2 – Análise Léxica

Linguagens e Programação

Ana Madureira
Engenharia Informática
Ano Letivo: 2021/2022

Fontes:
1. Compiladores Princípios e práticas, Keneth C.Louden, Thomson, 2004.
   Cap. 1 Introdução
   Cap. 2 Processo de Varrimento, Análise léxica

2. Compiladores - Princípios , Técnicas e Ferramentas, Alfred V. Aho, Monica S. Lam e Ravi Sethi, Pearson, 2ª edição, 2007.
   Cap. 3 Análise Léxica

3. Processadores de Linguagens – da concepção à implementação, Rui Gustavo Crespo. IST Press.1998.
   Cap. 2 Definição de Linguagens

1

Análise Léxica

- Agrupa sequências de caracteres em *tokens* - as unidades básicas da sintaxe
- A sequência de caracteres que formam um *token* chama-se lexema *tokens* típicos: constantes (literais), id's, operadores, palavras--chave, etc.
- Elimina sequências de caracteres inúteis (espaços, tabs, LF, CR, comentários)

Recorre-se de:
- **Expressões Regulares** - representam padrões de cadeias de caracteres
- **Autómatos Finitos –** forma matemática de descrever tipos particulares de algoritmos , para descrever o processo de reconhecimento de padrões em cadeias de caracteres.

2

Funções da Análise Léxica

… | a | [| i | n | d |] | | = | | 1 | 0 | | * | | x | 1 | | + | | y | 1 | …

Sequência de *tokens*:

… | a | [| ind |] | = | 10 | * | x1 | + | y1 | …

- Agrupar os caracteres individuais do programa fonte em "palavras", designadas por ***tokens***, que são as unidades básicas das linguagens de programação, e que serão usados pelo analisador sintáctico (*parser*)
  - ignorar alguns tipos de caracteres (espaços, *tabs*, LF, CR) e os comentários
  - produzir mensagens de erro quando algum caracter ou sequência de caracteres não são reconhecidos
  - localizar os *tokens* no texto fonte (nº de linha, nº de coluna)
  - actuar como pré-processador (em certos compiladores)

3

Tokens e Lexemas

Token – representa um conjunto de cadeias de entrada possível

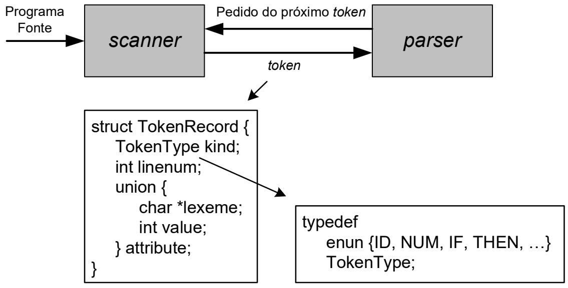
Lexema – é uma determinada cadeia de entrada associada a um *token*

Token	**Alguns lexemas possíveis**
for	for
if	if
while	while
número	1089, 142857, .0, 3.14159
identificador	i, j, contador, nome_aluno
abr_par	(

4

Analisador léxico

O analisador léxico (*scanner*) actua como uma subrotina do analisador sintáctico (*parser*):



Especificação do analisador léxico

- Definição do conjunto de *tokens*
  - Descrição genérica (através das suas propriedades) das entidades constituintes da linguagem fonte
  - Dependente da gramática da linguagem fonte
- Exemplos de algumas dessas entidades:
  - Palavras-chave da linguagem (p. ex: if then while do, etc.)
  - identificadores (de variáveis, procedimentos, tipos, campos de registos, etc.)
  - operadores (aritméticos, lógicos, booleanos, etc.)
  - constantes (também chamadas literais - inteiros, reais, caracteres, strings, etc.)
  - sinais de pontuação (; , : [(.. etc.)
- Definição das regras de formação dos *tokens*
  - Exemplo: identificador - sequência de letras ou dígitos, iniciada por letra

Especificação dos *tokens*

- Os *tokens* são geralmente especificados utilizando expressões regulares
- As expressões regulares são regras bem definidas que geram um conjunto de sequências de caracteres (*strings*)
- O conjunto de *strings* gerado por uma expressão regular chama-se uma linguagem regular
- As strings geradas por uma expressão regular são construídas com caracteres de um conjunto de símbolos chamado alfabeto (Σ)
  - A linguagem regular gerada por uma expressão regular 'r' denota-se como L(r) (conjunto de strings que são sequências de símbolos de Σ)
  - A linguagem regular vazia (sem qualquer string) nota-se por Φ.
  - A linguagem vazia (Φ) é diferente da linguagem que só contém a string vazia (que se nota por ε) L(ε) = { ε }

