3.7 Autômato com Pilha

♦ Classe das LLC

- pode ser associada a um formalismo do tipo autômato
- denominado *Autômato com Pilha*

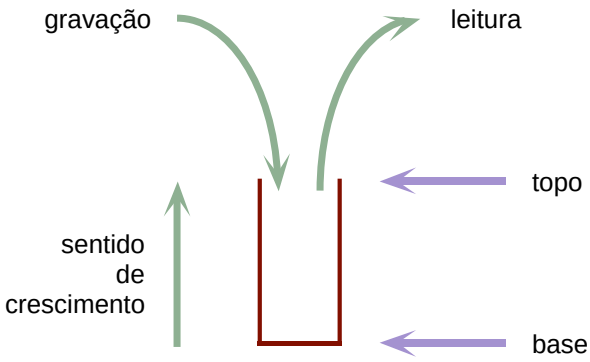
♦ Autômato com Pilha (AP)

- análogo ao Autômato Finito incluindo
  - ∗ não-determinismo
  - ∗ estrutura de pilha
- *não-determinismo* × AP
  - ∗ importante e necessária
  - ∗ aumenta o poder computacional
  - ∗ exemplo: o reconhecimento da linguagem

    $\{ww^r \mid w \text{ é palavra sobre } \{a, b\}\}$

    só é possível por um AP não-determinístico

- *pilha*
  - ∗ memória auxiliar
  - ∗ independente da fita de entrada
  - ∗ não possui limite máximo de tamanho ("infinita")
- estrutura de uma pilha:
  - ∗ último símbolo gravado é o primeiro a ser lido
  - ∗ *base*: é fixa e define o seu início
  - ∗ *topo*: é variável e define a posição do último símbolo gravado



♦ Poder computacional do AP

- muito superior ao do Autômato Finito
- mas ainda é restrito
- exemplo: não reconhece

  $\{ww \mid w \text{ é palavra sobre } \{a, b\}\}$

  $\{a^n b^n c^n \mid n \geq 0\}$

♦ AP × Número de estados

- qq LLC pode ser reconhecida por um AP com somente
  - ∗ *um estado* ou
  - ∗ *três estados*
  - ∗ dependendo da definição
- a pilha é suficiente como única memória
  - ∗ os estados não são necessários para "memorizar" informações passadas
  - ∗ poderiam ser excluídos sem reduzir o poder computacional

Definição do Autômato com Pilha

♦ AP

- duas definições universalmente aceitas
- diferem no critério de parada do autômato

♦ *Estados Finais*

- valor inicial da pilha é vazio
- AP pára aceitando ao atingir um estado final

♦ *Pilha Vazia*

- pilha contém, inicialmente, um símbolo especial denominado símbolo inicial da pilha
- não existem estados finais
- AP pára aceitando quando a pilha estiver vazia

♦ As duas definições são equivalentes

- possuem o mesmo poder computacional
- é fácil modificar um AP para satisfazer a outra definição
- adotamos o modelo com estados finais

AP ou AP Não-Determinístico

- *Fita*
- *Pilha*
- *Unidade de controle*
  - *Cabeça de fita*
  - *Cabeça da pilha*
- *Programa* ou *Função de Transição*

Fita

- Análoga à do Autômato Finito

Pilha

- memória auxiliar
- pode ser usada livremente para leitura e gravação
- dividida em células
- cada cédula
  - um símbolo de um alfabeto auxiliar
  - pode ser igual ao alfabeto de entrada
- leitura ou gravação
  - sempre na mesma extremidade
  - *topo*
- não possui tamanho fixo e nem máximo
- tamanho corrrente
  - tamanho da palavra armazenada
- valor inicial: vazio

Unidade de Controle

- reflete o estado corrente da máquina
- possui
  - *cabeça de fita*
  - *cabeça de pilha*

Cabeça de Fita

- unidade de leitura
- acessa uma célula da fita de cada vez
- movimenta-se exclusivamente para a direita
- pode-se testar se leu toda a entrada

Cabeça da Pilha

- unidade de *leitura* e *gravação*
- *leitura*
  - move para a direita ("para baixo")
  - acessa um símbolo de cada vez: topo
  - exclui o símbolo lido
  - é possível testar se a pilha está vazia
- *gravação*
  - move para a esquerda ("para cima")
  - é possível armazenar uma palavra composta por mais de um símbolo
  - neste caso, o símbolo do topo é o mais à esquerda da palavra gravada

Programa ou Função de Transição

- programa
  - comanda a leitura da fita
  - comanda a leitura e gravação da pilha
  - define o estado da máquina
- dependendo
  - estado corrente
  - símbolo lido da fita
  - símbolo lido da pilha

  determina
  - novo estado
  - palavra a ser gravada
- movimento vazio
  - análogamente ao Autômato Finito
  - pode mudar de estado sem ler da fita ou da pilha

