Compiladores (CC3001) — Aula 2: Análise Lexical

Mário Florido

DCC/FCUP

2024

Esta aula

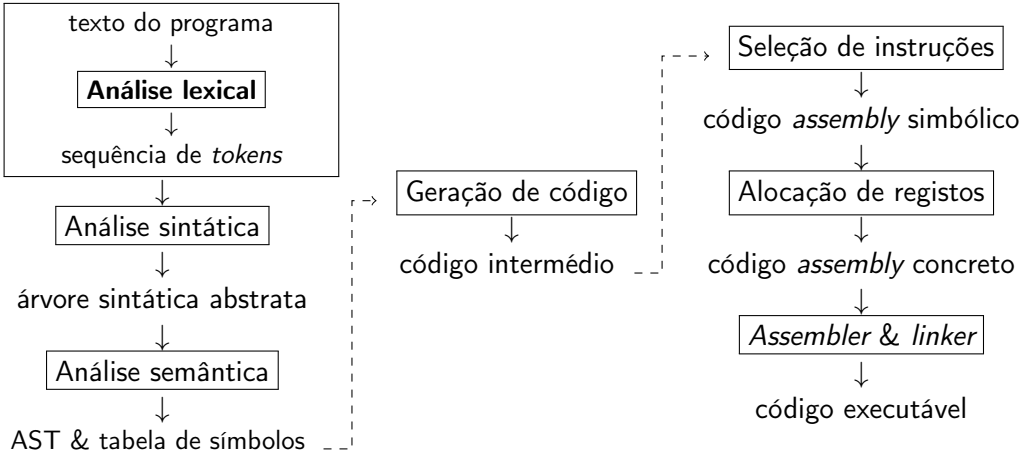
Análise lexical

Expressões regulares

Autómatos finitos determinísticos

Autómatos finitos não-determinísticos

Re-lembrar: fases dum compilador



Esta aula

Análise lexical

Expressões regulares

Autómatos finitos determinísticos

Autómatos finitos não-determinísticos

léxico: conjunto de palavras de uma determinada língua (...); sinónimo de vocabulário

Dicionário de Português Online

- Léxico de uma linguagem de programação: conjunto dos símbolos e palavras usados para compor programas (*tokens*)

  identificadores nomes de variáveis, funções, etc.

  literais números, caracteres, cadeias

  palavras reservadas `if`, `else`, `while`, etc.

  operadores +, *, /, =, ...

- Um analisador lexical (também chamado *lexer*, *scanner* ou *tokenizer*) decompõe o texto de entrada em *tokens*
- Esta decomposição facilita a análise sintática subsequente

Alguns exemplos de *tokens*

Para a linguagem C:

ID	`foo  main  __last_14`
NUM	`73  0  -13  0xff`
REAL	`66.1  .5  10.0  5.5e-10`
IF	`if`
COMMA	`,`
NOTEQ	`!=`
LPAREN	`(`
RPAREN	`)`
⋮	⋮

- Os *tokens* são identificados por uma etiqueta (*token type*)
- Nalguns casos podem ter também um valor associado

	etiqueta	valor
`if`	IF	—
`,`	COMMA	—
`!=`	NOTEQ	—
`(`	LPAREN	—
`)`	RPAREN	—
`main`	ID	"main"
`71`	NUM	71
`.5`	REAL	0.5

Análise lexical: exemplo

```
float match0(char *s)    /* find a zero */
{ if (!strncmp(s, "0.0", 3))
     return 0.0;
}
```

FLOAT, ID(“match0”), LPAREN, CHAR, STAR, ID(“s”), RPAREN, LBRACE, IF, LPAREN, BANG, ID(“strncmp”), LPAREN, ID(“s”), COMMA, STRING(“0.0”), COMMA, NUM(3), RPAREN, RPAREN, RETURN, REAL(0.0), SEMI, RBRACE, EOF

- Além de separar os *tokens*, a análise lexical consume:
  - *carateres brancos* (espaços, mudanças de linhas e tabulações);
  - *comentários* multi-linha `/* ... */` ou até ao final da linha `//`
- Usamos um *token* EOF para marcar o fim do *input*

Vamos implementar analisadores lexicais diretamente em Haskell e C.