7

Operações sobre Expressões Regulares

- São três as operações sobre expressões regulares que fazem parte da sua definição formal (em ordem crescente de prioridade):
  - **Alternativa**: Uma expressão regular da forma s | t, onde s e t são expressões regulares; se r = s | t então $L(r) = L(s) \cup L(t)$
  - **Concatenação**: Uma expressão regular da forma st, onde s e t são expressões regulares; se r = st então $L(r) = L(s)L(t) = \{st \mid s \in L(s) \text{ et } \in L(t)\}$
  - **Fecho de Kleene**: Uma expressão regular da forma s*, onde s é uma expressão regular; se r = s* então L(r) = L(s)*
- **Uso de parêntesis**: Uma expressão regular da forma (s), onde s é uma expressão regular; se r = (s) então L(r) = L(s); os parêntesis servem apenas para modificar a precedência das operações
- o uso dos caracteres | * (e) para designar as operações entre expressões regulares: estes caracteres com significado especial designam-se por **metacaracteres**. Se estes caracteres também pertencerem ao alfabeto Σ então o seu uso como caracteres deve ser distinguido de alguma forma (p. ex. colocando-os entre plicas - '('

8

Exemplos

- **Concatenação**

Se S = {aa, b} e T = {a, bb}   então R = ST = {aaa, aabb, ba, bbb}

- **Fecho de Kleene**

Se S = {a}   então S* = S0 ∪ S1 ∪ S2 ∪ S3 ∪ …

onde S0 = {ε}, S1 = S, S2 = SS, S3 = SSS, …

logo S* = {ε, a, aa, aaa, aaaa, …}

No alfabeto Σ = {a, b, c}
a) escrever uma expressão regular que represente todos as strings que contêm um só b:
(a | c)* b (a | c)*
A expressão gera uma linguagem que
contém por exemplo: b, abc, abaca, ccbaca, cccccb, etc.

b) escrever uma expressão regular que represente todos as strings que contêm no máximo um b:
(a | c)* | (a | c)* b (a | c)*
ou
(a | c)* (b | ε) (a | c)*
Ambas as expressão de cima geram a mesma linguagem que é a que foi especificada por palavras

9

Exemplos

No alfabeto Σ = {a, b, c} escrever uma expressão regular que represente todos as strings que não contêm b's consecutivos.

- Strings que não contêm b's: **(a | c)***
- Strings que têm qualquer coisa a seguir a um b:

  (b (a | c))*
- Combinando os 2 anteriores temos realmente strings sem b's consecutivos:

  ((a | c) * | (b (a | c))*)* = ((a | c) | b (a | c))*

  uma vez que **(r* | s*)* = (r | s)***
- No entanto esta ainda não é a solução para o problema inicial, uma vez que não contempla os strings que terminam em b.
- Para chegar à solução geral basta agora dar essa possibilidade:

  ((a | c) | b (a | c))* (b | ε)

10

Abreviaturas em expressões regulares

- Uma ou mais repetições: r r* pode abreviar-se para r+
- Zero ou uma instâncias (opcional): (r | ε) pode abreviar-se para r?
- Qualquer carácter do alfabeto: (c1 | c2 | …| cn) para todo o alfabeto, pode abreviar-se por .
- Alternativas entre alguns caracteres ou numa gama de caracteres considerando o alfabeto um conjunto ordenado:
  - (a | b | c) pode abreviar-se por [abc]
  - letra = [a-zA-Z]
  - char_identificador = [a-zA-Z_]
- Qualquer carácter do alfabeto excepto alguns:
  - alfabeto - todas as letras minúsculas
  - (qualquer símbolo excepto a|b|c) = [d-z] pode abreviar-se por
  - ~(a | b | c) ou mais simplesmente por ~[abc] ou ainda [^abc]
- Subexpressões:
  - natural = [0-9]+
  - natural_com_sinal = (+ | -)? natural

11

Expressões regulares na definição dos *tokens*

Para a construção de um *analisador léxico* deve-se começar por escrever uma expressão regular que compreenda a definição de todos os *tokens* da linguagem.