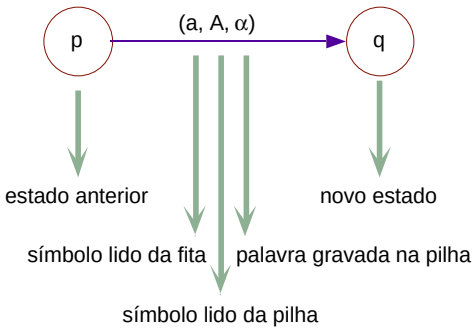
♦ *Definição: Autômato com Pilha*

- *Autômato com Pilha Não-Determinístico* (APN) ou simplesmente *Autômato com Pilha* (AP)
- $M = (\Sigma, Q, \delta, q_0, F, V)$
  - Σ alfabeto de *símbolos de entrada*
  - Q conjunto finito de *estados*
  - δ *função programa* ou *função de transição*
    $\delta: Q \times (\Sigma \cup \{\varepsilon, ?\}) \times (V \cup \{\varepsilon, ?\}) \rightarrow 2^{Q \times V^*}$
    função parcial
  - q_0 *estado inicial* do autômato tq $q_0 \in Q$
  - F conjunto de *estados finais* tq $F \subseteq Q$
  - V *alfabeto auxiliar* ou *alfabeto da pilha*

♦ Características da Função Programa

- a função pode não ser total
- "?" indica teste de
  - toda palavra de entrada lida
  - pilha vazia
- "ε"
  - leitura: movimento vazio da fita ou da pilha
  - gravação: nenhuma gravação é realizada na pilha
- Exemplo: $\delta(p, ?, \varepsilon) = \{(q, \varepsilon)\}$
  - no estado p
  - se a entrada foi completamente lida
  - não lê da pilha
  - assume o estado q
  - não grava na pilha

- representação como um grafo direto



♦ *Processamento* de um AP, para uma *palavra* de entrada

- sucessiva aplicação da função programa até ocorrer uma condição de parada
- é possível que um AP nunca atinja uma condição de parada
  - ciclo ou "loop" infinito
  - exemplo: empilha e desempilha um mesmo símbolo indefinidamente, sem ler da fita

♦ Parada de um AP

- pára aceitando
  - um dos caminhos alternativos assume um estado final
- pára rejeitando
  - todos os caminhos alternativos rejeitam
- loop infinito
  - pelo menos um caminho alternativo está em "loop"
  - os demais rejeitam ou também estão em "loop" infinito

♦ Notações

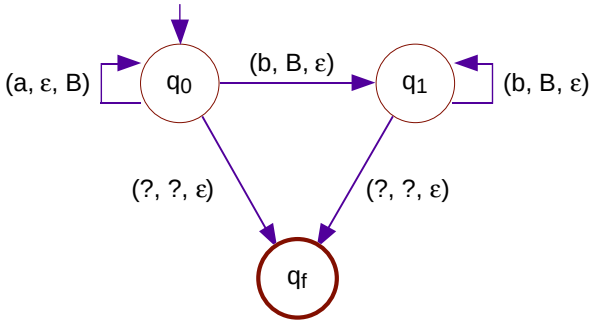
- **ACEITA(M)** ou **L(M)**
  - conj das palavras de Σ^* aceitas por M
- **REJEITA(M)**
  - conj das palavras de Σ^* rejeitadas por M
- **LOOP(M)**
  - conj das palavras de Σ^* para as quais M fica processando indefinidamente

♦ As seguintes afirmações são verdadeiras

- $ACEITA(M) \cap REJEITA(M) \cap LOOP(M) = \varnothing$
- $ACEITA(M) \cup REJEITA(M) \cup LOOP(M) = \Sigma^*$
- o complemento de
  ACEITA(M) é $REJEITA(M) \cup LOOP(M)$
  REJEITA(M) é $ACEITA(M) \cup LOOP(M)$
  LOOP(M) é $ACEITA(M) \cup REJEITA(M)$

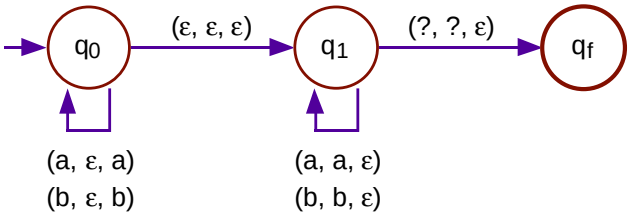
♦ Exemplo: $\{a^n b^n \mid n \geq 0\}$

- um AP determinístico
- $M_1 = (\{a, b\}, \{q_0, q_1, q_f\}, \delta_1, q_0, \{q_f\}, \{B\})$
  - $\delta_1 (q_0, a, \varepsilon) = \{(q_0, B)\}$
  - $\delta_1 (q_0, b, B) = \{(q_1, \varepsilon)\}$
  - $\delta_1 (q_0, ?, ?) = \{(q_f, \varepsilon)\}$
  - $\delta_1 (q_1, b, B) = \{(q_1, \varepsilon)\}$
  - $\delta_1 (q_1, ?, ?) = \{(q_f, \varepsilon)\}$

