Análise léxica

O papel do analisador léxico

Prof. Edson Alves

Faculdade UnB Gama

Sumário

Analisador léxico

- A análise léxica é a primeira fase de um compilador

Analisador léxico

- A análise léxica é a primeira fase de um compilador
- Um analisador léxico deve ler os caracteres da entrada e produzir uma sequência de tokens, os quais serão usados pelo *parser* durante a análise sintática

Analisador léxico

- A análise léxica é a primeira fase de um compilador
- Um analisador léxico deve ler os caracteres da entrada e produzir uma sequência de tokens, os quais serão usados pelo *parser* durante a análise sintática
- Uma forma de se construir um analisador léxico é escrever um diagrama que ilustre a estrutura dos tokens da linguagem fonte e o traduzir manualmente em um programa que os identifique

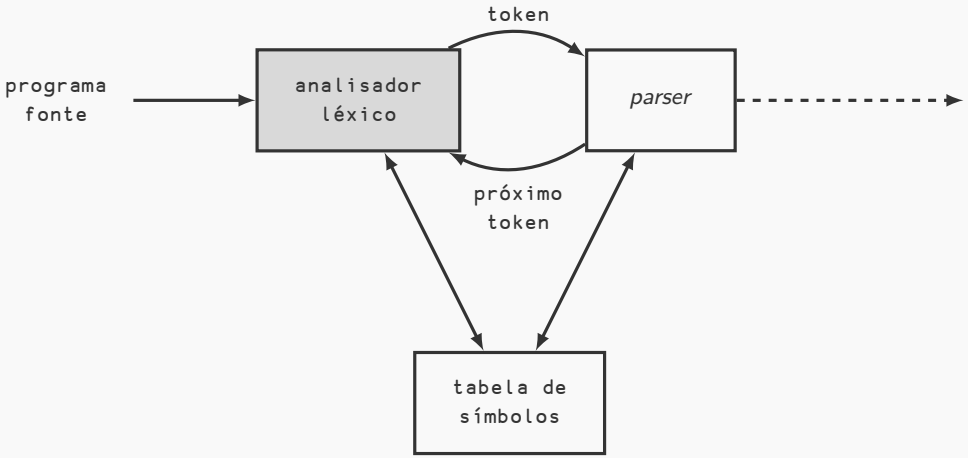
Analisador léxico

- A análise léxica é a primeira fase de um compilador
- Um analisador léxico deve ler os caracteres da entrada e produzir uma sequência de tokens, os quais serão usados pelo *parser* durante a análise sintática
- Uma forma de se construir um analisador léxico é escrever um diagrama que ilustre a estrutura dos tokens da linguagem fonte e o traduzir manualmente em um programa que os identifique
- As técnicas de construção de um analisador léxico podem ser utilizadas em outras áreas

Analisador léxico

- A análise léxica é a primeira fase de um compilador
- Um analisador léxico deve ler os caracteres da entrada e produzir uma sequência de tokens, os quais serão usados pelo *parser* durante a análise sintática
- Uma forma de se construir um analisador léxico é escrever um diagrama que ilustre a estrutura dos tokens da linguagem fonte e o traduzir manualmente em um programa que os identifique
- As técnicas de construção de um analisador léxico podem ser utilizadas em outras áreas
- Como o analisador léxico é responsável pela leitura do programa fonte, ele pode também realizar tarefas secundárias a nível de interface com o usuário, como a remoção e espaços e comentários, por exemplo

Interação entre o analisador léxico e o *parser*



Separação entre a análise léxica e a análise gramatical

Há quatro principais motivos para se separar a análise léxica da análise gramatical (*parsing*):

Separação entre a análise léxica e a análise gramatical

Há quatro principais motivos para se separar a análise léxica da análise gramatical (*parsing*):

1. A separação entre estas duas fases pode simplificar uma das duas (ou ambas)

Separação entre a análise léxica e a análise gramatical

Há quatro principais motivos para se separar a análise léxica da análise gramatical (*parsing*):

1. A separação entre estas duas fases pode simplificar uma das duas (ou ambas)
2. A eficiência do compilador é melhorada, uma vez que a separação permite o uso de técnicas especializadas, como buferização, para melhorar o desempenho da leitura da entrada e extração de tokens

Separação entre a análise léxica e a análise gramatical

Há quatro principais motivos para se separar a análise léxica da análise gramatical (*parsing*):

1. A separação entre estas duas fases pode simplificar uma das duas (ou ambas)
2. A eficiência do compilador é melhorada, uma vez que a separação permite o uso de técnicas especializadas, como buferização, para melhorar o desempenho da leitura da entrada e extração de tokens
3. A separação permite uma melhor portabilidade do compilador, uma vez que diferenças entre a captura da entrada e codificação de caracteres, em diferentes plataformas, podem ser tratadas de forma isolada na análise léxica

Separação entre a análise léxica e a análise gramatical

Há quatro principais motivos para se separar a análise léxica da análise gramatical (*parsing*):

1. A separação entre estas duas fases pode simplificar uma das duas (ou ambas)
2. A eficiência do compilador é melhorada, uma vez que a separação permite o uso de técnicas especializadas, como buferização, para melhorar o desempenho da leitura da entrada e extração de tokens
3. A separação permite uma melhor portabilidade do compilador, uma vez que diferenças entre a captura da entrada e codificação de caracteres, em diferentes plataformas, podem ser tratadas de forma isolada na análise léxica
4. A separação entre as fases permite a criação de ferramentas especilizadas para a automação da construção de analisadores léxicos e de *parsers*

Tokens, padrões e lexemas

- Tokens, padrões e lexemas são conceitos correlacionados e onipresentes na análise léxica

Tokens, padrões e lexemas

- Tokens, padrões e lexemas são conceitos correlacionados e onipresentes na análise léxica
- Token é um símbolo terminal da gramática da linguagem fonte (em geral, grafados em negrito)

Tokens, padrões e lexemas

- Tokens, padrões e lexemas são conceitos correlacionados e onipresentes na análise léxica
- Token é um símbolo terminal da gramática da linguagem fonte (em geral, grafados em negrito)
- Nas maioria das linguagens de programação, são tokens: palavras-chave, operadores, identificadores, constantes, pontuações, etc

Tokens, padrões e lexemas

- Tokens, padrões e lexemas são conceitos correlacionados e onipresentes na análise léxica
- Token é um símbolo terminal da gramática da linguagem fonte (em geral, grafados em negrito)
- Nas maioria das linguagens de programação, são tokens: palavras-chave, operadores, identificadores, constantes, pontuações, etc
- Um lexema é um conjunto de caracteres que é reconhecido como um token

Tokens, padrões e lexemas

- Tokens, padrões e lexemas são conceitos correlacionados e onipresentes na análise léxica
- Token é um símbolo terminal da gramática da linguagem fonte (em geral, grafados em negrito)
- Nas maioria das linguagens de programação, são tokens: palavras-chave, operadores, identificadores, constantes, pontuações, etc
- Um lexema é um conjunto de caracteres que é reconhecido como um token
- Um mesmo token pode ser representado por lexemas distintos (por exemplo, 1 e 42 são lexemas distintos para o token `NUM`)

Tokens, padrões e lexemas

- Tokens, padrões e lexemas são conceitos correlacionados e onipresentes na análise léxica
- Token é um símbolo terminal da gramática da linguagem fonte (em geral, grafados em negrito)
- Nas maioria das linguagens de programação, são tokens: palavras-chave, operadores, identificadores, constantes, pontuações, etc
- Um lexema é um conjunto de caracteres que é reconhecido como um token
- Um mesmo token pode ser representado por lexemas distintos (por exemplo, 1 e 42 são lexemas distintos para o token `NUM`)
- Um padrão descreve o conjunto de lexemas que podem representar um token em particular

Atributos para tokens

- Quando dois ou mais lexemas estão associados a um mesmo token, o analisador léxico deve prover informações adicionais para as fases subsequentes, para que elas possam distinguí-los

Atributos para tokens

- Quando dois ou mais lexemas estão associados a um mesmo token, o analisador léxico deve prover informações adicionais para as fases subsequentes, para que elas possam distinguí-los
- Estas informações são os atributos do token

