knuth[1]), mas muitas vezes eles exigem que a gramática esteja de uma certa forma (por exemplo, que ela não seja recursiva à direita, ou que ela não tenha produções nulas), e requerem muito trabalho para implementar corretamente.

esses algoritmos são extremamente úteis para compiladores analisarem eficientemente linguagens de programação, visto que as gramáticas dessas podem ser manualmente construidas de modo a facilitar a leitura (em seu artigo, Knuth chama essas linguagens especiais de *translatable from left to right*). entretanto, eles não são aplicáveis à qualquer gramática.

a análise de gramáticas ambíguas está especialmente atrelada ao processamento de linguagens naturais, mais especificamente ao processamento de análise da estrutura sintática de frases. para essa tarefa, existem dois algoritmos tradicionais: o algoritmo CYK[2] (desenvolvido independentemente e simultaneamente por 3 autores distintos: Cocke, Younger e Kasami) e o algoritmo de Earley [3].

o algoritmo de CYK é conhecido por ser um *bottom-up parser*, isto é, ele começa da string inicial e constrói todos os casos de baixo pra cima, até chegar na raiz da árvore, que no caso da gramática G_{exp} é o símbolo S. para o algoritmo funcionar, entretanto, ele precisa que a gramática fornecida esteja na forma normal de *Chomsky*, que significa que a gramática só pode ter regras da forma:

$$X \to AB$$
$$X \to a$$

para variáveis X, A, B e terminal a. é um resultado famoso que toda linguagem livre de contexto sem a palavra nula pode ser descrito por uma gramática nessa forma (e inclusive é dado um algoritmo para converter uma gramática arbitrária). ao invés de ser proporcional ao número de catalan, este algoritmo tem complexidade $O(n^3)$, que é anos luz melhor do que o programa proposto inicialmente.

entretanto, o *parser* inventado por Earley é muito mais interessante. não obstante ele aceitar as gramáticas em qualquer forma (isto é, não é necessário converter a gramática para a forma normal de *Chomsky*), ele tem no pior do pior dos casos uma complexidade $O(n^3)$, e para certas gramáticas ele funciona melhor que o próprio algoritmo de CYK. de fato, para gramáticas não ambíguas, o al-

[1] Donald E. Knuth. On the Translation of Languages from Left to Right, https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.361.8867&rep=rep1&type=pdf

[2] Daniel H. Younger. Recognition and Parsing of Context-Free Languages in Time n^{3*}. https://core.ac.uk/download/pdf/82305262.pdf

[3] Jay Earley. An Efficient context-Free Parsing Algorithm. https://lara.epfl.ch/w/_media/compilation/p94-earley.pdf

goritmo é, no pior dos casos, $O(n^2)$, e, para gramáticas $LR(k)$ (descritas por Knuth) ele termina em tempo linear $O(n)$.

de modo geral, a principal diferença entre a abordagem anterior e a proposta por Earley é a de restringir de maneira inteligente quais transições podem ser executadas a cada momento. para isso, ao invés de executar uma transição de cada vez, ele passa a executar todas as possíveis transições que não geram terminais em um único estado, e divide os conjuntos por caracter. isto é, o algoritmo começa analisando todos os estados que podem ser gerados a partir do símbolo inicial da gramática, e só passa para o próximo caracter quando todos os possíveis estados que geram o caracter inicial foram achados.

para resolver o problema das recursões pela direita ou pela esquerda, o algoritmo simplesmente proibe que existam estados iguais dentro do conjunto de estados de um caracter. dessa forma, quando uma regra do tipo $S \rightarrow SX$ é aplicada, ele percebe que o mesmo estado seria adicionado mais de uma vez e diretamente interrompe o loop infinito. dessa forma, é muito mais belo que o algoritmo de CYK e tão eficiente quanto, talvez até melhor.

como este post já está enorme, darei mais detalhes no próximo post, sobre as especificidades e implementações de um Earley *parser*.