Processo:

1. Escrever uma expressão regular para cada classe de token da linguagem
   - r1 = palavra_chave = if | then | while | do | …
   - r2 = identificador = letra (letra | digito)*
   - r3 = inteiro = digito+
   - r4 = espaço_em_branco = (\n | \t | ' ' | comentario)+

   …

   (esta última definição não é *token*, mas também tem de ser reconhecida)

2. Escrever uma expressão regular global (R) que agrupe todas as definições anteriores:

   R = palavra_chave | identificador | inteiro | espaço_em_branco | ...

12

Limitações das Expressões Regulares

- Certos *tokens* são difíceis de especificar numa expressão regular, como os comentários delimitados por 2 caracteres, em ordem inversa

  Ex: comentários em C - delimitados por /* e */ :

  Uma solução: /'*' /* ('*'* [^/ '*'] /*)* '*'+/

- Há certos *tokens* que não são especificáveis com expressões regulares

Ex: Os comentários em Modula-2 podem ser imbricados:

(* isto é um comentário em (* Modula - 2 *) *)

- Certas linguagens exigem que se conheçam vários caracteres para além do fim do *token* que se está a reconhecer (*lookahead*)

  Ex: No FORTRAN as palavras chave não são reservadas nem o espaço em branco é significativo:

  IF (I F .EQ. 0) THENTHEN=1.0 (instruções válidas em FORTRAN)

  DO99I=1,10

  DO99I=1.10

- Nestes casos é necessário usar um *scanner* não totalmente baseado em expressões regulares

13

Propriedades das Expressões Regulares

Sejam u e v expressões regulares. Tem-se que:

1. $u+v=v+u$	**11.** $u^*=(u+\ldots+u^k)^*$, $k\geq 1$
2. $u+u=u$	**12.** $u^*=\varepsilon+u+\ldots+u^{k-1}+u^k u^*$, $k>1$
3. $(u+v)+x=u+(v+x)$	**13.** $u^*u=uu^*$
4. $u\varepsilon=\varepsilon u=u$	**14.** $(u+v)^*=(u^*+v^*)^*=(u^*v^*)^*=(u^*v)^*u^*=u^*(vu^*)^*$
5. $(uv)x=u(vx)$	**15.** $u(vu)^*=(uv)^*u$
6. $u(v+x)=uv+ux$	**16.** $(u^*v)^*=\varepsilon+(u+v)^*v$
7. $(u+v)x=ux+vx$	**17.** $(uv^*)^*=\varepsilon+u(u+v)^*$
8. $\varepsilon^*=\varepsilon$	
9. $u^*=u^*u^*=(u^*)^*=u^*+u^*$	
10. $u^*=\varepsilon+u^*=(\varepsilon+u)^*=(\varepsilon+u)u^*=\varepsilon+uu^*$	

As propriedades das expressões regulares mencionadas acima servem para simplificar expressões regulares e provar a equivalência entre duas expressões regulares.

14

Autómato Finito (AF)

As expressões regulares são convenientes para especificar os *tokens* de uma linguagem.

Que formalismo usar para que possa ser facilmente implementado num programa?

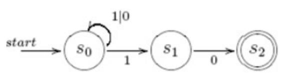
Modelo matemático conhecido por **autómato finito** (AFD), definido por um vetor (S, Σ, s_0, F, δ):

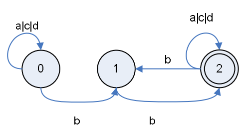
- o alfabeto de entrada Σ
- um conjunto finito de estados S ≠ ∅
- um conjunto de transições entre estados para um carácter de Σ (ou ε) do tipo "do estado x transita-se para o estado y com a entrada 'c' " δ : S x Σ → S (função parcial)
- um conjunto de estados de aceitação ou finais F (F ⊆ S)
- um estado inicial s_0 ∈ S

Um autómato finito pode ser representado por uma **tabela** ou, mais frequentemente por um **grafo**, em que os nós são estados e os ramos transições, etiquetados com os caracteres que determinam essa transição; os estados inicial e finais deverão estar marcados

Representação Gráfica

$\rightarrow s_0$	Estado inicial s_0
s_1	Estado s_1
$\xrightarrow{0}$	Transição que consome o símbolo 0
s_3	Estado final s_3





Representação de Autómatos finitos

Podem ser representados por tabelas ou grafos:

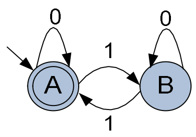
AFD = (S, Σ, s_0, F, δ)

	Ⓐ	B
0	A	B
1	B	A

← estados ∈ S

↑ caracteres de entrada ∈ ∑

Conjunto de transições *d* definido na tabela:
d(A, 0) = A
d(A, 1) = B
d(B, 0) = B
d(B, 1) = A