Atributos para tokens

- Quando dois ou mais lexemas estão associados a um mesmo token, o analisador léxico deve prover informações adicionais para as fases subsequentes, para que elas possam distinguí-los
- Estas informações são os atributos do token
- Deste modo, o analisador léxico deve identificar os tokens e seus respectivos atributos, caso existam

Atributos para tokens

- Quando dois ou mais lexemas estão associados a um mesmo token, o analisador léxico deve prover informações adicionais para as fases subsequentes, para que elas possam distinguí-los
- Estas informações são os atributos do token
- Deste modo, o analisador léxico deve identificar os tokens e seus respectivos atributos, caso existam
- Os tokens influenciam as decisões da análise gramatical

Atributos para tokens

- Quando dois ou mais lexemas estão associados a um mesmo token, o analisador léxico deve prover informações adicionais para as fases subsequentes, para que elas possam distinguí-los
- Estas informações são os atributos do token
- Deste modo, o analisador léxico deve identificar os tokens e seus respectivos atributos, caso existam
- Os tokens influenciam as decisões da análise gramatical
- Os atributos influenciam a tradução dos tokens

Atributos para tokens

- Quando dois ou mais lexemas estão associados a um mesmo token, o analisador léxico deve prover informações adicionais para as fases subsequentes, para que elas possam distinguí-los
- Estas informações são os atributos do token
- Deste modo, o analisador léxico deve identificar os tokens e seus respectivos atributos, caso existam
- Os tokens influenciam as decisões da análise gramatical
- Os atributos influenciam a tradução dos tokens
- Em tokens numéricos, o valor do número representado pelo lexema pode ser o atributo token

Atributos para tokens

- Quando dois ou mais lexemas estão associados a um mesmo token, o analisador léxico deve prover informações adicionais para as fases subsequentes, para que elas possam distinguí-los
- Estas informações são os atributos do token
- Deste modo, o analisador léxico deve identificar os tokens e seus respectivos atributos, caso existam
- Os tokens influenciam as decisões da análise gramatical
- Os atributos influenciam a tradução dos tokens
- Em tokens numéricos, o valor do número representado pelo lexema pode ser o atributo token
- No caso de identificadores, o próprio lexema pode ser o atributo do token

Erros léxicos

- Determinados erros não podem ser detectados em nível léxico

Erros léxicos

- Determinados erros não podem ser detectados em nível léxico
- Por exemplo, na expressão em C++

```
fi (a == f(x)) {
    ...
}
```

o analisador léxico identificaria `fi` como um identificador válido, e só na análise gramatical é que seria detectado o erro de digitação da palavra-chave `if`

Erros léxicos

- Determinados erros não podem ser detectados em nível léxico
- Por exemplo, na expressão em C++

```
fi (a == f(x)) {
    ...
}
```

  o analisador léxico identificaria `fi` como um identificador válido, e só na análise gramatical é que seria detectado o erro de digitação da palavra-chave `if`
- Os erros léxicos mais comuns são aqueles onde o analisador léxico não consegue associar o prefixo lido a nenhum dos padrões associados aos tokens da linguagem

Erros léxicos

- Determinados erros não podem ser detectados em nível léxico
- Por exemplo, na expressão em C++

```
fi (a == f(x)) {
    ...
}
```

  o analisador léxico identificaria `fi` como um identificador válido, e só na análise gramatical é que seria detectado o erro de digitação da palavra-chave `if`
- Os erros léxicos mais comuns são aqueles onde o analisador léxico não consegue associar o prefixo lido a nenhum dos padrões associados aos tokens da linguagem
- Neste ponto, o analisador léxico pode abordar a leitura, emitindo uma mensagem de erro

Erros léxicos

- Determinados erros não podem ser detectados em nível léxico
- Por exemplo, na expressão em C++

```
fi (a == f(x)) {
    ...
}
```

  o analisador léxico identificaria `fi` como um identificador válido, e só na análise gramatical é que seria detectado o erro de digitação da palavra-chave `if`
- Os erros léxicos mais comuns são aqueles onde o analisador léxico não consegue associar o prefixo lido a nenhum dos padrões associados aos tokens da linguagem
- Neste ponto, o analisador léxico pode abordar a leitura, emitindo uma mensagem de erro
- Outra alternativa é tentar tratar o erro de alguma maneira

Ações de recuperação de erros

Há quatro ações que configuram tentativas de recuperação de erros léxicos:

Ações de recuperação de erros

Há quatro ações que configuram tentativas de recuperação de erros léxicos:

- remover um caractere estranho da entrada

Ações de recuperação de erros

Há quatro ações que configuram tentativas de recuperação de erros léxicos:

- remover um caractere estranho da entrada
- inserir um caractere ausente

Ações de recuperação de erros

Há quatro ações que configuram tentativas de recuperação de erros léxicos:

- remover um caractere estranho da entrada
- inserir um caractere ausente
- substituir um dos caracteres incorretos por um caractere correto

Ações de recuperação de erros

Há quatro ações que configuram tentativas de recuperação de erros léxicos:

- remover um caractere estranho da entrada
- inserir um caractere ausente
- substituir um dos caracteres incorretos por um caractere correto
- transpor dois caracteres adjacentes

Ações de recuperação de erros

Há quatro ações que configuram tentativas de recuperação de erros léxicos:

- remover um caractere estranho da entrada
- inserir um caractere ausente
- substituir um dos caracteres incorretos por um caractere correto
- transpor dois caracteres adjacentes

Se uma ou mais ações conseguem tornar o prefixo em um token válido, o analisador podem indicar ao usuário a sequência de ações como sugestão de correção do programa fonte, ou mesmo prosseguir assumindo esta correção.

Buferização

- Como a análise léxica é a única fase do compilador que lê os caracteres do programa fonte, um a um, ela pode concentrar uma parte significativa do tempo de execução do compilador

Buferização

- Como a análise léxica é a única fase do compilador que lê os caracteres do programa fonte, um a um, ela pode concentrar uma parte significativa do tempo de execução do compilador
- Isto porque o acesso à entrada (em geral, um arquivo em disco) pode ser o gargalo, em termos de performance, do compilador

Buferização

- Como a análise léxica é a única fase do compilador que lê os caracteres do programa fonte, um a um, ela pode concentrar uma parte significativa do tempo de execução do compilador
- Isto porque o acesso à entrada (em geral, um arquivo em disco) pode ser o gargalo, em termos de performance, do compilador
- A buferização consiste no uso de um ou mais vetores auxiliares (*buffers*), que permitem a leitura da entrada em blocos, de modo que o analisador léxico leia os caracteres a partir destes *buffers*, os quais são atualizados e preenchidos à medida do necessário

Buferização

- Como a análise léxica é a única fase do compilador que lê os caracteres do programa fonte, um a um, ela pode concentrar uma parte significativa do tempo de execução do compilador
- Isto porque o acesso à entrada (em geral, um arquivo em disco) pode ser o gargalo, em termos de performance, do compilador
- A buferização consiste no uso de um ou mais vetores auxiliares (*buffers*), que permitem a leitura da entrada em blocos, de modo que o analisador léxico leia os caracteres a partir destes *buffers*, os quais são atualizados e preenchidos à medida do necessário
- Com a buferização os acesso aos disco são reduzidos e a leitura dos caracteres passa a ser feita em memória, com acessos consideravelmente mais rápidos

Estratégias para implementação de analisadores léxicos

Há três estratégias gerais para se implementar um analisador léxico, cada uma delas tratando a buferização de modo diferente. São elas, da mais simples para a mais complexa:

Estratégias para implementação de analisadores léxicos

Há três estratégias gerais para se implementar um analisador léxico, cada uma delas tratando a buferização de modo diferente. São elas, da mais simples para a mais complexa:

- Usar um gerador de analisador léxico, a partir de uma entrada especificada a partir de expressões regulares. A buferização é tratada pelo próprio gerador

Estratégias para implementação de analisadores léxicos

Há três estratégias gerais para se implementar um analisador léxico, cada uma delas tratando a buferização de modo diferente. São elas, da mais simples para a mais complexa:

