UNIVERSIDADE FEDERAL DA BAHIA
INSTITUTO DE COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

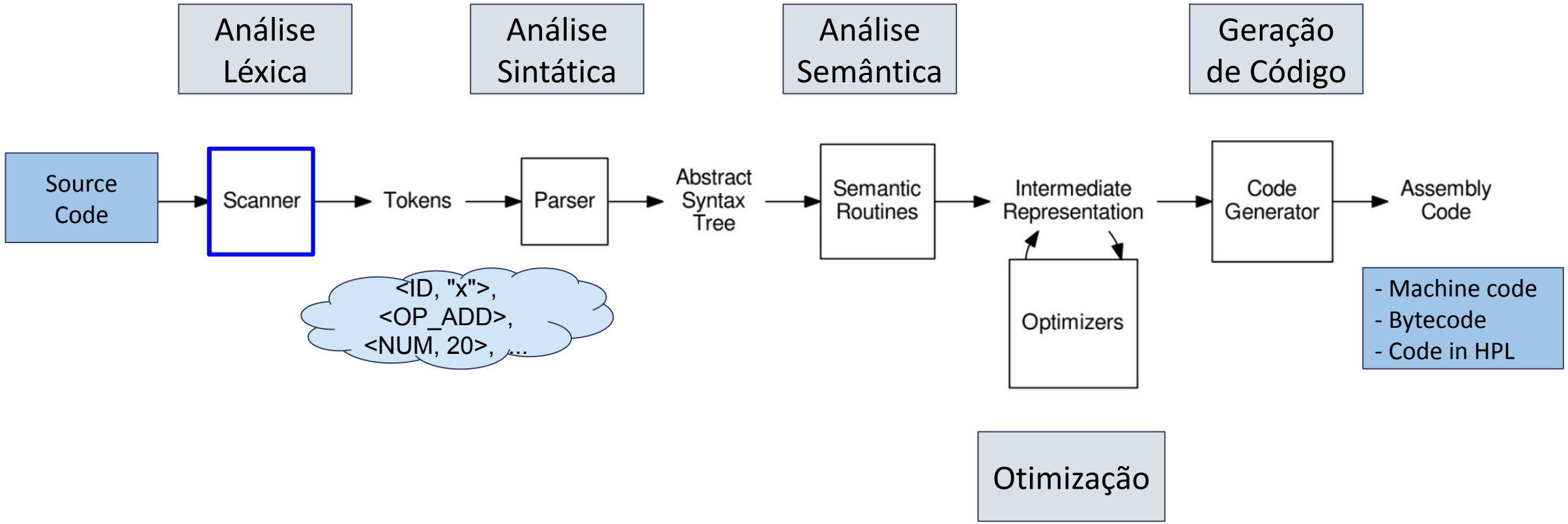
AULA 02 - ANÁLISE LÉXICA

MATA61 – COMPILADORES

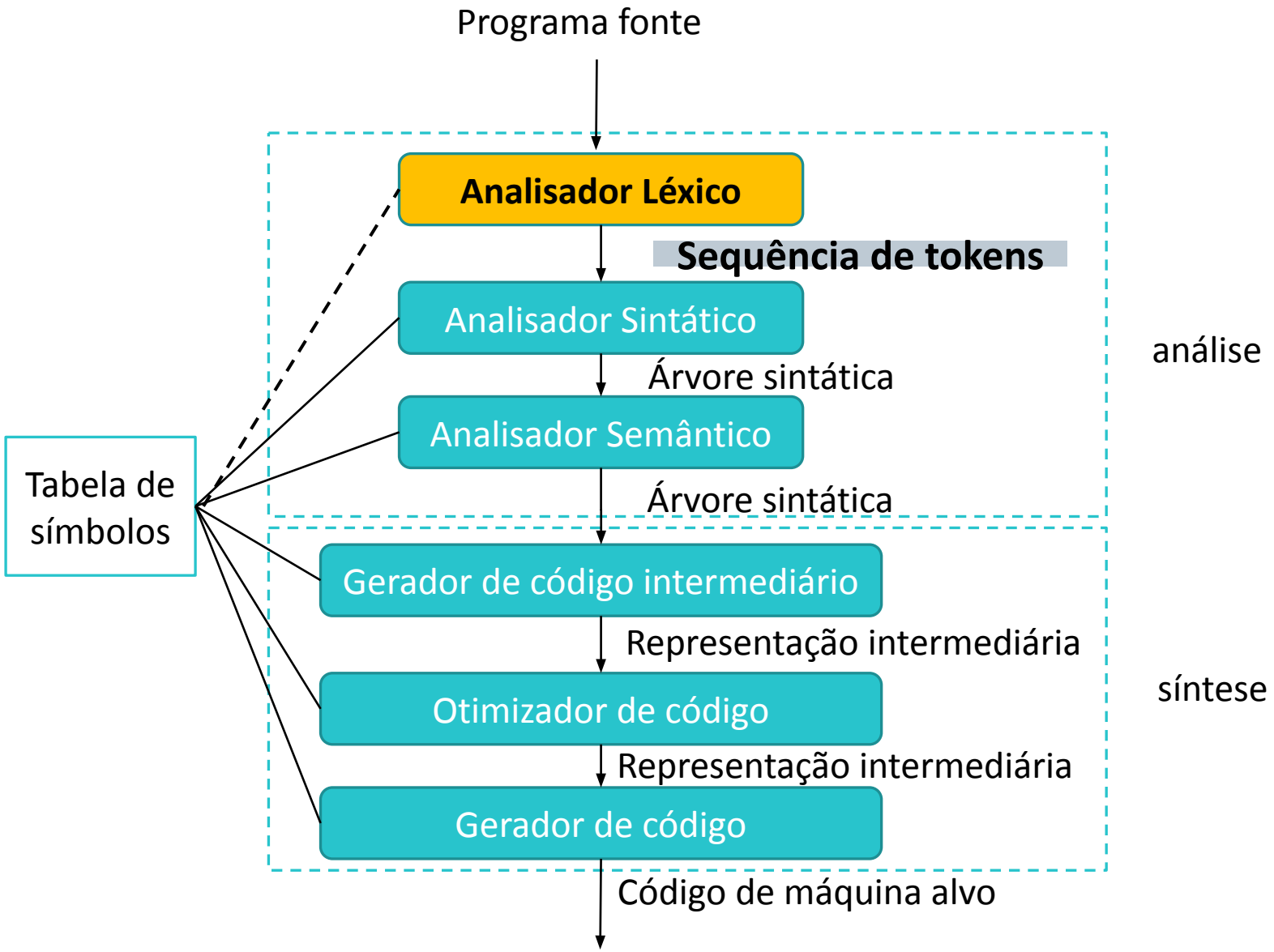
AGENDA

1. Funções
2. Padrões regulares
3. Especificação de tokens
4. Reconhecimento de tokens
5. Implementação de análise léxica

Processo de Compilação



ANÁLISE LÉXICA



Analisador Léxico (*scanner*)

Funções Básicas

- Ler o programa-fonte (texto)
- Identificar unidades básicas (*tokens*) com base em **padrões**

  Categoria **num** é especificada pelo padrão **[0-9]+** que permite identificar sequências de caracteres numéricos: **0, 61, 957, 4190**, etc.
- Identificar e reportar erro léxico

Outras Funções

- Pular comentários e "whitespace"
- Contar número de linhas lidas
- Inserir elementos na tabela de símbolos

Conceitos

token	*lexemes*	descrição informal de padrão	
const	const	palavra reservada const	const
if	if	palavra reservada if	if
id	x, total, i, cont	letra seguida por letras ı dígitos	[a-z][a-z0-9]*
num	923, 5.8E11, 10	qualquer constante numérica	[0-9]+, etc.
cadeia	"uma cadeia"	qualquer caracter entre " e ", exceto "	\".*\"
op_adic	+	símbolo de adição	+

- ***Token***: unidade básica do programa fonte (categoria e atributo)
- ***Lexeme***: sequência de caracteres da entrada associada a um *token*
- **Padrão:** regra de formação para identificação de lexemes associados a categorias de tokens.