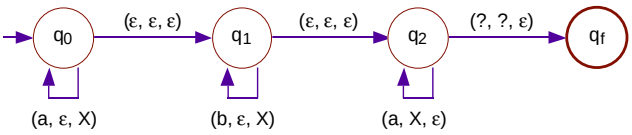


♦ Exemplo: $\{ww^r \mid w \in \{a, b\}^*\}$

- um AP não-determinístico



♦ Exemplo: $\{a^n b^m a^{n+m} \mid n \geq 0, m \geq 0\}$



AP × LLC

♦ A classe das linguagens reconhecidas pelos AP é igual à classe das LLC

♦ Outras conclusões

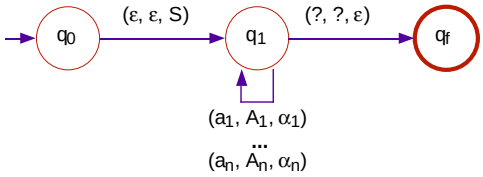
- para qualquer GLC, existe um AP que reconhece a linguagem gerada e sempre pára
- construção de um AP a partir de uma GLC
  - simples e imediata
- qualquer LLC pode ser reconhecida por um AP com somente um estado de controle lógico
  - a facilidade de memorização de informações através de estados (como nos autômatos finitos) não aumenta o poder computacional

♦ ***Teorema*. Se L é uma LLC, então existe M, AP tq ACEITA(M) = L**

♦ **Prova**

- suponha que a palavra vazia não pertence à L
- AP a partir de uma gramática na FNG
  - ∗ produções da forma $A \rightarrow a\alpha$, α palavra de variáveis
  - ∗ o AP simula a derivação mais à esquerda
    lê o símbolo a da fita
    lê o símbolo A da pilha
    empilha a palavra de variáveis α

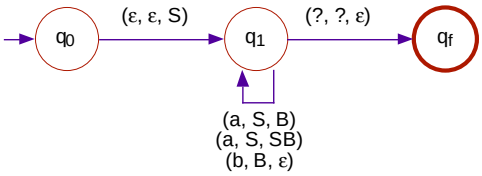
- AP M a partir da gramática G = (V, T, P, S)
  - ∗ G' = (V', T', P',S), é G na FNG
  - ∗ M = (T', {q_0, q_1, q_f}, δ, q_0, {q_f}, V')
    $\delta(q_0, \varepsilon, \varepsilon) = \{(q_1, S)\}$
    $\{(q_1, \alpha) \mid A \rightarrow a\alpha \in P'\}$
    $\delta(q_1, ?, ?) = \{(q_f, \varepsilon)\}$



- demonstração de ACEITA(M) = GERA(G')
  - ∗ indução no número de movimentos de M (ou derivações de G')
  - ∗ sugerida como exercício
  - ∗ como o autômato pode ser modificado para tratar a palavra vazia?

♦ **Exemplo $L = \{a^n b^n \mid n \geq 1\}$**

- FNG
  G = ({S, B}, {a, b}, P, S), onde
  P = {S → aB | aSB, B → b}
- AP
  M = ({a, b}, {q_0, q, q_f}, δ, q_0, {q_f}, {S, B})



♦ **os 2 teoremas que seguem são corolários do anterior**

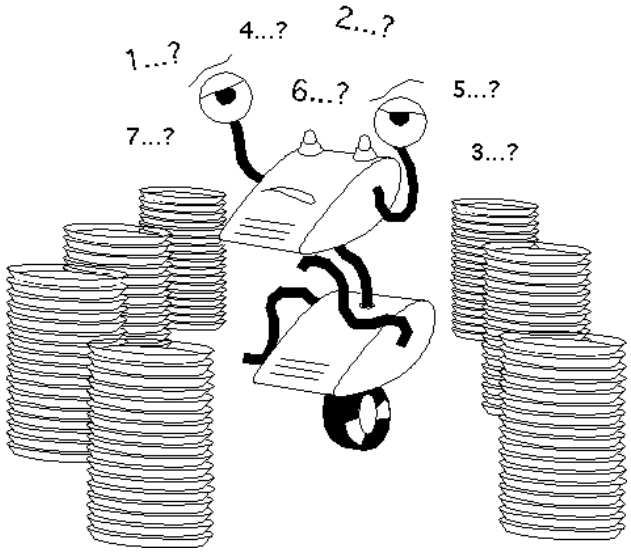
♦ ***Teorema*. Se L é uma LLC, então:**

a) existe M, AP que aceita por estado final, com somente 3 estados tq ACEITA(M) = L
b) existe M, AP que aceita por pilha vazia, com somente um estado tq ACEITA(M) = L