- Usar um gerador de analisador léxico, a partir de uma entrada especificada a partir de expressões regulares. A buferização é tratada pelo próprio gerador
- Escrever o analisador léxico em alguma linguagem de programação convencional (C, C++, etc). A buferização fica atrelada aos mecanismos de I/O da linguagem

Estratégias para implementação de analisadores léxicos

Há três estratégias gerais para se implementar um analisador léxico, cada uma delas tratando a buferização de modo diferente. São elas, da mais simples para a mais complexa:

- Usar um gerador de analisador léxico, a partir de uma entrada especificada a partir de expressões regulares. A buferização é tratada pelo próprio gerador
- Escrever o analisador léxico em alguma linguagem de programação convencional (C, C++, etc). A buferização fica atrelada aos mecanismos de I/O da linguagem
- Escrever o analisador em linguagem de montagem e tratar explicitamente a leitura da entrada e a buferização

Pares de *buffers*

- Na técnica de pares de *buffers*, um *buffer* (região contígua da memória) é dividido em duas metades, com N caracteres cada

Pares de *buffers*

- Na técnica de pares de *buffers*, um *buffer* (região contígua da memória) é dividido em duas metades, com N caracteres cada
- Em geral, N corresponde ao tamanho de um bloco do disco (por exemplo, 1024 ou 4096 caracteres)

Pares de *buffers*

- Na técnica de pares de *buffers*, um *buffer* (região contígua da memória) é dividido em duas metades, com N caracteres cada
- Em geral, N corresponde ao tamanho de um bloco do disco (por exemplo, 1024 ou 4096 caracteres)
- Cada metade do *buffer* é preenchida de uma única vez, por meio da chamada de uma função de leitura do sistema

Pares de *buffers*

- Na técnica de pares de *buffers*, um *buffer* (região contígua da memória) é dividido em duas metades, com N caracteres cada
- Em geral, N corresponde ao tamanho de um bloco do disco (por exemplo, 1024 ou 4096 caracteres)
- Cada metade do *buffer* é preenchida de uma única vez, por meio da chamada de uma função de leitura do sistema
- Caso restem na entrada menos do que N caracteres, é inserido um caractere especial no *buffer* para indicar o fim da entrada (em geral, o caractere `EOF` - *end of file*)

Pares de *buffers*

- Na técnica de pares de *buffers*, um *buffer* (região contígua da memória) é dividido em duas metades, com N caracteres cada
- Em geral, N corresponde ao tamanho de um bloco do disco (por exemplo, 1024 ou 4096 caracteres)
- Cada metade do *buffer* é preenchida de uma única vez, por meio da chamada de uma função de leitura do sistema
- Caso restem na entrada menos do que N caracteres, é inserido um caractere especial no *buffer* para indicar o fim da entrada (em geral, o caractere `EOF` - *end of file*)
- Usando esta técnica, os tokens devem ser extraídos do *buffer*, sem o uso de chamadas individuais da rotina que lê um caractere da entrada

Dois ponteiros

- Os tokens podem ser extraídos do par de *buffers* por meio do uso de dois ponteiros L e R

Dois ponteiros

- Os tokens podem ser extraídos do par de *buffers* por meio do uso de dois ponteiros L e R
- Uma cadeia de caracteres delimitada por este dois ponteiros é o lexema atual

Dois ponteiros

- Os tokens podem ser extraídos do par de *buffers* por meio do uso de dois ponteiros L e R
- Uma cadeia de caracteres delimitada por este dois ponteiros é o lexema atual
- Inicialmente, L e R apontam para o primeiro caractere do próximo lexema a ser identificado

Dois ponteiros

- Os tokens podem ser extraídos do par de *buffers* por meio do uso de dois ponteiros L e R
- Uma cadeia de caracteres delimitada por este dois ponteiros é o lexema atual
- Inicialmente, L e R apontam para o primeiro caractere do próximo lexema a ser identificado
- O ponteiro R então avança até que o padrão de um token seja reconhecido

Dois ponteiros

- Os tokens podem ser extraídos do par de *buffers* por meio do uso de dois ponteiros L e R
- Uma cadeia de caracteres delimitada por este dois ponteiros é o lexema atual
- Inicialmente, L e R apontam para o primeiro caractere do próximo lexema a ser identificado
- O ponteiro R então avança até que o padrão de um token seja reconhecido
- Daí o lexema é processado e ambos ponteiros se movem para o primeiro caractere após o lexema

Dois ponteiros

- Os tokens podem ser extraídos do par de *buffers* por meio do uso de dois ponteiros L e R
- Uma cadeia de caracteres delimitada por este dois ponteiros é o lexema atual
- Inicialmente, L e R apontam para o primeiro caractere do próximo lexema a ser identificado
- O ponteiro R então avança até que o padrão de um token seja reconhecido
- Daí o lexema é processado e ambos ponteiros se movem para o primeiro caractere após o lexema
- Neste cenário, espaços em branco e comentários são padrões que não produzem tokens

Atualização dos *buffers* e o ponteiro R

- Se o ponteiro R tentar se deslocar para além do meio do *buffer*, será preciso preencher a metade direita com N novos caracteres antes deste avanço

Atualização dos *buffers* e o ponteiro R

- Se o ponteiro R tentar se deslocar para além do meio do *buffer*, será preciso preencher a metade direita com N novos caracteres antes deste avanço
- De forma semelhante, se R atingir a extremidade direita do *buffer*, a metade à esquerda deve ser devidamente atualizada

Atualização dos *buffers* e o ponteiro R

- Se o ponteiro R tentar se deslocar para além do meio do *buffer*, será preciso preencher a metade direita com N novos caracteres antes deste avanço
- De forma semelhante, se R atingir a extremidade direita do *buffer*, a metade à esquerda deve ser devidamente atualizada
- Após esta atualização, R deve retornar para a primeira posição do *buffer*

Atualização dos *buffers* e o ponteiro R

- Se o ponteiro R tentar se deslocar para além do meio do *buffer*, será preciso preencher a metade direita com N novos caracteres antes deste avanço
- De forma semelhante, se R atingir a extremidade direita do *buffer*, a metade à esquerda deve ser devidamente atualizada
- Após esta atualização, R deve retornar para a primeira posição do *buffer*
- O uso de um par de *buffers* e dois ponteiros tem uma limitação clara: o lexema pode ter, no máxixmo, $2N$ caracteres

Atualização dos *buffers* e o ponteiro R

- Se o ponteiro R tentar se deslocar para além do meio do *buffer*, será preciso preencher a metade direita com N novos caracteres antes deste avanço
- De forma semelhante, se R atingir a extremidade direita do *buffer*, a metade à esquerda deve ser devidamente atualizada
- Após esta atualização, R deve retornar para a primeira posição do *buffer*
- O uso de um par de *buffers* e dois ponteiros tem uma limitação clara: o lexema pode ter, no máxixmo, $2N$ caracteres
- O recuo de R, se necessário, também é limitado pela posição que L ocupa

Avanço de R em um par de *buffers*

1: **if** R está no fim da primeira metade **then**
2: Atualize a segunda metade com a leitura de N novos caracteres
3: $R \leftarrow R + 1$
4: **else if** R está no fim da segunda metade **then**
5: Atualize a primeira metade com a leitura de N novos caracteres
6: $R \leftarrow 0$ ▷ *Assuma que os índices de buffer começem em zero*
7: **else if** **then**
8: $R \leftarrow R + 1$

Sentinelas

- O uso de um valor sentinela no fim de cada metade do *buffer* permite a redução dos testes para o avanço de R

Sentinelas

- O uso de um valor sentinela no fim de cada metade do *buffer* permite a redução dos testes para o avanço de R
- Além disso, o valor sentinela em outra posição do *buffer* indica o fim da entrada

Sentinelas

- O uso de um valor sentinela no fim de cada metade do *buffer* permite a redução dos testes para o avanço de R
- Além disso, o valor sentinela em outra posição do *buffer* indica o fim da entrada
- A redução do número de testes (de dois para um, na maioria dos casos) decorrente do uso de sentinelas leva a um ganho de performance do analisador léxico e, consequentemente, do compilador

