Compiladores

CIÊNCIA DA COMPUTAÇÃO

Prof. Dr. João Paulo Aramuni

UNIVERSIDADE FUMEC

Sumário

* **Análise Léxica**
  * Conceitos Básicos em Gramáticas e Linguagens
  * Analisador Léxico
  * Descrição de Símbolos com Gramáticas Regulares
  * Descrição de Símbolos com Autômatos Finitos

Conceitos Básicos em Gramáticas e Linguagens

* Veremos em detalhes como construir um analisador léxico.
* Para implementar um analisador léxico à mão, é importante começar com um diagrama ou outra descrição para os lexemas de cada token.
* Podemos, então, escrever código para identificar a ocorrência de cada lexema na entrada e retornar informações sobre o token identificado.

Conceitos Básicos em Gramáticas e Linguagens

* Também podemos produzir um analisador léxico automaticamente especificando os padrões dos lexemas para um ***gerador de analisador léxico*** e compilando esses padrões em código que funciona como um analisador léxico.

* Essa técnica facilita a modificação de um analisador léxico, pois só temos de reescrever os padrões afetados, e não o programa todo.

Conceitos Básicos em Gramáticas e Linguagens

* Esta abordagem também acelera o processo de implementação do analisador léxico, pois o programador especifica os padrões do software em bem alto nível e conta com o gerador para produzir o código detalhado.

* Começamos o estudo dos geradores de analisador léxico apresentando as ***expressões regulares***, uma notação conveniente para especificar os padrões dos lexemas.

Conceitos Básicos em Gramáticas e Linguagens

* Mostramos como essa notação pode ser transformada, primeiro em ***autômatos não determinísticos*** e depois em ***autômatos determinísticos***.

* Essas duas notações podem ser usadas como entrada para um "driver", ou seja, um código que simula esses autômatos e os utiliza como guia para determinar o próximo token. Esse driver e a especificação na forma de autômato formam o núcleo do analisador léxico.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * Como a **primeira fase** de um compilador, a tarefa principal do analisador léxico é ler os caracteres da entrada do programa fonte, agrupá-los em unidades lexicamente significativas, chamadas de lexemas, e produzir como saída uma sequência de tokens para cada lexema no programa fonte.

  * O fluxo de tokens é enviado ao analisador sintático para que a análise seja efetuada. É comum que o analisador léxico interaja com a tabela de símbolos também.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * Como vimos, um token consiste em dois componentes, um nome de token e um valor de atributo. Os nomes de tokens são símbolos abstratos usados pelo analisador para fazer o reconhecimento sintático.

  * Frequentemente, chamamos esses nomes de token ***terminais***, uma vez que eles aparecem como ***símbolos terminais*** na **gramática** para uma linguagem de programação.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * O valor do atributo, se houver, é um apontador para a tabela de símbolos que contém as informações adicionais sobre o token.

  * Essas informações adicionais não fazem parte da gramática, de modo que, em nossa discussão sobre **análise sintática**, normalmente nos referimos aos tokens e aos terminais como sinônimos.

Conceitos Básicos em Gramáticas e Linguagens

* **O Papel do Analisador Léxico**
  * Em um compilador, o analisador léxico lê os caracteres do programa fonte, agrupa-os em unidades lexicamente significativas, chamadas lexemas, e produz como saída tokens representando esses lexemas.