Conceitos

Tokens e seus atributos

```
x := y + z * 1.5;
```

<id, **"x"**>
<op_atrib>
<id, **"y"**>
<op_adic>
<id, **"z"**>
<op_mult>
<num, **"1.5"** ou seu **valor numérico**>

```
if (foo) then x++;
```

<if> // keyword
<lpar>
<id, **"foo"**>
<rpar>
<then> // keyword
<id, **"x"**>
<op_incr_pos>

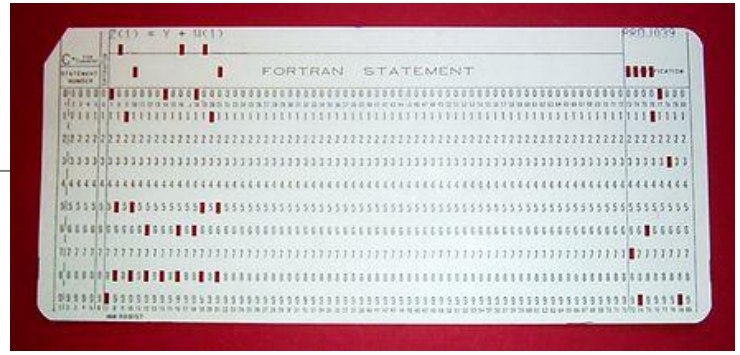
Conceitos

Categorias gerais de tokens

- **Key, Keyword:** nome com papel pré-definido na linguagem, como `class` ou `true`
- **Id, Identifier:** nome escolhido pelo programador para variáveis, funções, classes e outros elementos de código
- **Num, Number:** sequência de dígitos e/ou caracteres especiais, formatada de acordo com os tipos numéricos da linguagem (inteiro, ponto flutuante, etc.)
- **Sym, Symbol:** caracter(es) não alfanumérico(s) com significado especial (+, -, !=, etc.)
- **String:** sequência de caracteres, normalmente entre " "

Comentários e *whitespace* são, em geral, desconsiderados no processo de tradução.

Dificuldades



Formato fixo da entrada imposto por algumas linguagens

- Fortran e as primeiras colunas reservadas

Tratamento dado a *whitespace* em algumas linguagens

```
DO 5  I = 1.25    (ou DO5I = 1.25)
DO 5  I = 1,25
```