Sentinelas

- O uso de um valor sentinela no fim de cada metade do *buffer* permite a redução dos testes para o avanço de R
- Além disso, o valor sentinela em outra posição do *buffer* indica o fim da entrada
- A redução do número de testes (de dois para um, na maioria dos casos) decorrente do uso de sentinelas leva a um ganho de performance do analisador léxico e, consequentemente, do compilador
- O valor sentinela (em geral, `EOF`) deve ser diferente de qualquer caractere válido da entrada, para evitar um encerramento prematuro da entrada, caso tal caractere faça parte da entrada

Atualização de R com o uso de sentinelas

1: $R \leftarrow R + 1$

2: **if** $R =$ `EOF` **then**
3: **if** R está no fim da primeira metade **then**
4: Atualize a segunda metade com a leitura de N novos caracteres
5: $R \leftarrow R + 1$
6: **else if** R está no fim da segunda metade **then**
7: Atualize a primeira metade com a leitura de N novos caracteres
8: $R \leftarrow 0$ ▷ *Assuma que os índices de buffer começem em zero*
9: **else** ▷ `EOF` *está no buffer, indicando o fim da entrada*
10: Finalize a análise léxica

Módulo buffer.h

```
1 #ifndef BUFFER_H
2 #define BUFFER_H
3
4 const int N { 4 };
5
6 class IOBuffer {
7 public:
8     static IOBuffer& getInstance();
9
10    bool eof() const;
11    int tell() const;
12    void seek(int pos);
13
14    int get();
15    void unget();
```

Módulo buffer.h

```
17 private:
18     IOBuffer();
19
20     int pos, last_update;
21     char buffer[2*N + 2];
22
23     void update();
24 };
25
26 #endif
```

Módulo buffer.cpp

```cpp
#include <iostream>
#include "buffer.h"

using namespace std;

IOBuffer&
IOBuffer::getInstance()
{
    static IOBuffer buffer;
    return  buffer;
}

IOBuffer::IOBuffer() : pos(2*N), last_update(1)
{
    buffer[N] = buffer[2*N + 1] = EOF;
    update();
}
```

Módulo buffer.cpp

```
19 void IOBuffer::update()
20 {
21     ++pos;
22
23     if (buffer[pos] != EOF)
24         return;
25
26     if (pos == 2*N + 1)
27     {
28         pos = 0;
29
30         if (last_update == 1)
31         {
32             auto size = fread(buffer, sizeof(char), N, stdin);
33
34             if (size < N)
35                 buffer[size] = EOF;
36
37             last_update = 0;
38         }
39     }
```

Módulo buffer.cpp

```
40      else if (pos == N)
41      {
42          if (last_update == 0)
43          {
44              auto size = fread(buffer + N + 1, sizeof(char), N, stdin);
45
46              if (size < N)
47                  buffer[N + 1 + size] = EOF;
48
49              last_update = 1;
50          }
51
52          ++pos;
53      }
54  }
55
56  bool
57  IOBuffer::eof() const
58  {
59      return buffer[pos] == EOF;
60  }
```

Módulo buffer.cpp

```
62 int
63 IOBuffer::tell() const
64 {
65     return pos;
66 }
67
68 void
69 IOBuffer::seek(int p)
70 {
71     pos = p;
72 }
73
74 int
75 IOBuffer::get()
76 {
77     auto c = buffer[pos];
78     update();
79
80     return c;
81 }
```

Módulo buffer.cpp

```
83 void
84 IOBuffer::unget()
85 {
86     --pos;
87
88     if (pos < 0)
89         pos = 2*N;
90     else if (pos == N)
91         --pos;
92 }
```

Alfabetos

Definição de alfabeto

Um alfabeto, ou classe de caracteres, é um conjunto finito de símbolos.

Alfabetos

Definição de alfabeto

Um alfabeto, ou classe de caracteres, é um conjunto finito de símbolos.

Exemplos de alfabetos: ASCII, EBCDIC, a alfabeto binário $\{ 0, 1 \}$, os dígitos decimais, etc.

Cadeias

Definição de cadeia

Uma cadeia sobre um alfabeto $\mathcal{A}$ é uma sequência finita de elementos de $\mathcal{A}$. Os termos sentença, palavra e string são geralmente usados como sinônimos de cadeia.

Conceitos associados à cadeias

- O comprimento (número de caracteres) de uma cadeia s é denotado por $|s|$

Conceitos associados à cadeias

- O comprimento (número de caracteres) de uma cadeia s é denotado por $|s|$
- A cadeia vazia ϵ tem comprimento igual a zero

Conceitos associados à cadeias

- O comprimento (número de caracteres) de uma cadeia s é denotado por $|s|$
- A cadeia vazia ϵ tem comprimento igual a zero
- Um prefixo de s é uma cadeia obtida pela remoção de zero ou mais caracteres do fim de s

Conceitos associados à cadeias

- O comprimento (número de caracteres) de uma cadeia s é denotado por $|s|$
- A cadeia vazia ϵ tem comprimento igual a zero
- Um prefixo de s é uma cadeia obtida pela remoção de zero ou mais caracteres do fim de s
- Um sufixo de s é uma cadeia obtida pela remoção de zero ou mais caracteres do início de s

Conceitos associados à cadeias

- O comprimento (número de caracteres) de uma cadeia s é denotado por $|s|$
- A cadeia vazia ϵ tem comprimento igual a zero
- Um prefixo de s é uma cadeia obtida pela remoção de zero ou mais caracteres do fim de s
- Um sufixo de s é uma cadeia obtida pela remoção de zero ou mais caracteres do início de s
- Uma subcadeia de s é uma cadeia obtida pela remoção de um prefixo e de um sufixo de s

Conceitos associados à cadeias

- O comprimento (número de caracteres) de uma cadeia s é denotado por $|s|$
- A cadeia vazia ϵ tem comprimento igual a zero
- Um prefixo de s é uma cadeia obtida pela remoção de zero ou mais caracteres do fim de s
- Um sufixo de s é uma cadeia obtida pela remoção de zero ou mais caracteres do início de s
- Uma subcadeia de s é uma cadeia obtida pela remoção de um prefixo e de um sufixo de s
- Um prefixo, sufixo ou subcadeia de s são ditos próprios se diferem de ϵ e de s

Conceitos associados à cadeias

- O comprimento (número de caracteres) de uma cadeia s é denotado por $|s|$
- A cadeia vazia ϵ tem comprimento igual a zero
- Um prefixo de s é uma cadeia obtida pela remoção de zero ou mais caracteres do fim de s
- Um sufixo de s é uma cadeia obtida pela remoção de zero ou mais caracteres do início de s
- Uma subcadeia de s é uma cadeia obtida pela remoção de um prefixo e de um sufixo de s
- Um prefixo, sufixo ou subcadeia de s são ditos próprios se diferem de ϵ e de s
- Um subsequência de s é uma cadeia obtida pela remoção de zero ou mais símbolos de s, não necessariamente contíguos

Linguagens

Definição de linguagem

Uma linguagem é um conjunto de cadeias sobre algum alfabeto $\mathcal{A}$ fixo.

Linguagens

Definição de linguagem

Uma linguagem é um conjunto de cadeias sobre algum alfabeto $\mathcal{A}$ fixo.

Esta definição contempla também linguagens abstratas como $\emptyset$ (o conjunto vazio), ou $\{ \epsilon \}$, o conjunto contendo apenas a cadeia vazia.

