Análise léxica

O reconhecimento de *tokens*

Prof. Edson Alves

Faculdade UnB Gama

Fragmento de gramática que será utilizada nos exemplos

$$
\begin{array}{lcl}
cmd & \rightarrow & \textbf{if } expr \textbf{ then } cmd \\
 & | & \textbf{if } expr \textbf{ then } cmd \textbf{ else } cmd \\
 & | & \epsilon \\
 & & \\
expr & \rightarrow & termo \textbf{ relop } termo \\
 & | & termo \\
 & & \\
termo & \rightarrow & \textbf{id} \\
 & | & \textbf{num}
\end{array}
$$

Definições regulares dos tokens

$$
\begin{aligned}
\textbf{if} &\rightarrow \texttt{if} \\
\textbf{then} &\rightarrow \texttt{then} \\
\textbf{else} &\rightarrow \texttt{else} \\
\textbf{relop} &\rightarrow \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{<>} \mid \texttt{>} \mid \texttt{>=} \\
\textbf{letra} &\rightarrow \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \\
\textbf{dígito} &\rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \mid \texttt{9} \\
\textbf{id} &\rightarrow \textbf{letra}\ (\textbf{letra} \mid \textbf{dígito})^{*} \\
\textbf{num} &\rightarrow \textbf{dígito}^{+}(\texttt{.}\ \textbf{dígito}^{+})?(\texttt{E}(\texttt{+}\mid\texttt{-})?\ \textbf{dígito}^{+})?
\end{aligned}
$$

Tratamento de espaços em branco

- Assuma que os lexemas sejam separados por espaços em brancos
- São considerados espaços em branco: sequências de espaços em branco, tabulações e quebras de linha
- O analisador léxico deve ignorar os espaços em branco
- A definição regular **ws** identifica os espaços em branco:

$$\begin{aligned} \mathbf{delim} &\rightarrow \mathbf{branco} \mid \mathbf{tabulação} \mid \mathbf{quebradelinha} \\ \mathbf{ws} &\rightarrow \mathbf{delim}^{+} \end{aligned}$$

- Se o analisador léxico identificar o padrão **ws**, ele não irá gerar um token

Especificação dos tokens

Expressão regular	Token	Valor do atributo
ws	-	-
if	if	-
then	then	-
else	else	-
id	id	Lexema
num	num	Valor numérico do lexema
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Diagramas de transição

- Um diagrama de transição é um fluxograma estilizado que delineia as ações a serem tomadas pelo analisador léxico a cada requisição de novo token por parte do *parser*
- Os estados são representados por círculos rotulados e identificam posições do diagrama
- As transições são representadas por arestas direcionadas, rotuladas por um caractere
- Uma transição do estado `X` para o estado `Y` cujo rótulo é o caractere c indica que, se a execução está no estado `X` e o próximo caractere lido é c, então a execução deve consumir c e seguir para o estado `Y`
- Um diagrama de transição é determinístico se todas as transições que partem de um estado são rotuladas por caracteres distintos