♦ **Portanto**

- o uso dos estados como "memória" não aumenta o poder de reconhecimento dos AP relativamente às LLC

- ***Teorema*. Se L é uma LLC, então existe M, AP tal que**
  - ACEITA(M) = L
  - REJEITA(M) = Σ^* - L
  - LOOP(M) = $\varnothing$.
- **Ou seja**
  - para qualquer LLC existe um AP que sempre pára para qualquer entrada (por que?)
- ***Teorema*. Se L é aceita por um AP, então L é LLC**
  - não será demonstrado



de Pilhas × Poder Computacional

de Pilhas × Poder Computacional

- **Autômato com Pilha**
  - modelo adequado para estudos
    - aplicados
    - formais
  - estrutura de pilha
    - adequada para implementação em computadores
  - poucas modificações sobre a definição do AP
    - determinam significativas alterações no seu poder computacional
  - assim, os principais estudos de linguagens e computabilidade
    - podem ser desenvolvidos usando exclusivamente o AP
    - variando o número de pilhas
    - com ou sem a facilidade de não-determinismo

- **Principais variações**
  - Autômato com Pilha, *sem* usar a estrutura de *pilha*
  - Autômato com Pilha *Determinístico*
  - Autômato com Pilha *Não-Determinístico*
  - Autômato com *Duas Pilhas*
  - Autômato com *Mais de Duas Pilhas*
- **Autômato com Pilha, sem usar a estrutura de pilha**
  - estados
    - única forma de memorizar informações passadas
  - AP sem usar a pilha
    - muito semelhante ao Autômato Finito
  - Classe das Linguagens aceitas por AP sem pilha
    - com ou sem não-determinismo
    - é igual a Classe das Linguagens Regulares
    - exercício

♦ AP Determinístico - APD

- aceita um subconjunto próprio das LLC
  - *Linguagens Livres do Contexto Determinísticas* - *LLCD*
- LLCD inclui muitas das linguagens aplicadas em informática, com destaque para as de programação
- a implementação de um APD
  - simples e eficiente
  - facilita o desenvolvimento de processadores de linguagens
- é possível definir um tipo de gramática que gera exatamente a Classe das LLCD
  - não são restrições simples sobre a definição geral de gramática
- é fechada para a operação de complemento
- *não* é fechada para as operações de
  - união
  - intersecção
  - concatenação

♦ AP (Não-Determinístico)

- a classe das linguagens reconhecida pelo AP é exatamente a Livre do Contexto

♦ Autômato com Duas Pilhas - A2P

- é equivalente, em termos de poder computacional, à Máquina de Turing
- assim, se existe um algoritmo para resolver um problema (por exemplo, reconhecer uma determinada linguagem), então este algoritmo pode ser expresso como um A2P
- a facilidade de não-determinismo não aumenta o poder computacional do A2P

♦ Aut. com Mais de Duas Pilhas - AnP

- o poder computacional é equivalente ao do
- ou seja
  - se um problema é solucionado por um AnP
  - então o mesmo problema pode ser solucionado por um A2P

3.8 Propriedades das LLC

♦ As LLC são mais gerais que as LR

- mas ainda são relativamente restritas
- é fácil definir linguagens que *não* são LLC
  - $\{ww \mid w \text{ pertence a } \{a, b\}^*\}$
  - $\{a^n b^n c^n \mid n \geq 0\}$.

♦ Assim

- como determinar se uma linguagem é LLC?
- a Classe das LLC é fechada para operações como
  - união?
  - intersecção?
  - concatenação?
  - complemento ?
- como verificar se uma LLC é
  - infinita?
  - finita (ou até mesmo vazia)?

Investigação se é LLC

♦ Prova de que uma linguagem é LLC

- é suficiente expressá-la usando os formalismos
  - Gramática Livre do Contexto
  - Autômato com Pilha

♦ Prova de que uma linguagem *não* é LLC

- necessita ser realizada caso a caso
- "lema do bombeamento" para as LLC

♦ Lema. Bombeamento das LLC

- Se L é uma LLC, então:
  - existe uma constante n tal que,
  - para qq $w \in L$ onde $|w| \geq n$,
  - w pode ser definida como $w = uxvyz$ onde $|xvy| \leq n$, $|xy| \geq 1$ e,
  - para todo $i \geq 0$, $ux^ivy^iz \in L$.