Operações em cadeias

- Se x e y são duas cadeias, então a concatenação de x e y, denotada xy, é a cadeia formada pelo acréscimo, ao final de x, de todos os caracteres de y, na mesma ordem

Operações em cadeias

- Se x e y são duas cadeias, então a concatenação de x e y, denotada xy, é a cadeia formada pelo acréscimo, ao final de x, de todos os caracteres de y, na mesma ordem
- Por exemplo, se $x =$ `"rodo"` e $y =$ `"via"`, então $xy =$ `"rodovia"`

Operações em cadeias

- Se x e y são duas cadeias, então a concatenação de x e y, denotada xy, é a cadeia formada pelo acréscimo, ao final de x, de todos os caracteres de y, na mesma ordem
- Por exemplo, se $x =$ `"rodo"` e $y =$ `"via"`, então $xy =$ `"rodovia"`
- A cadeia vazia ϵ é o elemento neutro da concatenação

Operações em cadeias

- Se x e y são duas cadeias, então a concatenação de x e y, denotada xy, é a cadeia formada pelo acréscimo, ao final de x, de todos os caracteres de y, na mesma ordem
- Por exemplo, se $x =$ `"rodo"` e $y =$ `"via"`, então $xy =$ `"rodovia"`
- A cadeia vazia ϵ é o elemento neutro da concatenação
- Se a concatenação for visualizada como um produto, é possível definir uma "exponenciação" de cadeias

Operações em cadeias

- Se x e y são duas cadeias, então a concatenação de x e y, denotada xy, é a cadeia formada pelo acréscimo, ao final de x, de todos os caracteres de y, na mesma ordem
- Por exemplo, se $x =$ `"rodo"` e $y =$ `"via"`, então $xy =$ `"rodovia"`
- A cadeia vazia ϵ é o elemento neutro da concatenação
- Se a concatenação for visualizada como um produto, é possível definir uma "exponenciação" de cadeias
- Seja s uma cadeia e n um natural. Então

Operações em cadeias

- Se x e y são duas cadeias, então a concatenação de x e y, denotada xy, é a cadeia formada pelo acréscimo, ao final de x, de todos os caracteres de y, na mesma ordem
- Por exemplo, se $x =$ `"rodo"` e $y =$ `"via"`, então $xy =$ `"rodovia"`
- A cadeia vazia ϵ é o elemento neutro da concatenação
- Se a concatenação for visualizada como um produto, é possível definir uma "exponenciação" de cadeias
- Seja s uma cadeia e n um natural. Então
  1. $s^0 = \epsilon$

Operações em cadeias

- Se x e y são duas cadeias, então a concatenação de x e y, denotada xy, é a cadeia formada pelo acréscimo, ao final de x, de todos os caracteres de y, na mesma ordem
- Por exemplo, se $x =$ `"rodo"` e $y =$ `"via"`, então $xy =$ `"rodovia"`
- A cadeia vazia ϵ é o elemento neutro da concatenação
- Se a concatenação for visualizada como um produto, é possível definir uma "exponenciação" de cadeias
- Seja s uma cadeia e n um natural. Então
  1. $s^0 = \epsilon$
  2. $s^n = ss^{n-1}$

Operações em linguagens

Sejam L e M duas linguagens. São definidas as seguintes operações sobre linguagens:

Operações em linguagens

Sejam L e M duas linguagens. São definidas as seguintes operações sobre linguagens:

Operação	Notação	Definição
união	$L \cup M$	$L \cup M = \{ s \mid s \in L \ \vee \ s \in M \}$
concatenação	LM	$LM = \{ st \mid s \in L \ \wedge \ t \in M \}$
fechamento de Kleene	L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$
fechamento positivo	L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Exemplos de operações em linguagens

Seja $L = \{$ `A, B, C, ... Z, a, b, c, ... z` $\}$ e $M = \{$ `0, 1, 2, ... 9` $\}$. Então:

Exemplos de operações em linguagens

Seja $L = \{$ `A, B, C, ... Z, a, b, c, ... z` $\}$ e $M = \{$ `0, 1, 2, ... 9` $\}$. Então:

1. $L \cup M$ é o conjunto de letras e dígitos

Exemplos de operações em linguagens

Seja $L = \{$ `A, B, C, ... Z, a, b, c, ... z` $\}$ e $M = \{$ `0, 1, 2, ... 9` $\}$. Então:

1. $L \cup M$ é o conjunto de letras e dígitos
2. LM é o conjunto de cadeias formadas por uma letra, seguida de um dígito

Exemplos de operações em linguagens

Seja $L = \{$ `A, B, C, ... Z, a, b, c, ... z` $\}$ e $M = \{$ `0, 1, 2, ... 9` $\}$. Então:

1. $L \cup M$ é o conjunto de letras e dígitos
2. LM é o conjunto de cadeias formadas por uma letra, seguida de um dígito
3. L^4 é o conjunto de todas as cadeias formadas por exatamente quatro letras

Exemplos de operações em linguagens

Seja $L = \{$ `A, B, C, ... Z, a, b, c, ... z` $\}$ e $M = \{$ `0, 1, 2, ... 9` $\}$. Então:

1. $L \cup M$ é o conjunto de letras e dígitos
2. LM é o conjunto de cadeias formadas por uma letra, seguida de um dígito
3. L^4 é o conjunto de todas as cadeias formadas por exatamente quatro letras
4. L^* é o conjunto de todas as cadeias formadas por letras, incluíndo a cadeia ϵ

Exemplos de operações em linguagens

Seja $L = \{$ `A, B, C, ... Z, a, b, c, ... z` $\}$ e $M = \{$ `0, 1, 2, ... 9` $\}$. Então:

1. $L \cup M$ é o conjunto de letras e dígitos
2. LM é o conjunto de cadeias formadas por uma letra, seguida de um dígito
3. L^4 é o conjunto de todas as cadeias formadas por exatamente quatro letras
4. L^* é o conjunto de todas as cadeias formadas por letras, incluíndo a cadeia ϵ
5. $L(L \cup D)^*$ é o conjunto de cadeias de letras e dígitos, que iniciam com uma letra

Exemplos de operações em linguagens

Seja $L = \{$ `A, B, C, ... Z, a, b, c, ... z` $\}$ e $M = \{$ `0, 1, 2, ... 9` $\}$. Então:

1. $L \cup M$ é o conjunto de letras e dígitos
2. LM é o conjunto de cadeias formadas por uma letra, seguida de um dígito
3. L^4 é o conjunto de todas as cadeias formadas por exatamente quatro letras
4. L^* é o conjunto de todas as cadeias formadas por letras, incluíndo a cadeia ϵ
5. $L(L \cup D)^*$ é o conjunto de cadeias de letras e dígitos, que iniciam com uma letra
6. D^+ é o conjunto de cadeias formadas por um ou mais dígitos

Expressões regulares

Definição de expressão regular

Sejam Σ um alfabeto. As expressões regulares sobre Σ são definidas pelas seguintes regras, onde cada expressão regular define uma linguagem:

1. ϵ é uma expressão regular que denota a linguagem $\{ \epsilon \}$
2. Se $a \in \Sigma$, então a é uma expressão regular que denota a linguagem $\{ a \}$
3. Se r e s são duas expressões regulares que denotam as linguagens $L(r)$ e $L(s)$, então
   (a) (r) é uma expressão regular que denota $L(r)$
   (b) $(r)|(s)$ é uma expressão regular que denota $L(r) \cup L(s)$
   (c) $(r)(s)$ é uma expressão regular que denota $L(r)L(s)$
   (d) $(r)^*$ é uma expressão regular que denota $(L(r))^*$

Expressões regulares e parêntesis

O uso de parêntesis em expressões regulares pode ser reduzido se forem adotadas as seguintes convenções:

Expressões regulares e parêntesis

O uso de parêntesis em expressões regulares pode ser reduzido se forem adotadas as seguintes convenções:

1. o operador unário $*$ possui a maior precedência e é associativo à esquerda

Expressões regulares e parêntesis

O uso de parêntesis em expressões regulares pode ser reduzido se forem adotadas as seguintes convenções:

1. o operador unário $*$ possui a maior precedência e é associativo à esquerda
2. a concatenação tem a segunda maior precedência e é associativa à esquerda

Expressões regulares e parêntesis

O uso de parêntesis em expressões regulares pode ser reduzido se forem adotadas as seguintes convenções:

1. o operador unário $*$ possui a maior precedência e é associativo à esquerda
2. a concatenação tem a segunda maior precedência e é associativa à esquerda
3. o operador $|$ tem a menor precedência e é associativo à esquerda

Expressões regulares e parêntesis

O uso de parêntesis em expressões regulares pode ser reduzido se forem adotadas as seguintes convenções:

1. o operador unário $*$ possui a maior precedência e é associativo à esquerda
2. a concatenação tem a segunda maior precedência e é associativa à esquerda
3. o operador $|$ tem a menor precedência e é associativo à esquerda

Neste cenário, a expressão regular $(a) \mid ((b)^* \ (c))$ equivale a $a \mid b^* \ c$.