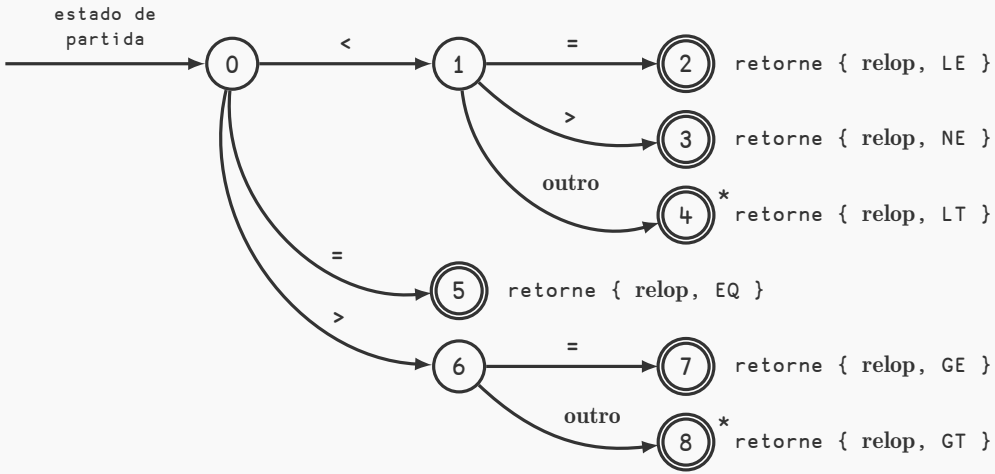
Estados e ações

- Um estado deve ser rotulado como estado de partida, o qual marca o início da execução
- Se os rótulos dos estados são numéricos, a convenção é que o estado inicial seja o de número zero (ou um)
- Alguns estados podem ter ações associadas, as quais são executadas quando a execução atinge tal estado
- Executada a ação, se existir, deve ser lido o próximo caractere c da entrada: se existir uma transição rotulada por c, a execução segue para o novo estado, indicado pela aresta; caso contrário, deve ser sinalizado um erro
- Os estados de aceitação, que indicam que um token foi reconhecido, são marcados com um círculo duplo

Retrações e erros léxicos

- Um estado de aceitação que demande o retorno do último caractere lido para o buffer de entrada é marcado um símbolo *
- Isto ocorre, por exemplo, em casos em que um token é finalizado por um espaço ou por um caractere que inicia um novo token
- Um analisador léxico pode ter vários diagramas de transição
- Se acontecer um erro no fluxo de execução de um diagrama, o ponteiro de leitura deve ser reposicionado ao ponto que estava no estado de partida e um novo diagrama deve ser seguido
- Se ocorrem erros em todos os diagramas, então há um erro léxico no programa fonte

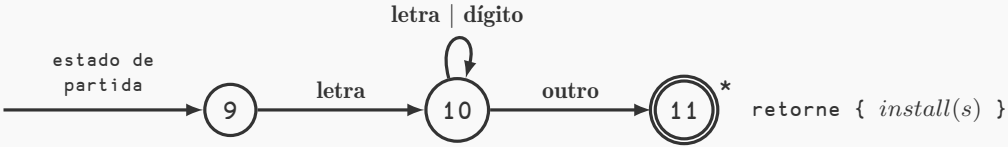
Diagrama de transição para operadores relacionais



Identificadores e palavras-chave

- Não é prático identificar as diferentes palavras-chave da linguagem por meio de diagramas de transição
- Na maioria das linguagens, as palavras-chave obedecem à mesma regra de construção dos identificadores
- Uma abordagem mais geral e efetiva é construir o diagrama de transição dos identificadores e usá-los para reconhecer tanto os identificadores quanto as palavras-chave
- Para isto, os lexemas de todas as palavras-chave devem ser inseridos na tabela de símbolos, com seus respectivos tokens e atributos
- A função $install(s)$ insere o lexema s na tabela de símbolos como um token **id**, caso s não esteja presente na tabela; caso contrário, a função retorna o token e os atributos associados a s na tabela

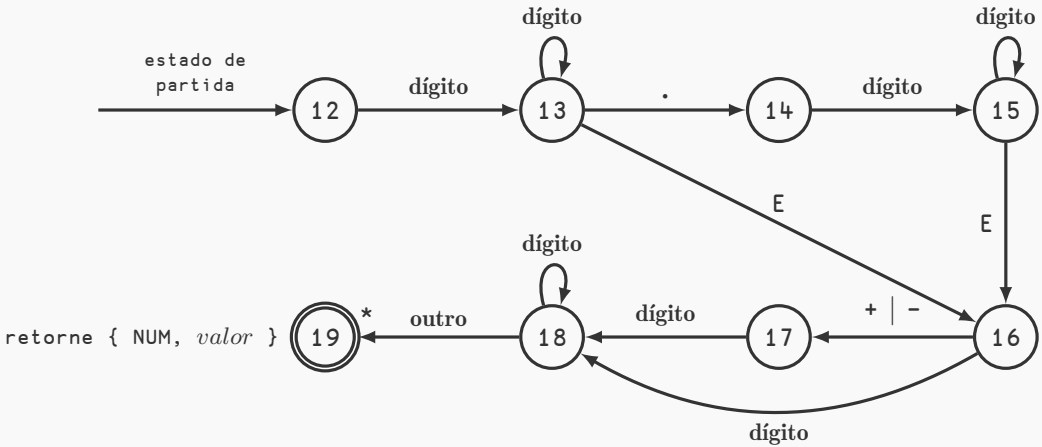
Diagrama de transição para identificadores e palavras-chave