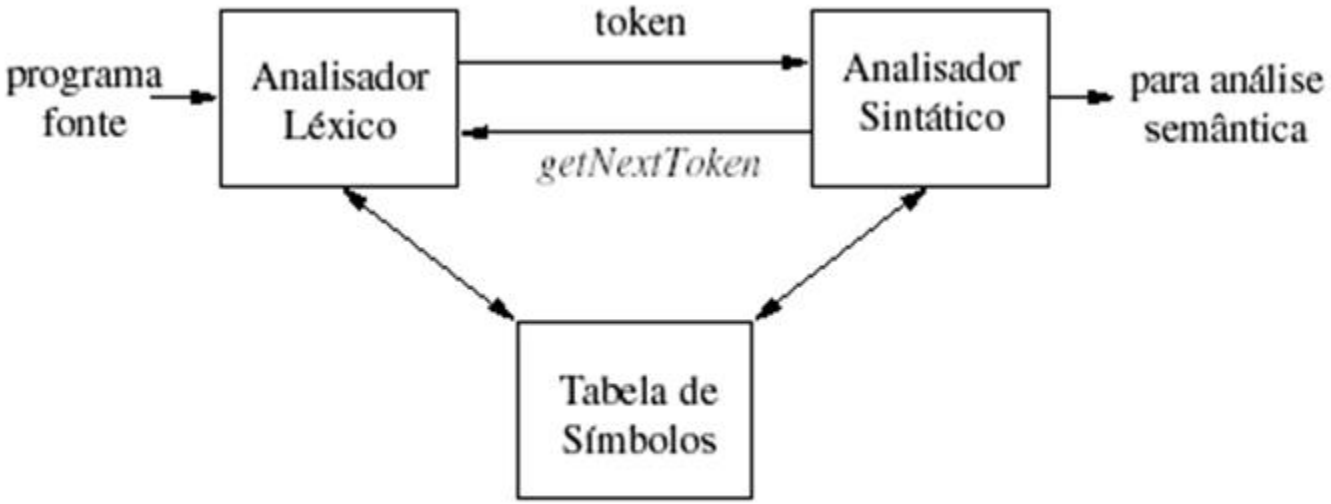
  * Como vimos, um token consiste em dois componentes, um nome de token e um valor de atributo. Os nomes de tokens são símbolos abstratos usados pelo analisador para fazer o reconhecimento sintático.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * Por exemplo, quando o analisador léxico descobre que um lexema é um identificador, precisa inserir esse lexema na tabela de símbolos.

  * Em alguns casos, as informações referentes ao identificador devem ser lidas da tabela de símbolos pelo analisador léxico para ajudá-lo a determinar o token apropriado que ele precisa passar ao analisador sintático.

* **O Papel do Analisador Léxico**
  * Essas interações são mostradas na Fig. 1, abaixo.
  * Normalmente, a interação é implementada fazendo-se com que o analisador sintático chama o analisador léxico.



Interações entre o analisador léxico e o analisador sintático.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * A chamada, sugerida pelo comando ***getNextToken***, faz com que o analisador léxico leia caracteres de sua entrada até que ele possa identificar o próximo lexema e produza para ele o próximo token, que retorna ao analisador sintático.

  * Como o analisador léxico é a parte do compilador que lê o texto fonte, ele pode realizar outras tarefas além da identificação de lexemas. Uma dessas tarefas é remover os comentários e o espaço em branco (espaço, quebra de linha, tabulação e talvez outros caracteres que são usados para separar os tokens na entrada).

