Compiladores 2025/26

Análise Lexical

Raul Barbosa

Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra

Análise lexical

```
if (a==b)
  c=1;
else
  c=0;
```

```
\tif␣(a==b)\n\t\tc=1;\n\telse\n\t\tc=0;
```

Objetivo: dividir o programa nos seus símbolos léxicos (*tokens*)

- **Léxico** ≡ vocabulário ≡ conjunto de palavras de uma linguagem
- Atribui-se também uma categoria a cada símbolo léxico
- Categorias lexicais: palavras-chave, identificadores, operadores, números, espaços em branco, classes específicas para ( ) = ;

Símbolos léxicos (ou *tokens*)

- **Identificadores** – Strings compostas por letras ou dígitos, começadas por uma letra. Exemplos: `a1`, `cnt`, `n`
- **Naturais** – Strings não vazias compostas apenas por dígitos. Exemplos: `0`, `52`, `007`, `00`
- **Palavras-chave** – "`if`" ou "`else`" ou "`while`" ou "`break`" ou "`return`" ou... (em regra são palavras reservadas)
- **Espaços em branco** – Sequências não vazias de espaços, quebras de linha ou tabulações. Exemplo: `if␣␣␣(x␣<␣0)`
- ...

Linguagens regulares

- A estrutura lexical de uma linguagem de programação é especificada usando linguagens regulares.
- A estrutura lexical consiste nas categorias de símbolos léxicos.
- Temos de caraterizar (por compreensão) o conjunto de *strings* que pertence a cada uma das categorias.
- Para tal, habitualmente usa-se linguagens regulares.

Expressões regulares

Dado um alfabeto Σ, são expressões regulares as constantes:

- Caráter literal a tal que $a \in \Sigma$ denota o conjunto {"a"}
- Cadeia vazia ϵ denota o conjunto {""}
- Conjunto vazio ϕ denota o conjunto $\{\}$ sendo que $\phi \neq \epsilon$

Sendo R e S expressões regulares são também expressões regulares:

- União $R|S = \{r : r \in R\} \cup \{s : s \in S\}$
- Concatenação $RS = \{rs : r \in R \wedge s \in S\}$
- Repetição $R^* = \bigcup_{i \geqslant 0} R^i$ com $R^0 = \{\epsilon\}$, $R^i = \underbrace{RR...R}_{i \text{ vezes}}$

Cada expressão R denota uma linguagem $L(R)$, que é o conjunto de todas as *strings* caraterizado por R.

Expressões regulares

Notação	Significado
a	A string "a"
ϵ	A string vazia
R|S	Escolher R ou S, selecionar um de R $\cup$ S
RS	R seguido de S, concatenação
R*	Repetir R zero ou mais vezes
R$^+$	Repetir R pelo menos uma vez, RR*
R$^?$	R é opcional (zero ou uma vez), R|ϵ
[a-zA-Z]	Seleção de uma gama de carateres, a|b|...|Y|Z
[^a-z]	Qualquer caráter que não esteja na gama
.	Qualquer caráter exceto quebra de linha
"a.b*"	A string entre aspas

Categorias lexicais

Categoria	Expressão regular
letter	[A-Za-z]
digit	[0-9]
ID	letter(letter|digit)*
NATURAL	digit$^+$
IF	if
ELSE	else
LPAR	(
RPAR	)
SEMICOLON	;
...	...

Especificação lexical

1. Escrever uma regexp R_i para cada categoria lexical (números, palavras-chave, identificadores, parêntesis, ...)
2. Construir uma regexp R que é a união de todas as regexps ($R = R_1|R_2|R_3|...$)
3. Tomando a entrada $x_1x_2x_3x_4x_5x_6...x_n$, para cada prefixo determinar se $x_1x_2...x_i \in L(R)$ (se pertence à linguagem de R)
4. Se a resposta for *sim*, então $x_1x_2...x_i \in L(R_j)$ para algum j
5. Apagar $x_1x_2...x_i$ da entrada (remover prefixos sucessivamente)
6. Enquanto a entrada não terminar, repetir a partir de 3.

Regras de operação

1. Quantos carateres de input usar de cada vez?

$x_1 ... x_i \in L(R)$
$x_1 ... x_j \in L(R)$ Ambos são categorias lexicais válidas.
$i \neq j$

Regra: Escolher sempre a *substring* mais longa (*longest match*).