Alternativas para os *tokens*:

```
data Token = ID String
           | NUM Int
           | LPAREN
           | RPAREN
           | COMMA
           | IF
           ⋮
           deriving (Eq,Show)
```

O analisador é implementado como uma função:

```
lexer :: [Char] -> [Token]
```

- Transforma uma lista de carateres numa lista de *tokens*
- Definida por análise de casos sobre o próximo carater

```
typedef enum { ID, NUM, LPAREN, RPAREN, COMMA, IF,
               ⋮
             } TokenType;
```

- Uma enumeração para os tipos de *tokens*
- Os valores de *tokens* como ID e NUM são representados separadamente (porque a linguagem C não tem tipos algébricos)

O analisador é implementado como uma função:

```
TokenType getToken(TokenValue*);
```

- Retorna o tipo do próximo *token* da entrada
- O valor dos *tokens* é guardado numa união:

```
typedef union {
   int  token_num;            // valor de NUM
   char token_txt[MAX_SIZE];  // texto do identificador
} TokenValue;
```

- Faltam: comentários, números de vírgula flutuante, operadores e palavras reservadas...
- Estes programas são simples mas bastante repetitivos
- É necessário algum cuidado com a ordem das condições
  (e.g. "`if`" deve ser uma palavra reservada e não um identificador)
- São difíceis de modificar

Em vez de escrever o analisador lexical manualmente podemos:

- descrever os *tokens* usando expressões regulares;
- gerar automaticamente o analisador a partir dessa descrição.

Esta aula

Análise lexical

Expressões regulares

Autómatos finitos determinísticos

Autómatos finitos não-determinísticos

Como são *identificadores* em C?

> *An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper- and lowercase letters are different. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characteres that could possibly constitute a token. Blanks, tabs, newlines, and comments are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.*
>
> *(fonte: Modern Compiler Implementation in ML)*

Como são *identificadores* em C?

> *An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper- and lowercase letters are different. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characteres that could possibly constitute a token. Blanks, tabs, newlines, and comments are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.*
>
> *(fonte: Modern Compiler Implementation in ML)*

Os analisadores anteriores estão incompletos — consegue ver o que falta?

- Em vez de implementar analisadores diretamente, vamos descrever os *tokens* usando expressões regulares
- Vantagens:
  1. as expressões regulares permitem descrições concisas e não-ambíguas
  2. podem ser automaticamente traduzidas para automátos finitos que implementam analisador lexicais eficientes

Uma linguagem é um sub-conjunto $L \subseteq \Sigma^*$ de *palavras* formadas a partir de um *alfabeto* Σ.

As expressões regulares (REs) são:

a	uma letra $a \in \Sigma$
ε	palavra vazia
$M \mid N$	união de duas expressões regulares M e N
$M \cdot N$	concatenação[1] de duas expressões regulares M e N
M^*	repetição[2] de M, i.e., a concatenação de zero ou mais palavras de M

[1] Ou seja: as palavras de $M \cdot N$ são $u \cdot v$ tal que $u \in L(M) \wedge v \in L(N)$.

[2] Também designada *fecho de Kleene*.

Exemplos

Expressão regular	Linguagem
$a \cdot b$	"ab"
$a \mid b$	"a", "b"
$(a \cdot b) \cdot a$	"aba"
$(a \cdot b) \mid (b \cdot a)$	"ab", "ba"
$(a \mid b) \cdot a$	"aa", "ba"
$(a \mid \varepsilon) \cdot b$	"b", "ab"
$(a \mid b)^*$	"", "a", "b", "aa", "ab", "ba", "bb", ...
$(a \cdot b)^*$	"", "ab", "abab", "ababab", ...

- Podemos omitir o sinal da concatenação (não é ambíguo por causa da associatividade)

$$abc = (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

- Prioridades: repetição > concatenação > união

$$ab|c = (ab)|c \qquad abb* = ab(b*)$$

- Abreviaturas:

$$M? \stackrel{\text{def}}{=} (M \mid \varepsilon) \qquad \text{opcional}$$

$$M^+ \stackrel{\text{def}}{=} M \cdot M^* \qquad \text{repetição (uma ou mais vezes)}$$

$$[x_1 x_2 \ldots x_n] \stackrel{\text{def}}{=} (x_1|x_2|\ldots|x_n) \qquad \text{alternativas}$$

$$[a_1 - a_n] \stackrel{\text{def}}{=} (a_1|a_2|\ldots|a_n) \qquad \text{intervalos de carateres contíguos}$$