♦ Para $w = uxvyz$, tem-se que

- ou x ou y pode ser a palavra vazia
- mas não ambas

♦ Prova

- uma forma de demonstrar o lema é usando gramáticas na Forma Normal de Chomsky
  - se a gramática possui s variáveis
  - pode-se assumir que $n = 2^s$
- o lema não será demonstrado

♦ Exemplo. $L = \{a^n b^n c^n \mid n \geq 0\}$

- prova
  - usa o bombeamento
  - é por absurdo
- Suponha que L é LLC
  - então existe uma GLC na FNC com s variáveis que gera L
  - sejam $r = 2^s$ e $w = a^r b^r c^r$
- pelo bombeamento
  - w pode ser definida como $w = uxvyz$
  - $|xvy| \leq r$
  - $|xy| \geq 1$
  - para qq $i \geq 0$, $ux^ivy^iz \in L$
- *absurdo!!!*, pois como $|xvy| \leq r$
  - xy *não* possui símbolos a e c
  - qq ocorrências de a e c estão separadas por, pelo menos, r ocorrências de b
  - aplicação do bombeamento: desbalanceamento!

Operações sobre LLC

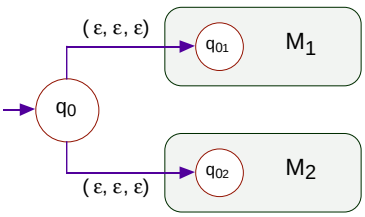
♦ *Teorema*: As LLC são fechadas p/

- união
- concatenação

♦ Prova: *União*

- demonstração é baseada em AP (GLC: exercício)
- suponha L_1 e L_2, LLC. Então, existem
  $M_1 = (\Sigma_1, Q_1, \delta_1, q_{0_1}, F_1, V_1)$ e
  $M_2 = (\Sigma_2, Q_2, \delta_2, q_{0_2}, F_2, V_2)$
  tq $ACEITA(M_1) = L_1$ e $ACEITA(M_2) = L_2$
- seja

$M_3 = (\Sigma_1 \cup \Sigma_2, Q_1 \cup Q_2 \cup \{q_0\}, \delta_3, q_0, F_1 \cup F_2, V_1 \cup V_2)$



- claramente, M_3 reconhece $L_1 \cup L_2$

♦ Prova: *Concatenação*

- demonstração é baseada em GLC (AP: exercício)
- suponha L_1 e L_2, LLC. Então, existem
  $G_1 = (V_1, T_1, P_1, S_1)$ e
  $G_2 = (V_2, T_2, P_2, S_2)$
  tq $GERA(G_1) = L_1$ e $GERA(G_2) = L_2$
- seja

$G_3 = (V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \rightarrow S_1S_2\}, S)$

- a única produção de S é $S \rightarrow S_1S_2$
- qq palavra terá, como
  - prefixo, uma palavra de L_1
  - sufixo, uma palavra de L_2

♦ O próximo teorema mostra que a Classe das LLC *não* é fechada para

- intersecção
- complemento

♦ Aparentemente, é uma contradição

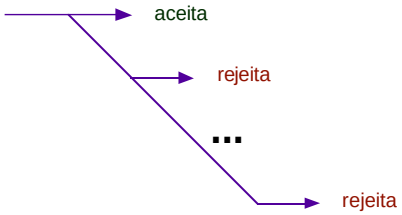
- já foi verificado que
  - se L é LLC, então existe M, AP tal que ACEITA(M) = L e REJEITA(M) = L'
  - ou seja, M é capaz de *rejeitar* qualquer palavra que não pertença à L
- o próximo teorema mostra que
  - se L é LLC, não implica que L' também é LLC
  - ou seja, *não* se pode afirmar que existe um AP que *aceite* L'

♦ Assim

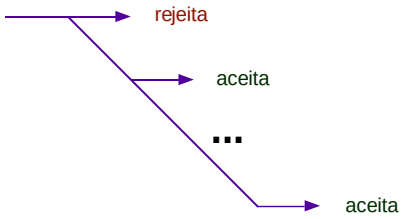
- é perfeitamente possível *rejeitar* o complemento de uma LLC
- embora nem sempre seja possível *aceitar* o complemento

♦ Uma explicação intuitiva

- um AP não-determinista
  - aceita se pelo menos um dos caminhos alternativos aceita



- inverção de aceita por rejeita e vice-versa
  - a condição continua sendo de aceitação



♦ Portanto

- considerando a facilidade de não-determinismo
  - o fato de existir um AP capaz de rejeitar o complemento de uma linguagem
  - não implica que existe um AP capaz de aceitar o mesmo complemento

♦ *Teorema*: A Classe das LLC *não* é fechada para as operações

- intersecção
- complemento

♦ Prova: *Intersecção*

- contra- exemplo
- sejam
  - $L_1 = \{a^n b^n c^m \mid n \geq 0 \text{ e } m \geq 0\}$ e
  - $L_2 = \{a^m b^n c^n \mid n \geq 0 \text{ e } m \geq 0\}$
- é fácil mostrar que L_1 e L_2 são LLC
- entretanto
  - $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$
  - não é LLC

♦ Prova: *Complemento*

- considerando que
  - *não* é fechada para a intersecção
  - intersecção pode ser representada em termos de união e complemento
  - é fechada para a união
- *não* pode-se afirmar que é fechada para o complemento