Exemplos de expressões regulares

Seja $\Sigma = \{ a, b \}$. Então

Exemplos de expressões regulares

Seja $\Sigma = \{ a, b \}$. Então

- $a \mid b$ denota a linguagem $\{ a, b \}$

Exemplos de expressões regulares

Seja $\Sigma = \{ a, b \}$. Então

- $a \mid b$ denota a linguagem $\{ a, b \}$
- $(a \mid b)(a \mid b)$ denota $\{ aa, ab, ba, bb \}$

Exemplos de expressões regulares

Seja $\Sigma = \{ a, b \}$. Então

- $a \mid b$ denota a linguagem $\{ a, b \}$
- $(a \mid b)(a \mid b)$ denota $\{ aa, ab, ba, bb \}$
- a^* denota $\{ \epsilon, a, aa, aaa, \ldots \}$

Exemplos de expressões regulares

Seja $\Sigma = \{ a, b \}$. Então

- $a \mid b$ denota a linguagem $\{ a, b \}$
- $(a \mid b)(a \mid b)$ denota $\{ aa, ab, ba, bb \}$
- a^* denota $\{ \epsilon, a, aa, aaa, \ldots \}$
- $(a \mid b)^*$ denota todas as cadeias formas por zero ou mais instâncias de a ou de b

Exemplos de expressões regulares

Seja $\Sigma = \{ a, b \}$. Então

- $a \mid b$ denota a linguagem $\{ a, b \}$
- $(a \mid b)(a \mid b)$ denota $\{ aa, ab, ba, bb \}$
- a^* denota $\{ \epsilon, a, aa, aaa, \ldots \}$
- $(a \mid b)^*$ denota todas as cadeias formas por zero ou mais instâncias de a ou de b
- $a \mid a^* \, b$ denota a cadeia a e todas as cadeias iniciadas por zero ou mais a's, seguidos de um b

Propriedades das expressões regulares

Sejam r, s, t expressões regulares. Valem as seguintes propriedades:

Axioma	Descrição
$r\|s = s\|r$	$\|$ é comutativo
$r\|(s\|t) = (r\|s)\|t$	$\|$ é associativo
$r(st) = (rs)t$	a concatenação é associativa
$r(s\|t) = rs\|rt$ $(r\|s)t = rt\|st$	a concatenação é distributiva em relação a $\|$
$\epsilon r = r$ $r\epsilon = r$	ϵ é o elemento neutro da concatenação
$r^* = (r\|\epsilon)^*$	relação entre ϵ e $*$
$r^{**} = r^*$	$*$ é idempotente

Definições regulares

Definição

Seja Σ um alfabeto. Uma definição regular sobre Σ é uma sequência de definições da forma

$$
\begin{aligned}
d_1 &\to r_1 \\
d_2 &\to r_2 \\
&\dots \\
d_n &\to r_n
\end{aligned}
$$

onde cada d_i é um nome distinto e r_i uma expressão regular sobre o alfabeto $\Sigma \cup \{\ d_1, d_2, \dots, d_{i-1}\ \}$.

Exemplo de definição regular

Os identificadores de Pascal, e em muitas outras linguagens, são formados por cadeias de caracteres e dígitos, começando com uma letra.

Exemplo de definição regular

Os identificadores de Pascal, e em muitas outras linguagens, são formados por cadeias de caracteres e dígitos, começando com uma letra.

Abaixo segue a definição regular para o conjunto de todos os identificadores válidos em Pascal:

Exemplo de definição regular

Os identificadores de Pascal, e em muitas outras linguagens, são formados por cadeias de caracteres e dígitos, começando com uma letra.

Abaixo segue a definição regular para o conjunto de todos os identificadores válidos em Pascal:

$$
\begin{aligned}
\textbf{letra} &\rightarrow \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \\
\textbf{digito} &\rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \mid \texttt{9} \\
\textbf{id} &\rightarrow \textbf{letra} \mid (\textbf{letra} \mid \textbf{digito})^*
\end{aligned}
$$

Simplificações notacionais

As seguintes notações podem simplificar as expressões regulares:

Simplificações notacionais

As seguintes notações podem simplificar as expressões regulares:

1. *Uma ou mais ocorrências*. Se r é uma expressão regular, então $(r)^+$ denota $(L(r))^+$. O operador $+$ tem a mesma associatividade e precedência do operator $*$. Vale que $r^* = r^+|\epsilon$ e que $r^+ = rr^*$.

Simplificações notacionais

As seguintes notações podem simplificar as expressões regulares:

1. *Uma ou mais ocorrências*. Se r é uma expressão regular, então $(r)^+$ denota $(L(r))^+$. O operador $+$ tem a mesma associatividade e precedência do operator $*$. Vale que $r^* = r^+|\epsilon$ e que $r^+ = rr^*$.
2. *Zero ou mais ocorrências*. Se r é uma expressão regular, então $r?$ denota $L(r) \cup \epsilon$. O operador $?$ é posfixo e unário, e $r? = r|\epsilon$.

Simplificações notacionais

As seguintes notações podem simplificar as expressões regulares:

1. *Uma ou mais ocorrências*. Se r é uma expressão regular, então $(r)^+$ denota $(L(r))^+$. O operador $+$ tem a mesma associatividade e precedência do operator $*$. Vale que $r^* = r^+|\epsilon$ e que $r^+ = rr^*$.
2. *Zero ou mais ocorrências*. Se r é uma expressão regular, então $r?$ denota $L(r) \cup \epsilon$. O operador $?$ é posfixo e unário, e $r? = r|\epsilon$.
3. *Classes de caracteres*. A notação `[`abc`]`, onde a, b, c são símbolos do alfabeto, denota a expressão regular $a \mid b \mid c$. A notação `[`a`-`z`]` abrevia a expressão regular $a \mid b \mid \ldots \mid z$.

Limitações das expressões regulares

- Existem linguagens que não podem ser descritas por meio de expressões regulares

Limitações das expressões regulares

- Existem linguagens que não podem ser descritas por meio de expressões regulares
- Por exemplo, não é possível descrever o conjunto $\mathcal{P}$ de todas as cadeias de parêntesis balanceados por meio de expressões regulares

Limitações das expressões regulares

- Existem linguagens que não podem ser descritas por meio de expressões regulares
- Por exemplo, não é possível descrever o conjunto $\mathcal{P}$ de todas as cadeias de parêntesis balanceados por meio de expressões regulares
- Contudo, o conjunto $\mathcal{P}$ pode ser descrito por meio de uma gramática livre de contexto

Limitações das expressões regulares

- Existem linguagens que não podem ser descritas por meio de expressões regulares
- Por exemplo, não é possível descrever o conjunto $\mathcal{P}$ de todas as cadeias de parêntesis balanceados por meio de expressões regulares
- Contudo, o conjunto $\mathcal{P}$ pode ser descrito por meio de uma gramática livre de contexto
- Existem linguagens que não podem ser descritas nem mesmo por meio de uma gramática livre de contexto

Limitações das expressões regulares

- Existem linguagens que não podem ser descritas por meio de expressões regulares
- Por exemplo, não é possível descrever o conjunto $\mathcal{P}$ de todas as cadeias de parêntesis balanceados por meio de expressões regulares
- Contudo, o conjunto $\mathcal{P}$ pode ser descrito por meio de uma gramática livre de contexto
- Existem linguagens que não podem ser descritas nem mesmo por meio de uma gramática livre de contexto
- Por exemplo, o conjunto

$$\mathcal{C} = \{wcw \mid w \text{ é uma cadeia de } a\text{'s e } b\text{'s}\}$$

não pode ser descrito nem por expressões regulares e nem por meio de uma gramática livre de contexto

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

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

Tratamento de espaços em branco

- Assuma que os lexemas sejam separados por espaços em brancos

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

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

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:

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

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 $\mathbf{ws}$ 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 $\mathbf{ws}$, 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*