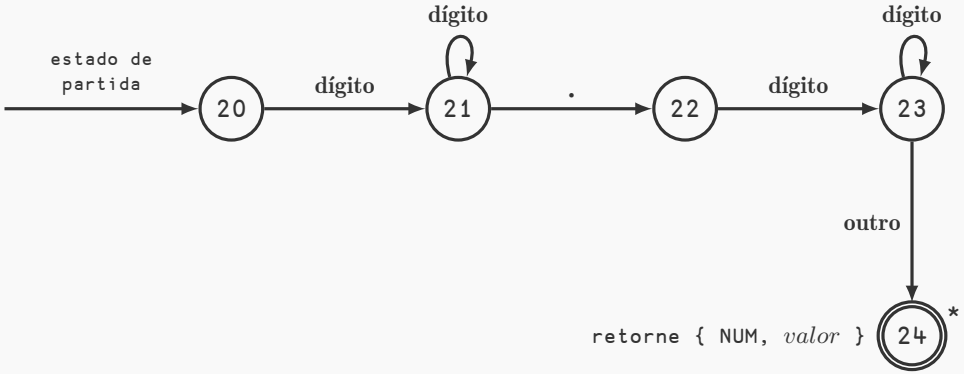
Identificação de constantes numéricas

- A identificação de tokens deve ser gulosa
- Por exemplo, se a entrada consiste em `12.3E4`, o analisador léxico não deve retornar a constante inteira `12` e nem mesmo a constante em ponto flutuante `12.3`: ele deve retornar a constante `12.3E4`
- Assim, o token deve ser o maior lexema aceito por um diagrama de transição
- Uma forma de implementar a abordagem gulosa é tratar os casos mais longos antes dos mais curtos
- Isto pode ser feito assumindo a convenção de que os estados de partida com menores rótulos devem ser testados antes dos estados com maiores rótulos e escrevendo os diagramas apropriadamente

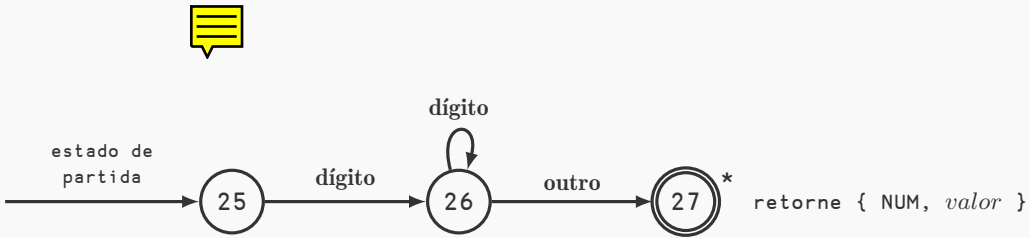
Diagramas de transição para constantes numéricas



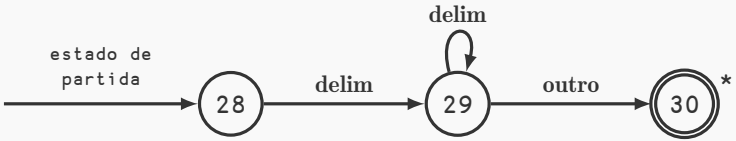
Diagramas de transição para constantes numéricas



Diagramas de transição para constantes numéricas



Diagramas de transição para espaços em branco



Implementação dos digramas de transição em C++

```
#include <bits/stdc++.h>
#include "buffer.h"

using namespace std;
using pattern = int (*)(int);
using edge = pair<int, pattern>;

template<int c> int match(int x) { return c == x; };
int wildcard(int) { return 1; };

map<int, vector<edge>> diagram {
    { 0, { { 1, match<'<'> }, { 5, match<'='> }, { 6, match<'>'> } } },
    { 1, { { 2, match<'='> }, { 3, match<'>'> }, { 4, wildcard } } },
    { 6, { { 7, match<'='> }, { 8, wildcard } } },
    { 9, { { 10, isalpha } } },
    { 10, { { 10, isalnum }, { 11, wildcard } } },
    { 28, { { 29, isspace } } },
    { 29, { { 29, isspace }, { 30, wildcard } } },
};
```