2. Qual categoria lexical escolher?

$x_1 ... x_i \in L(R_j)$
$x_1 ... x_i \in L(R_k)$ Pertence a várias categorias (*e.g.*, "if").
$x_1 ... x_i \in L(R_l)$

Regra: Prioritizar as categorias, ordenando as expressões regulares.

3. E se não se encontrar nenhuma categoria lexical?

Evita-se situações em que $x_1 ... x_i \notin L(R)$, criando uma categoria *errors* (todas as strings que não pertençam à especificação lexical).

Autómatos de estados finitos

Regexps e autómatos finitos especificam as mesmas linguagens.

- Os autómatos finitos (AFNDs e AFDs) reconhecem *strings*, dizendo *sim* ou *não* relativamente a cada cadeia de carateres.

Um autómato finito tem

- um conjunto finito de estados S
- um alfabeto ou conjunto de símbolos Σ
- um estado inicial $s_0 \in S$
- um conjunto F de estados finais (estados de aceitação)
- um conjunto de transições $s_i \xrightarrow{input} s_j$

"Mecânica" dos autómatos

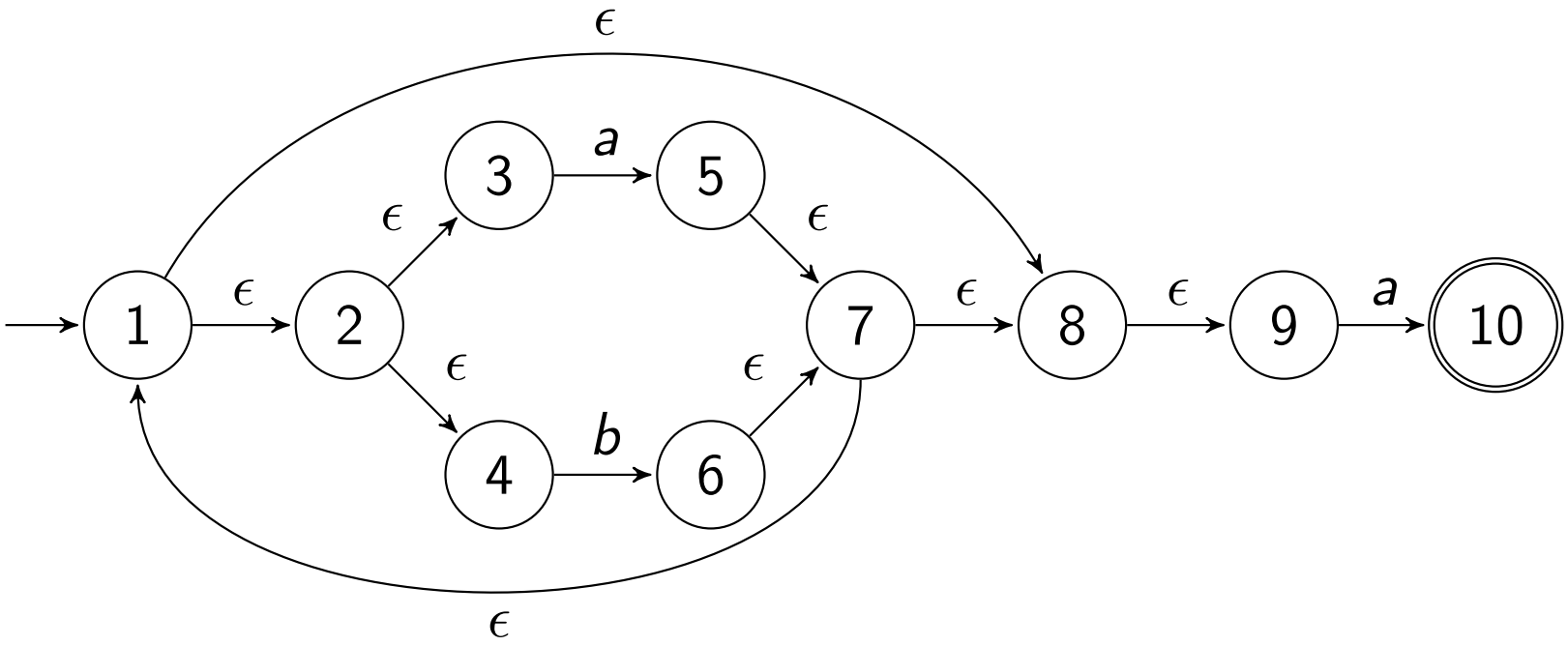
- $s_1 \xrightarrow{a} s_2$ se estivermos no estado s_1 e lermos um caráter 'a' o autómato pode transitar para o estado s_2.
- Se chegar ao final do *input* **e** estiver num estado final ($s_n \in F$) então **aceita** a *string* (pertence à linguagem).
- Caso contrário, **rejeita** a *string* uma vez que
  - ou termina num estado $s \notin F$,
  - ou o autómato "encrava" com a leitura.