Investigação se uma LLC é Vazia, Finita ou Infinita

♦ **Teorema. Se L é LLC, então é possível determinar se L é**

- vazia
- finita
- infinita

♦ **Prova: *Vazia***

- seja G = (V, T, P, S), GLC tq GERA(G) = L
- seja G'= (V', T', P', S) equivalente a G, eliminando os símbolos inúteis
- se P' for vazio, então L é vazia

♦ **Prova: *Finita e Infinita***

- seja G = (V, T, P, S) uma GLC tq GERA(G) = L
- seja G' = (V', T', P', S) equivalente a G
  - Forma Normal de Chomsky
  - ($A \rightarrow a$ ou $A \rightarrow BC$)
- considere somente as produções da forma $A \rightarrow BC$
- se existe A tq
  - $A \rightarrow BC$ (A no lado esquerdo)
  - $X \rightarrow YA$ ou $X \rightarrow AY$ (A no lado direito)

  e se existe um ciclo em A do tipo $A \Rightarrow^{+} \alpha A \beta$ então
  - A é capaz de gerar palavras de qq tamanho
  - a linguagem é infinita
- caso *não* exista tal A, então a linguagem e finita

3.9 Algoritmos de Reconhecimento

♦ **Uma palavra pertence ou não a uma linguagem?**

- uma das principais questões relacionadas com o estudo de Linguagens Formais

♦ **"Dispositivo" de reconhecimento**

- pode ser especificado como um
  - modelo de autômato ou
  - algoritmo implementável em computador
- em qualquer caso, é importante determinar
  - "quantidade de recursos" necessários
  - por exemplo: tempo e espaço
- objetivo
  - gerar dispositivos de reconhecimento válidos para qualquer linguagem dentro de uma classe
  - algoritmos apresentados: específicos para LLC

♦ **Algoritmos de reconhecimento**

- construídos a partir de uma GLC
- reconhecedores que usam Autômato com Pilha
  - muito simples
  - em geral, ineficientes
  - tempo de processamento é proporcional a $k^{|w|}$ (w - entrada; k depende do autômato)
  - não são recomendáveis para entradas de tamanhos consideráveis
- existe uma série de algoritmos bem mais eficientes
  - tempo de processamento proporcional a $|w|^3$
  - ou até um pouco menos
  - não é provado se o tempo proporcional a $|w|^3$ é efetivamente necessário para que um algoritmo genérico reconheça LLC

Tipos de reconhecedores

- *Top-Down* ou *Preditivo*
- *Bottom-Up*

Top-Down ou *Preditivo*

- constrói uma árvore de derivação para a entrada
  - a partir da raiz (símbolo inicial da gramática)
  - gera os ramos em direção às folhas (símbolos terminais que compõem a palavra

Bottom-Up

- basicamente, o oposto do Top-Down
- parte das folhas e constrói a árvore de derivação em direção à raiz

AP como Reconhecedor

Construção de reconhecedores usando AP

- relativamente simples e imediata
- existe uma relação quase direta entre
  - produções da gramática
  - transições do AP
- algoritmos
  - tipo Top-Down
  - simulam a derivação mais à esquerda da palavra a ser reconhecida
- não-determinismo
  - testa as diversas produções alternativas da gramática para gerar os símbolos terminais

AP a Partir de uma GLC na FNG

Foi visto que

- qualquer LLC pode ser especificada como um AP
- existe um algoritmo que define um AP a partir de uma Gramática na FNG

Cada produção na FNG gera exatamente um terminal

- w é gerada em $|w|$ etapas de derivação
- entretanto
  - cada variável pode ter mais de uma produção
  - é necessário testar as diversas alternativas
- número de passos para reconhecer w
  - proporcional a $k^{|w|}$ onde k depende do AP
  - aproximação para k: metade do número médio de produções associadas às diversas variáveis
- portanto
  - tempo: proporcional ao expoente em $|w|$
  - pode ser muito ineficiente para entradas longas

AP Descendente

Forma alternativa de construir um AP a partir de uma GLC

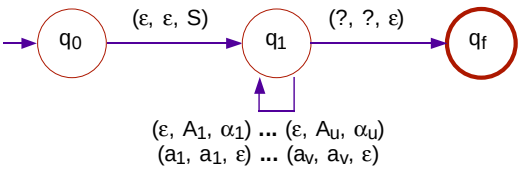
- algoritmo igualmente simples
- mesmo nível de eficiência
- construção:
  - gramática sem recursão à esquerda
  - simula a derivação mais à esquerda

Algoritmo

- inicialmente, empilha o símbolo inicial
- sempre que existir uma variável no topo da pilha, substitui (de forma não-determinística) por todas as produções da variável
- se o topo da pilha for um terminal, verifica se é igual ao próximo símbolo da entrada