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

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

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`

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

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)

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

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

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 $\star$

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

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

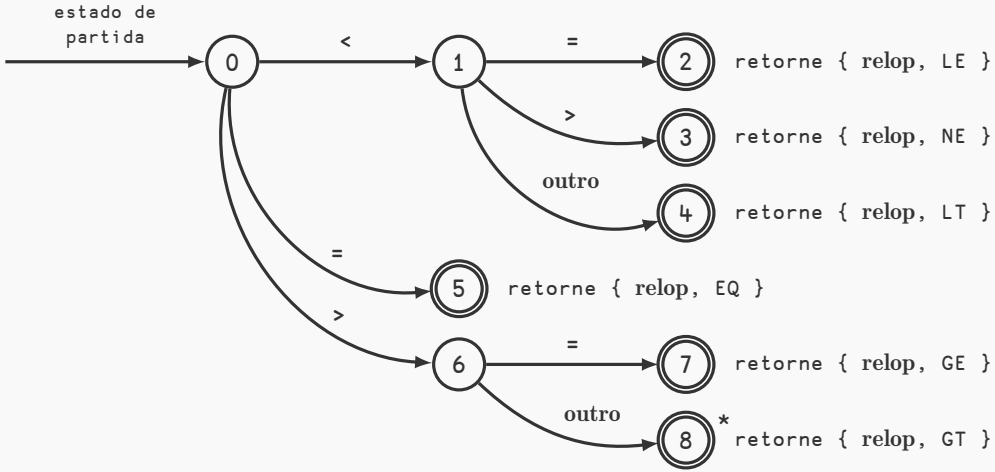
Retrações e erros léxicos

- 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

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

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

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

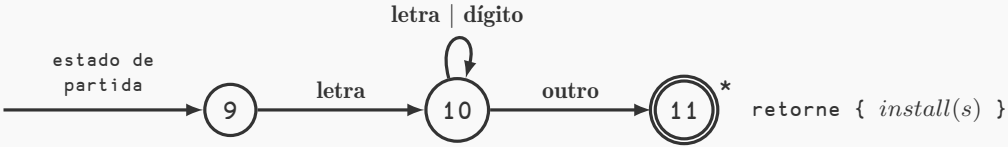
Identificadores e palavras-chave

- 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

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

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`

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

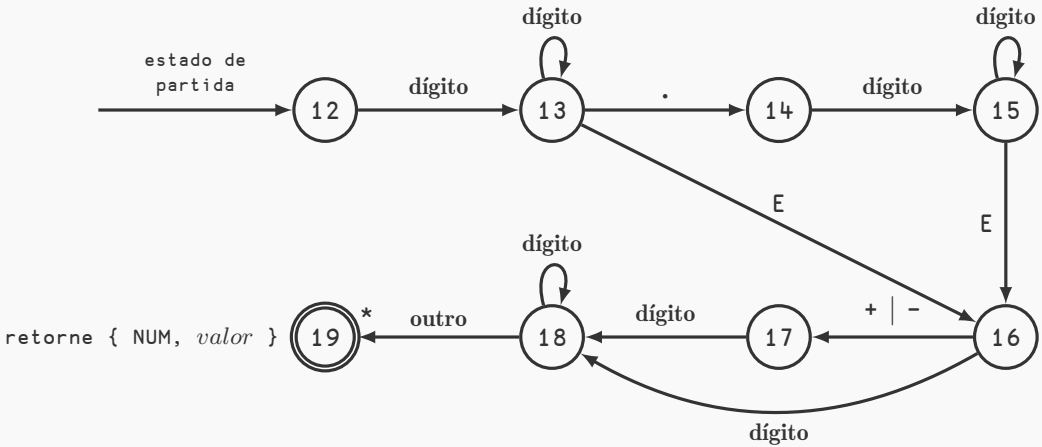
Identificação de constantes numéricas

- 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

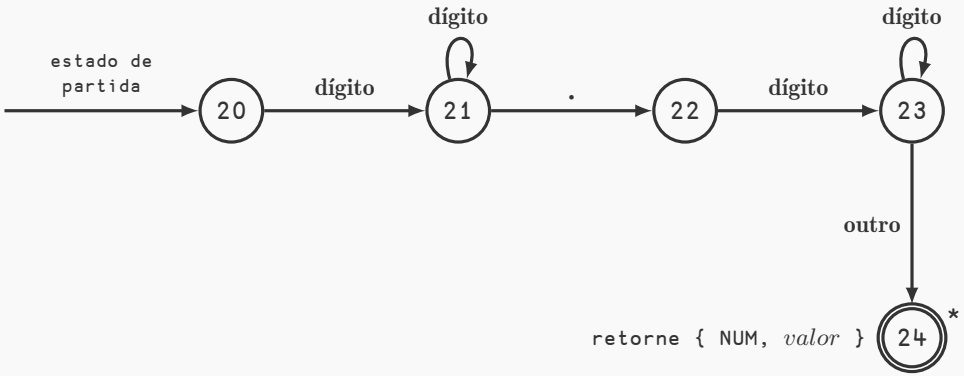
Identificação de constantes numéricas

- 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

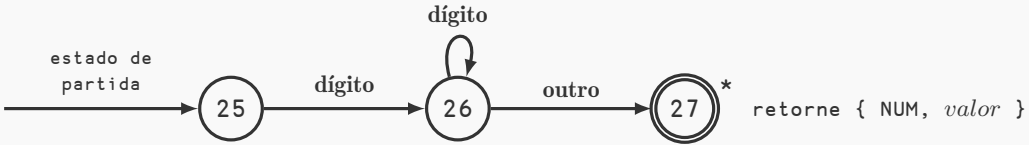
Diagramas de transição para constantes numéricas



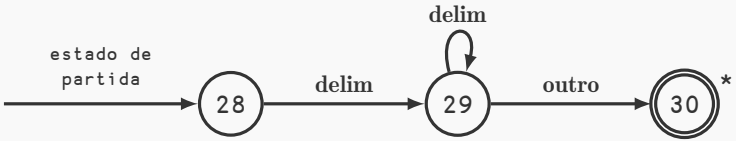
Diagramas de transição para constantes numéricas



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, isalpha }, { 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 }
```

Flex

- Flex (*Fast Lexical Analyzer Generator*) é um programa para a geração de analisadores léxicos

Flex

- Flex (*Fast Lexical Analyzer Generator*) é um programa para a geração de analisadores léxicos
- Ele foi escrito em linguagem C por Vern Paxson por volta de 1987

Flex

- Flex (*Fast Lexical Analyzer Generator*) é um programa para a geração de analisadores léxicos
- Ele foi escrito em linguagem C por Vern Paxson por volta de 1987
- Ele pode ser usado em conjunto com um gerador de analisadores sintáticos (por exemplo, o Yacc e o GNU Bison

Flex

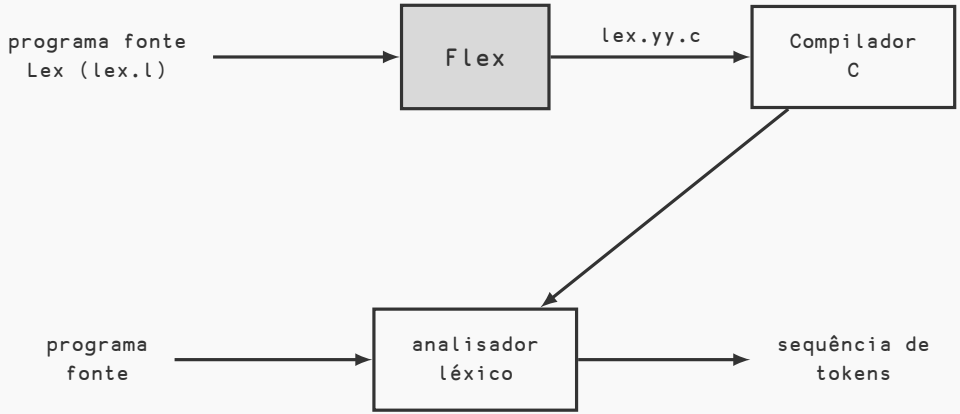
- Flex (*Fast Lexical Analyzer Generator*) é um programa para a geração de analisadores léxicos
- Ele foi escrito em linguagem C por Vern Paxson por volta de 1987
- Ele pode ser usado em conjunto com um gerador de analisadores sintáticos (por exemplo, o Yacc e o GNU Bison
- Flex é mais flexível e gera códigos mais rápidos que o Lex, outro programa gerador de analisadores léxicos

Flex

- Flex (*Fast Lexical Analyzer Generator*) é um programa para a geração de analisadores léxicos
- Ele foi escrito em linguagem C por Vern Paxson por volta de 1987
- Ele pode ser usado em conjunto com um gerador de analisadores sintáticos (por exemplo, o Yacc e o GNU Bison
- Flex é mais flexível e gera códigos mais rápidos que o Lex, outro programa gerador de analisadores léxicos
- Ele pode ser instalado, em distribuições Linux baseadas no Debian, por meio do comando