Implementação dos digramas de transição em C++

```
21 enum TokenType { IF, THEN, ELSE, RELOP, ID, DONE  };
22
23 struct Token {
24     TokenType type;
25     variant<int, string> value;
26
27     Token(TokenType t, int v) : type(t), value(v) { }
28     Token(TokenType t = DONE, const string& v = "") : type(t), value(v) { }
29 };
30
31 map<string, Token> symbolTable {
32     { "if", { IF } },
33     { "then", { THEN } },
34     { "else", { ELSE } },
35 };
```

Implementação dos digramas de transição em C++

```
37 const string relationalOperatorsNames[] { "LE", "NE", "LT", "EQ", "GE", "GT" };
38 const string keywordsNames[] { "IF", "THEN", "ELSE" };
39
40 Token install(const string& lexema)
41 {
42     if (not symbolTable.count(lexema))
43         symbolTable[lexema] = Token(ID, lexema);
44
45     return symbolTable[lexema];
46 }
```

Implementação dos digramas de transição em C++

```
48 using returnToken = function<optional<Token>(string&)>;
49
50 enum RelationalOperators { LE, NE, LT, EQ, GE, GT };
51 auto in = IOBuffer::getInstance();
52
53 map<int, returnToken> accept {
54     { 2, [](string&) { return Token(RELOP, LE); } },
55     { 3, [](string&) { return Token(RELOP, NE); } },
56     { 4, [](string&) { in.unget(); return Token(RELOP, LT); } },
57     { 5, [](string&) { return Token(RELOP, EQ); } },
58     { 7, [](string&) { return Token(RELOP, GE); } },
59     { 8, [](string&) { in.unget(); return Token(RELOP, GT); } },
60     { 11, [](string& lexema) { in.unget(); lexema.pop_back(); return install(lexema); } },
61     { 30, [](string& lexema) { in.unget(); return optional<Token>(); } }
62 };
```

Implementação dos digramas de transição em C++

```
64 ostream& operator<<(ostream& os, const Token& token)
65 {
66     switch (token.type) {
67     case RELOP:
68         os << "RELOP (" << relationalOperatorsNames[get<int>(token.value)] << ")";
69         break;
70 
71     case ID:
72         os << "ID (" << get<string>(token.value) << ")";
73         break;
74 
75     case IF:
76     case THEN:
77     case ELSE:
78         os << "Keyword (" << keywordsNames[token.type] << ")";
79         break;
80     }
81 
82     return os;
83 }
```

Implementação dos digramas de transição em C++

```
85 optional<int> nextState(int state, int lookahead)
86 {
87     for (auto [next, isMatch] : diagram.at(state))
88         if (isMatch(lookahead))
89             return next;
90
91     return { };
92 }
```

Implementação dos digramas de transição em C++

```
94 optional<Token> nextToken()
95 {
96     if (in.eof())
97         return Token(DONE);
98
99     vector<int> beginStates { 0, 9, 28 };
100    auto start = in.tell();
101
102    for (auto state : beginStates)
103    {
104        string lexema;
105
106        while (not accept.count(state))
107        {
108            auto c = in.get();
109            auto next = nextState(state, c);
110
111            if (not next)
112                break;
```

Implementação dos digramas de transição em C++

```
114             lexema.push_back((char) c);
115             state = next.value();
116         }
117
118         if (accept.count(state))
119             return accept[state](lexema);
120
121         in.seek(start);
122     }
123
124     cerr << "Lexical error!\n";
125     exit(-1);
126 }
```

Implementação dos digramas de transição em C++

```
128 int main()
129 {
130     while (true)
131     {
132         auto token = nextToken();
133 
134         if (token)
135         {
136             if (token.value().type == DONE)
137                 break;
138 
139             cout << token.value() << '\n';
140         }
141     }
142 }
```

Referências

1. **AHO**, Alfred V, **SETHI**, Ravi, **ULLMAN**, Jeffrey D. *Compiladores: Princípios, Técnicas e Ferramentas*, LTC Editora, 1995.
2. GeeksForGeeks. Flex (Fast Lexical Analyzer Generator), acesso em 04/06/2022.