Construção de um analisador lexical

Um analisador lexical é um programa que lê uma *string* e a particiona em símbolos léxicos.

1. Escreve-se uma regexp para cada categoria lexical.
2. Constrói-se um AFND para cada regexp, que aceite a mesma linguagem dessa regexp.
3. Obtém-se um só AFND pela união dos AFNDs anteriores.
4. Converte-se o AFND (não-determinístico) para AFD (determinístico).
5. Obtém-se a tabela de transições do AFD.
6. Escreve-se um programa que use a tabela para simular o AFD, que é o pretendido analisador lexical.

REGEXP $(a|b)^*a$ convertida para AFND



Construção por indução a partir das sub-expressões
com o algoritmo de Thompson.

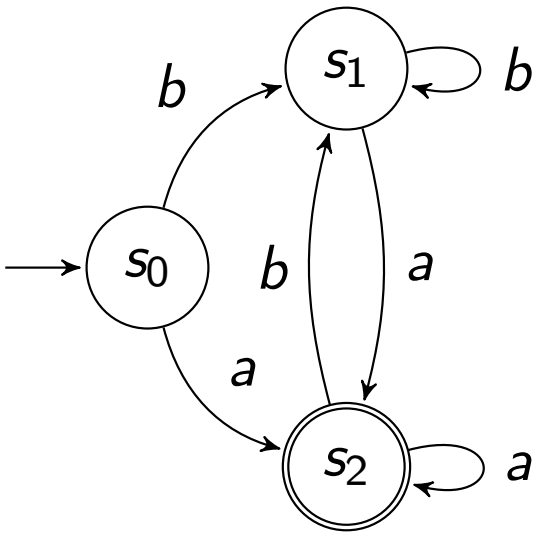
Conversão de AFND em AFD

- fecho-ϵ de um estado P é o conjunto de estados que podem ser alcançados a partir de P através de transições ϵ (zero ou mais)
- fecho-ϵ de um conjunto T de estados é definido de forma semelhante: $\bigcup_{s \in T}$ fecho-ϵ (s)
- a(X) $= \{y : x \in X \wedge x \xrightarrow{a} y\}$ é o conjunto de estados alcançáveis a partir de $x \in X$ com o símbolo 'a'

AFD resultante

- Estados: subconjuntos não-vazios de S
- Estado inicial: fecho-ϵ(s)
- Estados finais: $\{X : X \cap F \neq \phi\}$
- Transições: $x \xrightarrow{a} y$ se e só se $y =$ fecho-$\epsilon(a(x))$

AFND convertido para AFD



$$s_0 = \{1, 2, 3, 4, 8, 9\}$$
$$s_1 = \{6, 7, 1, 2, 3, 4, 8, 9\}$$
$$s_2 = \{5, 7, 10, 1, 2, 3, 4, 8, 9\}$$

Tabela **T** de dupla entrada

Um AFD pode ser simulado com uma tabela T de dupla entrada.

- Uma dimensão é o estado do autómato
- Outra dimensão é o símbolo de entrada lido
- Para cada transição $s_i \xrightarrow{a} s_j$, T[i,a]=j

	a	b
0	2	1
1	2	1
2	2	1

Tabela T: "a partir de um estado, lendo um símbolo, o próximo estado é..."

Simulação de autómato determinístico em C

```
char input[] = "abababa";

int T[3][2] = { {2, 1},
                {2, 1},
                {2, 1} };

int F[3] = {0, 0, 1};

int main(void) {
  int index = 0;
  int state = 0;

  while(input[index])
    state = T[state][input[index++]-'a'];

  if(F[state])
    printf("accept\n");
  else
    printf("reject\n");
}
```

Geradores de analisadores lexicais

Automatizam a conversão REGEXP $\rightarrow$ AFND $\rightarrow$ AFD $\rightarrow$ Analisador

- Cada categoria lexical é especificada por uma expressão regular
- A prioridade é determinada pela ordem de especificação das expressões regulares
- É gerado um analisador lexical que lê texto e produz uma lista de *tokens*

Exemplos de geradores:

- Lex, flex (linguagem C)
- Quex (linguagens como C e C++)
- JLex, JFlex (linguagem Java)