```
$ sudo apt-get install flex
```

Fluxo de uso do Flex para geração de analisadores léxicos



Programas Lex

- Programas Lex são salvos em arquivos com extensão `.l` (ou `.lex`)

Programas Lex

- Programas Lex são salvos em arquivos com extensão `.l` (ou `.lex`)
- Este programas exportam uma função chamada `yylex()` que, ao ser chamada, extraí o próximo token do programa fonte

Programas Lex

- Programas Lex são salvos em arquivos com extensão `.l` (ou `.lex`)
- Este programas exportam uma função chamada `yylex()` que, ao ser chamada, extraí o próximo token do programa fonte
- O código gerado (arquivo `lex.yy.c`) pode ser usado para gerar um executável independente, ou pode ser compilado como código objeto e ser integrado ao analisador sintático

Programas Lex

- Programas Lex são salvos em arquivos com extensão `.l` (ou `.lex`)
- Este programas exportam uma função chamada `yylex()` que, ao ser chamada, extraí o próximo token do programa fonte
- O código gerado (arquivo `lex.yy.c`) pode ser usado para gerar um executável independente, ou pode ser compilado como código objeto e ser integrado ao analisador sintático
- Os programas Lex são divididos em três partes: a seção de definições, a seção de regras e a seção de códigos de usuário

Programas Lex

- Programas Lex são salvos em arquivos com extensão `.l` (ou `.lex`)
- Este programas exportam uma função chamada `yylex()` que, ao ser chamada, extraí o próximo token do programa fonte
- O código gerado (arquivo `lex.yy.c`) pode ser usado para gerar um executável independente, ou pode ser compilado como código objeto e ser integrado ao analisador sintático
- Os programas Lex são divididos em três partes: a seção de definições, a seção de regras e a seção de códigos de usuário
- A vantagem do uso de programas Lex é que eles permitem a especificação dos tokens por meio de expressões regulares, e a implementação dos diagramas de transição é feita automaticamente pelo Flex

Seção de definições

- Nesta seção são declaradas variáveis, constantes e definições regulares

Seção de definições

- Nesta seção são declaradas variáveis, constantes e definições regulares
- As declarações desta seção deve ser delimitado pelas sequências de caracteres `"%{"` e `"%}"`

Seção de definições

- Nesta seção são declaradas variáveis, constantes e definições regulares
- As declarações desta seção deve ser delimitado pelas sequências de caracteres `"%{"` e `"%}"`
- O conteúdo desta seção é copiado diretamente para o arquivo `lex.yy.c`

Seção de definições

- Nesta seção são declaradas variáveis, constantes e definições regulares
- As declarações desta seção deve ser delimitado pelas sequências de caracteres `"%{"` e `"%}"`
- O conteúdo desta seção é copiado diretamente para o arquivo `lex.yy.c`
- As definições regulares devem ser declaradas após esta seção, na forma

```
nome regex
```

Seção de definições

- Nesta seção são declaradas variáveis, constantes e definições regulares
- As declarações desta seção deve ser delimitado pelas sequências de caracteres `"%{"` e `"%}"`
- O conteúdo desta seção é copiado diretamente para o arquivo `lex.yy.c`
- As definições regulares devem ser declaradas após esta seção, na forma

```
nome regex
```

- Uma vez definido um nome, ele pode ser usado nas definições regulares subsequentes, desde que sejam delimitados por chaves

Exemplo de seção de declarações

```
%{
    enum {
        LT, LE, EQ, NE, GT, GE,
        IF, THEN, ELSE, ID, NUM, RELOP, END_OF_FILE
    };

    int yylval;
    int instalar_id();
    int instalar_num();
%}
```

```
delim   [ \t\n]
ws      {delim}+
letra   [A-Za-z]
digito  [0-9]
id      {letra}({letra}|{digito})*
num     {digito}+(\.{digito}+)?(E[+-]?(digito)+)?
```

Seção de regras

- Esta seção contém uma série de regras, uma por linha, na forma

```
padrão ação
```

Seção de regras

- Esta seção contém uma série de regras, uma por linha, na forma

```
padrão ação
```

- O padrão não deve estar indentado e deve estar na mesma linha da ação

Seção de regras

- Esta seção contém uma série de regras, uma por linha, na forma

```
padrão ação
```

- O padrão não deve estar indentado e deve estar na mesma linha da ação
- O padrão pode conter algum nome presente nas declarações regulares

Seção de regras

- Esta seção contém uma série de regras, uma por linha, na forma

```
padrão ação
```

- O padrão não deve estar indentado e deve estar na mesma linha da ação
- O padrão pode conter algum nome presente nas declarações regulares
- Neste caso, o nome deve ser delimitado por chaves

Seção de regras

- Esta seção contém uma série de regras, uma por linha, na forma

```
padrão ação
```

- O padrão não deve estar indentado e deve estar na mesma linha da ação
- O padrão pode conter algum nome presente nas declarações regulares
- Neste caso, o nome deve ser delimitado por chaves
- Esta seção é limitada pela sequência de caracteres **%%**

Exemplo de seção de declarações

```
19 %%
20
21 {ws}     { printf("WS\n"); /* Nenhuma ação e nenhum valor retornado */ }
22 if       { return IF; }
23 then     { return THEN; }
24 else     { return ELSE; }
25 {id}     { yylval = instalar_id(); return ID; }
26 {num}    { yylval = instalar_num(); return NUM; }
27 "<"      { yylval = LT; return RELOP; }
28 "<="     { yylval = LE; return RELOP; }
29 "="      { yylval = EQ; return RELOP; }
30 "<>"     { yylval = NE; return RELOP; }
31 ">"      { yylval = GT; return RELOP; }
32 ">="     { yylval = GE; return RELOP; }
33
34 %%
```

Seção de código de usuário

- Esta seção também é copiada diretamente para o arquivo `lex.yy.c`

Seção de código de usuário

- Esta seção também é copiada diretamente para o arquivo `lex.yy.c`
- Uma outra alternativa é definir estes códigos em arquivos separados e depois carregar este código na compilação do analisador léxico

Seção de código de usuário

- Esta seção também é copiada diretamente para o arquivo `lex.yy.c`
- Uma outra alternativa é definir estes códigos em arquivos separados e depois carregar este código na compilação do analisador léxico

```
36 int instalar_id()
37 {
38     // Insere o lexema e o token na tabela de símbolos e retorna o índice da tabela
39     // onde o símbolo foi inserido. O lexema fica armazenado na variável yytext
40     return -1;
41 }
42
43 int instalar_num()
44 {
45     // Insere o valor do lexema na tabela de números e retorna o índice da tabela
46     // onde o número foi inserido. O lexema fica armazenado na variável yytext
47     return -2;
48 }
```

Exemplo de função `main()` para um analisador léxico independente

```
50 int yywrap() { return 1; }
51
52 int main()
53 {
54     while (1)
55     {
56         int token = yylex();
57
58         if (token == END_OF_FILE)
59         {
60             printf("Fim da entrada\n");
61             return 0;
62         }
63
64         printf("Token = %d, yytext = %s, yylval = %d\n", token, yytext, yylval);
65     }
66
67     return 0;
68 }
```

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.