Notação para REs

`a`	um carater literal
M`|`N	alternativa entre M ou N
MN	concatenação de M com N
M`*`	repetição (zero ou mais vezes)
M`+`	repetição (uma ou mais vezes)
M`?`	opção (zero ou uma ocorrência de M)
`[a-zA-Z]`	qualquer carater destes intervalos
`[^a-zA-Z]`	complementar (qualquer carater fora dos intervalos)
`.`	qualquer carater exepto mudança de linha
`\`	escape (para carateres especiais)
`"a+"`	cadeia literal (entre aspas)
`""`	cadeia vazia

`if`	palavra reservada (IF)
`[_a-zA-Z][_a-zA-Z0-9]*`	identificadores (ID)
`[0-9]+`	número inteiro (NUM)
`([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)`	número fracionário (REAL)
`//.*`	comentário até ao final da linha

- As expressões regulares são *descrições declarativas* dos *tokens*
- Para implementar o analisador necessitamos de um *modelo operacional*
- Solução: converter as expressões regulares em autómatos finitos
- A conversão pode ser efetuada por um gerador de analisadores (próxima aula)

Esta aula

Análise lexical

Expressões regulares

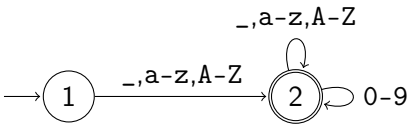
Autómatos finitos determinísticos

Autómatos finitos não-determinísticos

$(\Sigma, Q, q_0, F, \delta)$	autómato
Σ	alfabeto
Q	conjunto dos estados
$q_0 \in Q$	estado inicial
$F \subseteq Q$	estados finais
$\delta \subseteq Q \times \Sigma \times Q$	transições

- Um conjunto finito de estados e um conjunto finito de transições
- Um único estado inicial e um conjunto de estados finais
- As transições são determinísticas:
  - para cada $q \in Q$ e $\sigma \in \Sigma$ existe no máximo um q' tal que $(q, \sigma, q') \in \delta$
  - podemos também ver δ como uma função: $q' = \delta(q, \sigma)$

Exemplo: DFA para identificadores



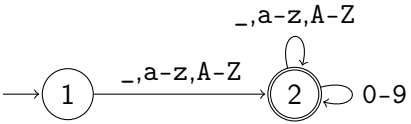
Σ é o conjunto de carateres ASCII

$Q = \{1, 2\}$

$q_0 = 1$

$F = \{2\}$

$$\begin{aligned}\delta = \{ &(1, \_, 2),\ (1, \mathtt{a}, 2),\ (1, \mathtt{b}, 2), \ldots, (1, \mathtt{z}, 2), (1, \mathtt{A}, 2),\ (1, \mathtt{B}, 2), \ldots, (1, \mathtt{Z}, 2),\\ &(2, \_, 2),\ (2, \mathtt{a}, 2),\ (2, \mathtt{b}, 2), \ldots, (2, \mathtt{z}, 2), (2, \mathtt{A}, 2),\ (2, \mathtt{B}, 2), \ldots, (2, \mathtt{Z}, 2),\\ &(2, 0, 2),\ (2, 1, 2), \ldots, (2, 9, 2)\ \}\end{aligned}$$

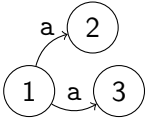


- Este DFA é equivalente à expressão `[_a-zA-Z][_a-zA-Z0-9]*`
- No caso geral: para converter uma RE num DFA temos de obter primeiro um autómato não-determinístico (NFA)
- Não podemos implementar o NFA diretamente (por causa do não-determinísmo)
- Contudo podemos sempre converter o NFA num DFA equivalente

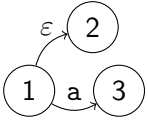
Esta aula

$(\Sigma, Q, q_0, F, \delta)$	autómato finito não-determinístico
$\vdots$	(definições de alfabeto, estados, etc. inalteradas)
$\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$	transições

- Várias transições com o mesmo símbolo a partir de um estado, e.g.



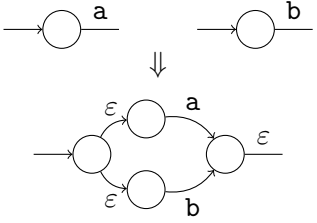
- Transições-ε (não consomem um símbolo), e.g.