Por exemplo, a cadeia 00111 não reconhecida pelo autómato, enquanto que a cadeia 0011 é reconhecida porque a análise termina num estado final

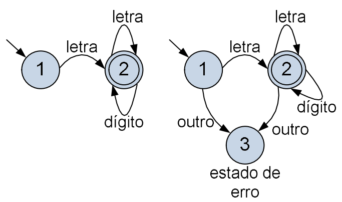
Esboço de um programa para implementação de um AFD:

```
CASE estado of
   A: CASE entrada of
      0: …
         estado = A;
         END;
      1: …
         estado = B;
         END;
   B: CASE entrada of
      0: …
         estado = B;
         END;
      1: …
         estado = A;
         END;
END;
```

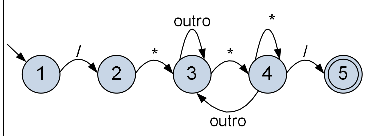
17

Autómatos finitos (exemplos)

<identificador> = <letra>(<letra>|<dígito>)*
<letra> = [a-zA-Z]
<dígito> = [0-9]



<comentário> = /'*'<qualquer>*'*'/



Reconhecimento de uma *string* com n caracteres por um AFD:

Começando no estado inicial, para cada caracter da *string*, o autómato executa exactamente uma transição de estado para um próximo estado, seguindo o ramo etiquetado com esse caracter.

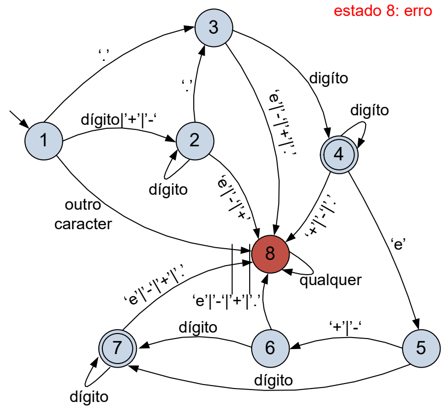
Se após as n transições o autómato estiver num estado final, então aceita a *string*.

Se não estiver num estado final ou se em qualquer altura não existir um ramo etiquetado com o caracter actual da *string* da entrada, então essa *string* é rejeitada.

18

Autómatos finitos (exemplos)

<número real> = ('+'|'-')?<dígito>*'.'<dígito>+(e('+'|'-')?<dígito>+)?

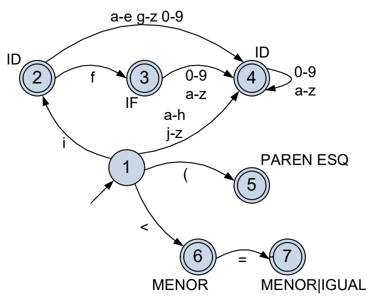



```
function realnum: boolean;
var   entrada: char;
      estado: integer;
begin
      estado:=1; read(entrada);
      while (entrada>='0' and entrada<='9')
or
            (entrada='+') or (entrada='-') or
            (entrada='e') or (entrada='.') do
      begin
            case estado of
               1: case entrada of
                  '0','1',…'9': estado=2;
                  '-','+': estado=2;
                  '.': estado=3;
                  otherwise: estado=8;
                  end;
               2: …
               …
            end;
            read(entrada);
      end;
      realnum:=(estado=4) or (estado=7)
end
```

19

Autómatos finitos (exemplos)

Autómato finito que reconhece os *tokens*: **if id (< <=**



Para reconhecer uma entrada como:

if (a <= …

- seguem-se as transições de estado começando no estado inicial até não se conseguir ir mais além (máxima *substring*);
- nessa altura se estivermos num estado final reconhece-se o *token* respectivo;
- se não estivermos num estado final recuamos, carácter a carácter, até passarmos por um estado final.

20

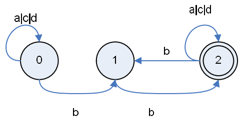
Tipos de autómatos

Seja A=(S, Σ, i, F, δ) um autómato finito.