Inexistência de palavras-chave reservadas

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
```

Prefixos

`=` e **`==;`** **`+,`** **`++`** e **`+=;`** **`!`** e **`!=`**

O analisador léxico precisa guardar informação e olhar adiante (“lookahead”) para decidir a categoria de token(s)

Tratamento de Erros

Erros que o analisador léxico pode identificar e reportar

- **Símbolo não pertencente ao alfabeto da linguagem:** `$`
- **Identificador mal formado:** `j@c, a`
- **Tamanho do identificador:** `minha_variavel_para_...`
- **Número mal formado:** `2.a3`
- **Fim de arquivo inesperado / comentário não fechado:** `{...`
- **Caractere ou string mal formado:** `'a, "hello world`

Tratamento de Erros

Recuperação de erros léxicos

- Remoção de sucessivos caracteres da entrada, até que o Analisador encontre um caractere que sinalize o início de um novo token (Panic Mode)
- Remoção de caractere incorreto
- Inserção de caractere ausente

Analisador Léxico

Implementação

Desenvolvimento tradicional

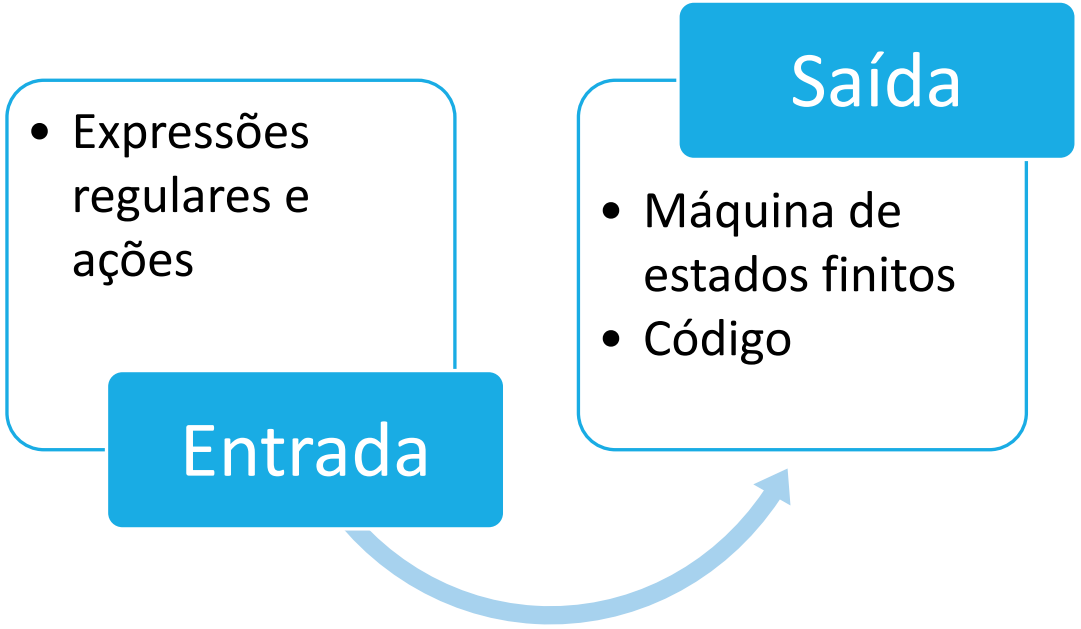
Programação do analisador léxico em C, Assembly xxx, …

Geração automática

Ferramentas como lex e flex geram código do analisador léxico em diferentes linguagens de programação

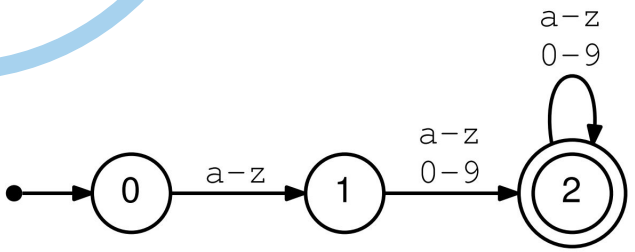
Analisador Léxico

Geração automática

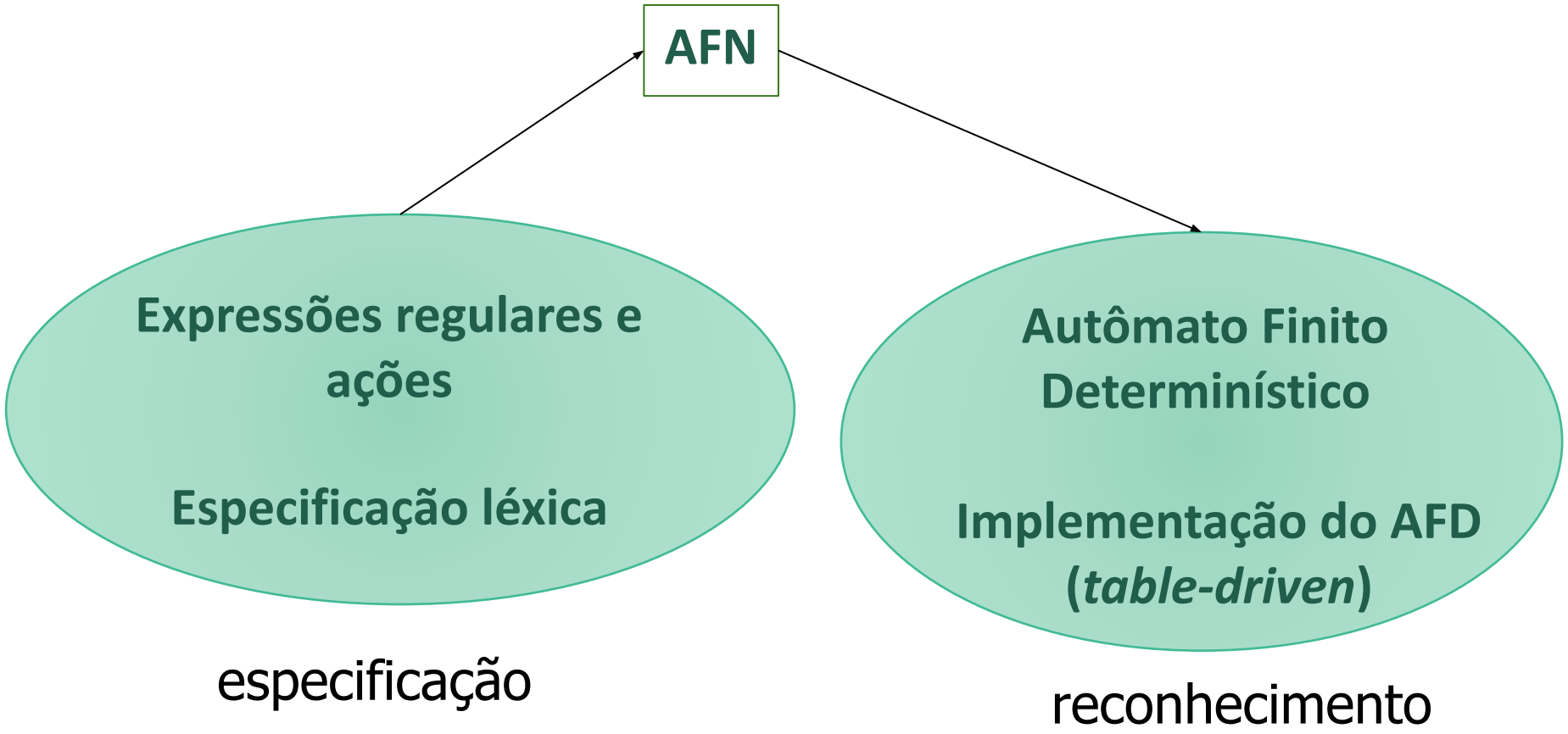


Expressões regulares **especificam**
e
autômatos finitos **reconhecem**
linguagens regulares

$[a\text{-}z]\ [a\text{-}z0\text{-}9]^{+}$



Geração Automática de Analisador Léxico



Especificação de Tokens

Expressões regulares (ERs) são uma linguagem para expressar padrões