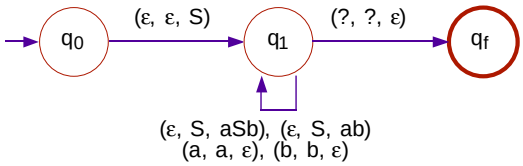
♦ Construção de um Autômato com Pilha Descendente

- seja $G = (V, T, P, S)$
  - GLC
  - sem recursão à esquerda
- $M = (T, \{q_0, q_1, q_f\}, \delta, q_0, \{q_f\}, V \cup T)$, onde
  - $\delta(q_0, \varepsilon, \varepsilon) = \{(q_1, S)\}$
  - $\delta(q_1, \varepsilon, A) = \{(q_1, \alpha) \mid A \rightarrow \alpha \in P\}$
  - $\delta(q_1, a, a) = \{(q_1, \varepsilon)\}$
  - $\delta(q_1, ?, ?) = \{(q_f, \varepsilon)\}$



♦ *Exemplo.* $L = \{a^n b^n \mid n \geq 1\}$

- $G = (\{S\}, \{a, b\}, P, S)$, onde
  $P = \{S \rightarrow aSb \mid ab\}$ (sem recursão à esquerda)
- $M = (\{a, b\}, \{q_0, q_1, q_f\}, \delta, q_0, \{q_f\}, \{S\})$



Algoritmo de Cocke-Younger-Kasami

♦ Cocke-Younger-Kasami (CYK)

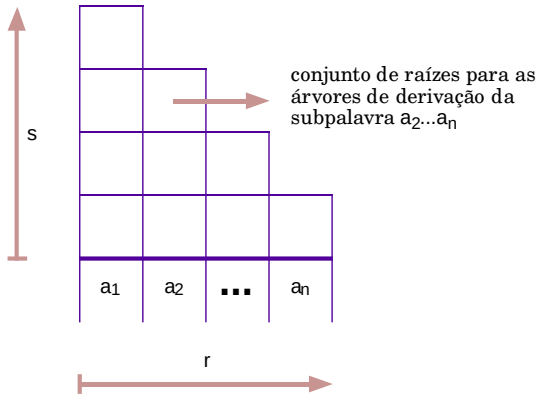
- desenvolvido independentemente por
  - Cocke, Younger e Kasami
  - em 1965
- construído sobre uma gramática na FNC
- tipo *bottom-up*
- gera todas as árvores de derivação da entrada
- tempo de processamento proporcional a $|w|^3$

♦ Algoritmo

- construção de uma tabela triangular de derivação
- cada célula representa o conjunto de raízes que pode gerar a correspondente sub-árvore

♦ Algoritmo de CYK

- $G = (V, T, P, S)$ na FNC onde
  $T = \{a_1, a_2, ..., a_t\}$
- V_{r_s} representa as células da tabela triangular de derivação (suponha $w = a_1 a_2 ... a_n$)



a) *Variáveis q. geram terminais diretamente* $A \rightarrow a$

```
para r variando de 1 até n
faça  Vr1 = {A | A → ar ∈ P}
```

b) *Produção que gera duas variáveis* $A \rightarrow BC$

```
para s variando de 2 até n
faça para r variando de 1 até n-s+1
     faça Vrs = ∅
          para k variando de 1 até s-1
          faça Vrs = Vrs ∪
               {A | A → BC ∈ P, B ∈ Vrk e
               C ∈ V(r+k)(s-k)}
```

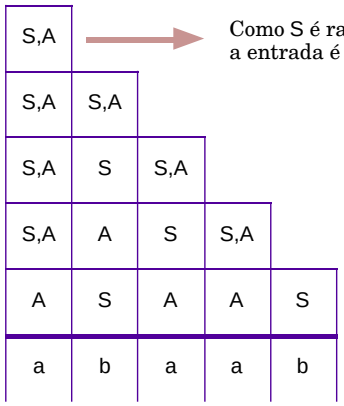
- Note-se que:
  - limite de iteração para r é $n - s + 1$, pois a tabela de derivação é triangular
  - os vértices V_{r_k} e $V_{(r+k)_{(r-k)}}$ são as raízes das sub-árvores de V_{r_s}
  - se uma célula for vazia, significa que esta célula não gera qualquer sub-árvore

c) *Condição de aceitação da entrada*

- se o símbolo inicial pertence ao vértice V_{1_n} (raiz da árvore de derivação de toda palavra), então a entrada é aceita

♦ *Exemplo.*

- $G = (\{S, A\}, \{a, b\}, P, S)$, onde
  $P = \{S \rightarrow AA \mid AS \mid b,\ A \rightarrow SA \mid AS \mid a\}$
- tabela triangular de derivação para abaab



Algoritmo de Early

♦ Early

- desenvolvido em 1968
- considerado o mais rápido algoritmo de reconhecimento conhecido para GLC
- tempo de processamento
  - proporcional a $|w|^3$
  - para gramáticas não-ambíguas, pode ser proporcional a $|w|^2$
  - para muitas gramáticas de interesse prático, o tempo é proporcional a $|w|$