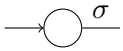
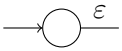
Para cada RE definimos um fragmento de autómato:

- Uma entrada unica (seta) e saida única (linha)

- A composição de fragmentos segue a estrutura da expressão, e.g.:

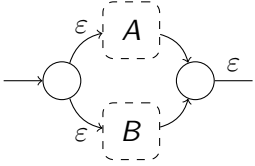


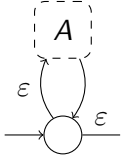
(fragmento para a|b)

σ  $(\sigma \in \Sigma)$ ε 

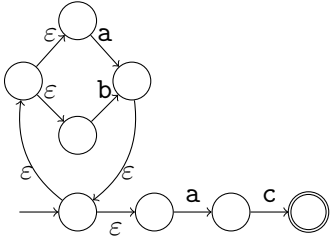
Se A, B são duas sub-expressões:

$A \cdot B$ 

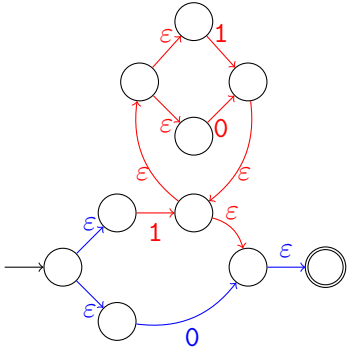
$A \mid B$ 

A^* 

(a|b)*ac



`0 | 1(0|1)*`



Dificuldades:

- as transições-ε podem ocorrem sem consumir o próximo símbolo;
- num estado q e para o próximo símbolo σ pode have transições para dois ou mais estados distintos.

Dificuldades:

- as transições-ε podem ocorrem sem consumir o próximo símbolo;
- num estado q e para o próximo símbolo σ pode have transições para dois ou mais estados distintos.

Solução: temos de converter o NFA num DFA equivalente ("*construção dos sub-conjuntos*").

- os estado do DFA são *sub-conjuntos de estados* do NFA;
- as transições do DFA "simulam" todas as transições possíveis pelo NFA.

Seja $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ um NFA.

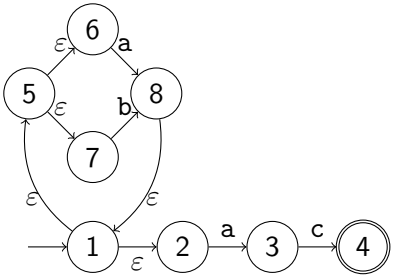
Definimos:

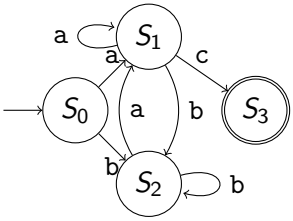
$$\text{closure}(S) = \{\ q' \in Q : \text{existe } q \in S \text{ e um caminho de transições-}\varepsilon \text{ entre } q \text{ e } q'\ \}$$

Transformamos $\mathcal{A}$ no DFA $\mathcal{A}' = (\Sigma,\ 2^Q,\ S_0,\ F', \delta')$ tal que:

$$\begin{aligned} S_0 &= \text{closure}(\{q_0\}) \\ F' &= \{\ S \subseteq Q : S \cap F \neq \emptyset\ \} \\ \delta'(S, \sigma) &= \text{closure}(\{q' : q \in S \wedge (q, \sigma, q') \in \delta\}) \end{aligned}$$

(Nota: podemos definir δ' como uma função porque $\mathcal{A}'$ é determinístico.)





$\Sigma = \{a, b, c\}$ símbolos

$Q = \{1, 2, 3, 4\}$ estados

$q_0 = 1$ estado inicial

$F = \{4\}$ estados finais

$S_0 = \{1, 2, 5, 6, 7\}$

$S_1 = \{1, 2, 3, 5, 6, 7, 8\}$

$S_2 = \{1, 2, 5, 6, 7, 8\}$

$S_3 = \{4\}$

- O DFA obtido pela construção dos sub-conjuntos pode ainda ter estados redudantes
- Podemos "fundir" estados equivalentes e obter assim o autómato mínimo
- Os geradores de analisadores lexicais efetuam esta minimização automaticamente