Uma expressão regular s é uma string que denota $L(s)$, um conjunto de strings tiradas de um alfabeto Σ. $L(s)$ é conhecido como a "linguagem de s"

$L(s)$ é definido indutivamente com os seguintes casos base:

- Se $a \in \Sigma$ então a é uma expressão regular e $L(a) = \{a\}$
- ε é uma expressão regular e $L(\varepsilon)$ contém apenas a string vazia.

Então, para quaisquer expressões regulares s e t:

1. $s|t$ é um RE tal que $L(s \mid t) = L(s) \cup L(t)$
2. st é um RE tal que $L(st)$ contém todas as strings formadas pelo concatenação de uma string em $L(s)$ seguida por uma string em $L(t)$
3. s^* é um RE tal que $L(s^*) = L(s)$ concatenado zero ou mais vezes.

Especificação de Tokens

Expressão regular s	**Linguagem L(s)**
Hello	{ hello }
d(o|i)g	{ dog,dig }
moo*	{ mo,moo,mooo,... }
(moo)*	{vazio,moo,moomoo,... }
a(b|a)*a	{ aa,aaa,aba,aaaa,aaba,abaa,... }

s? indica que s é opcional.
s? pode ser escrito como (s | ε)
s+ indica que s é repetido uma ou mais vezes.
s+ pode ser escrito como ss^*
[a-z] indica qualquer caractere nesse intervalo.
[a-z] pode ser escrito como (a | b | ... | z)
[^X] indica qualquer caractere, exceto um.
[^X] pode ser escrito como Σ - x

Propriedades algébricas:
Associativa: a | (b | c) = (a | b) | c
Comutativa: a | b = b | a
Distributiva: a (b | c) = ab | ac
Idempotência: a ** = a *

Especificação de Tokens

- Exemplo 1: identificadores da linguagem de programação Pascal

letra A | B | ... | Z | a | ... | z
digito 0 | 1 | ... | 9
id letra (letra | digito)*

ou

letra [A-Za-z]
digito [0-9]
id letra (letra | digito)*

Especificação de Tokens

- Exemplo 2: constantes numéricas em Pascal