♦ Algoritmo

- tipo *top-down*
- a partir de uma GLC qualquer
- executa sempre a derivação mais à esquerda
- cada ciclo gera um terminal
  - comparado com o símbolo da entrada
- comparação com sucesso
  - construção de um conjunto de produções que pode gerar o próximo símbolo

♦ Algoritmo de Early

- $G = (V, T, P, S)$ uma GLC qualquer
- $w = a_1 a_2 ... a_n$ palavra a ser reconhecida
- símbolo "."
  - é usado como um marcador
  - em cada produção antecede a posição que será analisada na tentativa de gerar o próximo terminal
- sufixo "/u"
  - adicionado a cada produção
  - indica o u-ésimo ciclo em que esta produção passou a ser considerada

a) *Construção do primeiro conjunto de produções*

- produções que partem de S
- produções que podem ser aplicadas em sucessivas derivações mais à esquerda a partir de S

```
D0 = ∅
para toda produção S → α ∈ P                (1)
faça D0 = D0 ∪ {S → .α/0}
repita                                      (2)
      para toda produção A → .Bβ/0 ∈ D0
      faça para toda produção B → ϕ ∈ P
            faça D0 = D0 ∪ {B → .ϕ/0}
até não ocorrerem mais inclusões
```

(1) produções que partem de S

(2) as produções que podem ser aplicadas em derivação mais à esquerda a partir de S

b) *Construção dos demais conjuntos de produção*

- $n = |w|$ conjuntos de produção a partir de D_0
- ao gerar o símbolo a_r de w constrói D_r: produções que podem gerar o símbolo a_{r+1}

```
para r variando de 1 até n                  (1)
faça Dr = ∅;
     para toda A → α.arβ/s ∈ Dr-1           (2)
     faça Dr = Dr ∪ {A → αar.β/s};
     repita
          para toda A → α.Bβ/s ∈ Dr         (3)
          faça para toda B → ϕ ∈ P
               faça Dr = Dr ∪ {B → .ϕ/r}
     para toda A → α./s de Dr               (4)
     faça para toda B → β.Aϕ/k ∈ Ds
          faça Dr = Dr ∪ {B → βA.ϕ/k}
     até não ocorrerem mais inclusões
```

(1) cada ciclo gera um conjunto de produções D_r

(2) gera o símbolo a_r

(3) produções que podem derivar o próximo símbolo

(4) uma subpalavra de w foi reduzida à variável A

* inclui em D_r produções de que referenciaram .A

c) *Condição de aceitação da entrada*

- se uma produção $S \rightarrow \alpha./0$ pertence a D_n, então a palavra w de entrada foi aceita
- $S \rightarrow \alpha./0$ é uma produção que
  * parte do símbolo inicial S
  * foi incluída em D_0 ("/0")
  * todo o lado direito da produção foi analisado com sucesso ("." está no final de α)

♦ Otimização das etapas a) e b)

- ciclos `repita-até`
  * restritos exclusivamente às produções recentemente incluídas em D_r ou em D_0 ainda não-analisadas

♦ *Exemplo.* Gramática análoga à definição de "expressão simples" do PASCAL

G = ({E, T, F}, {+, *, [,], x}, P, E), onde
P = {E → T | E+T, T → F | T*F, F → (E) | x}
reconhecimento da palavra x*x

D_0:

E → .T/0	produções que partem
E → .E+T/0	do símbolo inicial
T → .F/0	produções que podem ser aplicadas
T → .T*F/0	em derivação mais à esquerda
F → .(E)/0	a partir do símbolo inicial
F → .x/0	

D_1: reconhecimento de "x" em x*x

F → x./0	x foi reduzido à F
T → F./0	inclui as produções de D_0 (F → x./0)
T → T.*F/0	que referenciaram .F direta ou
E → T./0	indiretamente, movendo "."
E → E.+T/0	um símbolo para a direita

D_2: reconhecimento de "$*$" em $x*x$

$T \to T*.F/0$	gerou $*$; o próximo será gerado por F
$F \to .(E)/2$	inclui as produções de P que podem
$F \to .x/2$	gerar o próximo terminal a partir de F

D_3: reconhecimento de "x" em $x*x$

$F \to x./2$	x foi reduzido à F
$T \to T*F./0$	incluído de D_2 ($F \to x./2$); a entrada foi reduzida à T;
$E \to T./0$	incluído de D_0 ($T \to T*F./0$); a entrada foi reduzida à E;
$T \to T.*F/0$	incluído de D_0 ($T \to T*F./0$);
$E \to E.+T/0$	incluído de D_0 ($E \to T./0$).

$w = x*x$ foi reduzida ao símbolo inicial E

- $E \to T./0$ pertence a D_3
- a entrada foi aceita