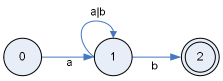
- A diz-se **autómato finito determinístico** (AFD) se, perante um símbolo x de Σ, puder transitar, no máximo, para um único estado, isto é:

  $((s, x, s') \in \delta \wedge (s, x, s'') \in \delta) \Rightarrow s' = s''$
- Caso contrário, A diz-se **não determinístico**
- A diz-se autómato finito ε se é possível transitar de estado sem usar nenhum símbolo de Σ, isto é:

  $\delta \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times S$



Para o alfabeto Σ = {a, b, c, d}, qualquer palavra com um número par de símbolos "b".



uma string em a,b em que todas as strings acabam em "b" e começam em "a";

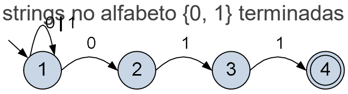
21

Autómatos finitos não-determinísticos

Os autómatos anteriores são determinísticos (as transições entre estados fazem-se apenas com caracteres do alfabeto, e todas as transições que saem de um estado estão etiquetadas com caracteres diferentes)

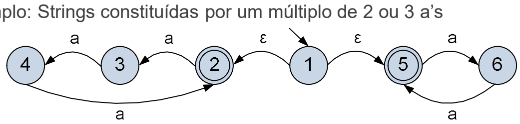
Certas *strings* são mais fáceis de especificar com um autómato não determinístico

- Exemplo: Todas as strings no alfabeto {0, 1} terminadas por 0 1 1



Os autómatos finitos não determinísticos podem ter transições de estado sem consumirem qualquer entrada - as chamadas transições ε

- Exemplo: Strings constituídas por um múltiplo de 2 ou 3 a's



22

Aceitação de AFN's

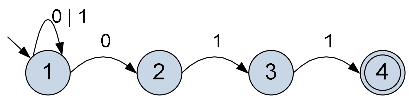
- Uma *string* $x_1 \ldots x_n$ é aceite por um AFN se existir um caminho começando no estado inicial e terminando num estado final, cujas etiquetas dos ramos percorridos coincidam, por ordem, com os caracteres x_1 a x_n, com a possível inclusão, em qualquer local, de qualquer número de transições ε
- Os autómatos finitos (determinísticos e não determinísticos) definem linguagens, que são exactamente o conjunto de *strings* aceites pelo autómato
- Existe "equivalência" entre expressões regulares (ER), autómatos finitos determinísticos (AFD) e autómatos finitos não determinísticos (AFN)
  - Dada uma expressão regular é sempre possível construir um AFN que aceita exactamente a mesma linguagem
  - Dado um AFN é sempre possível construir um AFD que aceita exactamente a mesma linguagem – algoritmo da "construção dos subconjuntos"
  - Dado um AFD é sempre possível construir uma ER que aceita exactamente a mesma linguagem

23

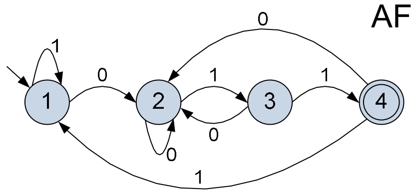
Equivalência entre AFN e AFD (exemplo)

Exemplo: Todas as *strings* no alfabeto {0, 1} terminadas por 0 1 1

AFN



AFD

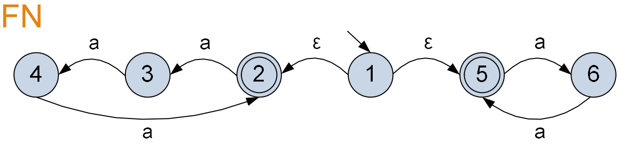


24

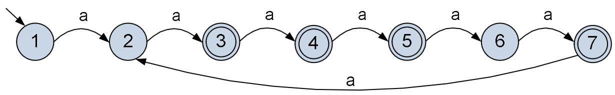
Equivalência entre AFN e AFD

Exemplo: *Strings* constituídas por um múltiplo de 2 ou 3 a's

AFN



AFD



25

Caminho e rótulo

Seja A=(S, Σ, i, F, δ) um autómato finito:

- Um caminho não trivial é uma sequência $(s_0, a_1, s_1), (s_1, a_2, s_2), \ldots, (s_{n-1}, a_n, s_n)$ onde $(s_{i-1}, a_i, s_i) \in \delta$
- Um caminho trivial é uma tripla da forma (s, ε, s), com $s \in S$
- O rótulo do caminho é $a_1\ a_2 \ldots a_n$

26

Linguagem reconhecida por um AF

Seja A=(S, Σ, i, F, δ) um autómato finito.

Um caminho diz-se bem sucedido se começa num estado inicial e termina num estado final

Linguagem reconhecida por A:

- L(A) = {u $\in \Sigma^*$: u é o rótulo de um caminho bem sucedido em A}

27