digito	0 | 1 | ... | 9
digitos	digito digito*
fração_opc	. digitos | ε
exp_opc	(E (+| - | ε) digitos) | ε
num	digitos fração_opc exp_opc

(notação científica) **45.22E-5**

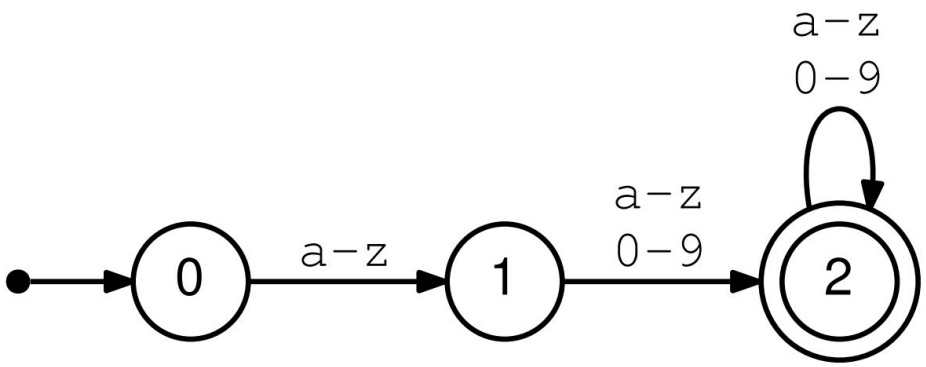
ou

digito	[0-9]
digitos	digito+
fração_opc	(. digitos)?
exp_opc	(E (+| -)? digitos) ?
num	digitos fração_opc exp_opc

Reconhecimento de Tokens

Autômatos Finitos (AF)

- AF que reconhece cadeias **s**, |**s**| ≥ 2, com caracteres alfanuméricos, sendo que o primeiro deve ser alfabético.
- Expressão regular: [a-z] [a-z0-9] +



Autômatos Finitos - Definição Formal

Um autômato finito é uma 5-tupla $(\Sigma, Q, \delta, q, F)$ em que:

- Σ é um alfabeto de entrada
- Q é o conjunto de estados
- q é o estado inicial do autômato
- $F \subseteq Q$ é o conjunto de estados finais
- δ é a função de transição

  $Q \times \Sigma \rightarrow Q$

Reconhecimento de Tokens

Autômato Finito Determinístico (AFD)

Autômato Finito Não-Determinístico (AFN)

Procedimento comum:

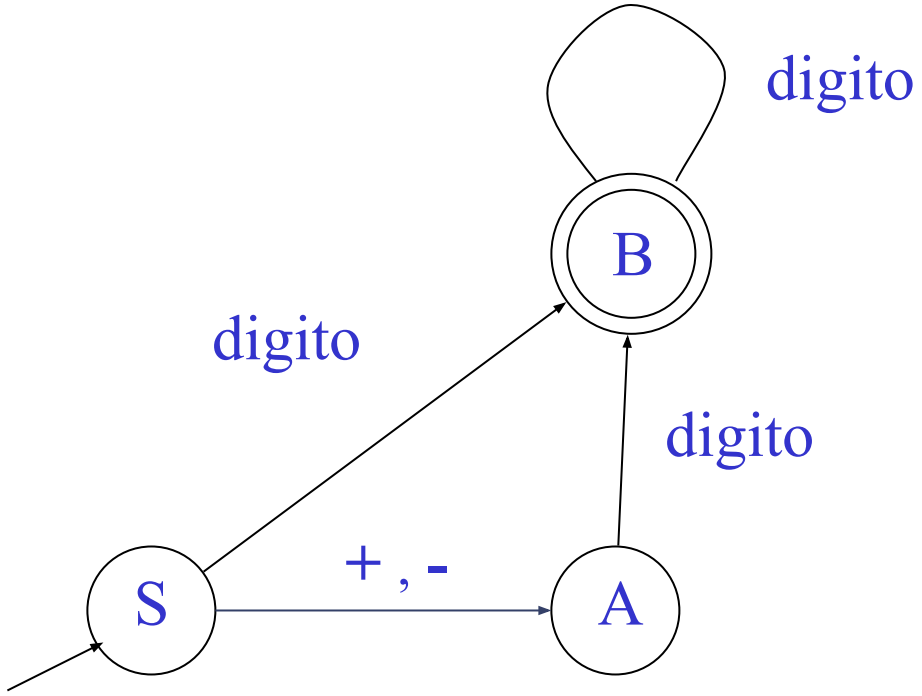
- obter um AFN a partir de uma ER, transformar um AFN em AFD, minimizar o AFD.

Exemplo - Constantes Inteiras com Sinal

- Definição regular

digito [0-9]
sinal_opc (+|-)?
constante sinal_opc digito+

- Autômato finito determinístico



Autômatos Finitos

- Implementação de AFD *table-driven*
  - uma linha para cada estado do autômato, e
  - uma coluna para cada símbolo do alfabeto de entrada
- Table[j][k]
  - define para qual estado deve-se ir a partir da leitura do símbolo k no estado j,
  - uma entrada vazia corresponde a erro.

	+ -	[0-9]
S	A	B
A		B
B		B

Análise Léxica

Programa *Table-Driven* para um AFD

```
state = S                                   // S is the start state
repeat {
    k = next character from the input
    if k == EOF then                        // end of input
        if state is a final state then
            accept
        else
            reject
    state = T[state,k]                      // transition
    if state == empty then
        reject                              // got stuck
}
```

FLEX

- Gerador de "scanners" usados para análise léxica
  - ER para AFN para AFD
- A especificação em Flex contém expressões regulares, fragmentos de código C e algumas diretivas especializadas
- O programa Flex produz código C compilável

FLEX

Estrutura geral de um programa em Flex

```
%{
    (C Preamble Code)
%}
    (Character Classes)
%%
    (Regular Expression Rules)
%%
    (Additional Code)
```

Código C arbitrário que será copiado para o início do código gerado.

Definições regulares, que são uma abreviação simbólica para expressões regulares comumente usadas.

Declaração das expressões regulares e ações correspondentes (código C) que serão executadas.

Código C arbitrário, normalmente funções auxiliares adicionais, que será copiado para o final do código gerado.

FLEX

Especificação em Flex

```
%{
#include "token.h"
%}
DIGIT  [0-9]
LETTER [a-zA-Z]
%%
(" "|\t|\n)   /* skip whitespace */
\+            { return TOKEN_ADD; }
while         { return TOKEN_WHILE; }
{LETTER}+     { return TOKEN_IDENT; }
{DIGIT}+      { return TOKEN_NUMBER; }
.             { return TOKEN_ERROR; }
%%
int yywrap() { return 1; }
```

Contém a definição de categorias de tokens

Definições regulares

Expressões regulares para "whitespace" (com nenhuma ação associada), símbolo de adição, a palavra-chave while, identificador e número (com ações para retorno de *tokens*).

O ponto (.) captura qualquer caracter e a ação definida é retornar TOKEN_ERROR, para sinalizar erro léxico.

FLEX

Categorias de tokens

Um arquivo, por exemplo, **"token.h"**, deve definir as categorias de token da linguagem a serem usados no programa Flex. O arquivo que contém o programa Flex deve incluir "token.h".

"token.h"

```
typedef enum {
      TOKEN_EOF=0,
      TOKEN_WHILE,
      TOKEN_ADD,
      TOKEN_IDENT,
      TOKEN_NUMBER,
      TOKEN_ERROR
} token_t;
```

Especificação de Tokens

Definições Regulares

- Definições regulares são usadas para vincular **nomes** a expressões regulares, promovendo o reuso.

Uma definição regular é uma sequência de definições na forma:

```
d1  r1
d2  r2
...
dn  rn
```

onde cada `di` é um nome distinto e cada `ri` é uma expressão regular que pode conter nomes definidos `{d1,d2,...,dn}`.

FLEX

Arquivo com definição da função main()

```
#include "token.h"
#include <stdio.h>

extern FILE *yyin;
extern int yylex();
extern char *yytext;

int main() {
    yyin = fopen("program.c","r");
    if(!yyin) {
        printf("could not open program.c!\n");
        return 1;
    }

    while(1) {
        token_t t = yylex();
        if(t==TOKEN_EOF) break;
        printf("token: %d  text: %s\n",t,yytext);
    }
}
```

Declaração de elementos externos definidos no código C gerado: **yyin** designa o arquivo de entrada, **yylex** é a função que implementa o analisador léxico e **yytext** aponta para o lexema do token atual.

REFERÊNCIAS

- Capítulo 3 - Livro do Dragão
- Capítulo 3 – Livro do Douglas Thain