Analisador Léxico

* **O Papel do Analisador Léxico**
  * Outra tarefa é correlacionar as mensagens de **erro** geradas pelo compilador com o programa fonte.
  * Por exemplo, o analisador léxico pode registrar o número de caracteres de quebra de linha vistos, de modo que possa associar um número de linha a cada mensagem de erro. Em alguns compiladores, o analisador léxico faz uma cópia do programa fonte com as mensagens de erro inseridas nas posições apropriadas.
  * Se o programa fonte utilizar um pré-processador de *macro*, a expansão de macros também pode ser realizada pelo analisador léxico.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Comentários e Espaço em Branco**
    * Se o espaço em branco for eliminado pelo analisador léxico, o analisador sintático nunca precisará considerá-lo.
    * A alternativa de modificar a gramática para incorporar o espaço em branco à sintaxe não é tão fácil de implementar.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Ler adiante**
    * Um analisador léxico talvez precise ler alguns caracteres adiante antes de poder decidir sobre o token a ser retornado ao analisador sintático.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Ler adiante**
    * Por exemplo, um analisador léxico para C ou Java precisa ***ler adiante*** depois de ver o caractere >. Se o caractere seguinte for =, então > faz parte da sequência de caracteres >=, o lexema do token para o operador "maior ou igual a". Caso contrário, o próprio > forma o operador "maior que", e o analisador léxico terá lido um caractere a mais.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Ler adiante**
    * Uma técnica geral para ler adiante na entrada é manter um ***buffer*** de entrada, do qual o analisador léxico pode ler e colocar caracteres de volta.
    * Os buffers de entrada podem ser justificados apenas com base na eficiência, pois ler um bloco de caracteres normalmente é mais eficiente do que ler um caractere de cada vez.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Ler adiante**
    * Em relação aos buffers de entrada, um **apontador** registra a parte da entrada que está sendo analisada; para voltar um caractere, move-se o apontador para trás.
    * As técnicas para uso de buffers de entrada, serão discutidas adiante.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Ler adiante**
    * O analisador léxico lê adiante apenas quando necessário.
    * Um operador como * pode ser identificado sem leitura adiante.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Constantes**
    * Sempre que um único dígito aparece em uma gramática para expressões, é razoável permitir qualquer constante inteira em seu lugar.
    * Existem duas formas de definir constantes inteiras: criar um símbolo terminal, digamos ***num***, para tais constantes, ou incorporar a sintaxe das constantes inteiras na gramática.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Constantes**
    * A tarefa de coletar caracteres em inteiros e calcular seu valor numérico geralmente é atribuída a um analisador léxico, de modo que os números podem ser tratados como uma única unidade durante a análise sintática e a tradução.

    * Quando uma sequência de dígitos aparece no fluxo de entrada, o analisador léxico passa para o analisador sintático um token consistindo no terminal ***num*** junto com um atributo de valor inteiro calculado a partir dos dígitos lidos.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Constantes**
    * Se escrevermos os tokens como tuplas delimitadas por <>, a entrada 31+28+59 será transformada na sequência:

      <***num***, 31><+><***num***, 28><+><***num***, 59>

    * Neste exemplo, o símbolo terminal + não possui atributos, portanto sua tupla é simplesmente <+>.

    * Por último, um pedaço de código lê os dígitos em um inteiro e calcula o valor do inteiro utilizando alguma variável.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Reconhecendo Palavras-Chave e Identificadores**
    * A maioria das linguagens utiliza sequências de caracteres fixos como **`for`**, **`do`** e **`if`** como marcas de pontuação ou para identificar suas construções.
    * Essas sequências de caracteres são chamadas ***palavras-chave***.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Reconhecendo Palavras-Chave e Identificadores**
    * As sequências de caracteres também são usadas como identificadores para nomes de variáveis, arranjos, funções e outras construções desse tipo.

    * As gramáticas normalmente tratam os identificadores como terminais para simplificar o reconhecedor sintático, que pode então esperar o mesmo terminal, digamos **id**, toda vez que algum identificador aparecer na sua entrada.

Analisador Léxico

- **O Papel do Analisador Léxico**
  - **Reconhecendo Palavras-Chave e Identificadores**
    - Por exemplo, para a entrada:

      count = count + increment;

    - O analisador sintático trabalha com o fluxo de terminais **id = id + id**. O token **id** possui um atributo que contém o lexema. Escrevendo tokens como tuplas, vemos que as tuplas para o fluxo de entrada acima são:

<**id**, "count"> <=> <**id**, "count"> <+> <**id**, "increment"><;>

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Reconhecendo Palavras-Chave e Identificadores**
    * A palavras-chave geralmente satisfazem as regras para formar identificadores, portanto é necessário haver um mecanismo para decidir quando um lexema forma uma palavra-chave e quando ele forma um identificador.

    * O problema é mais fácil de resolver se as palavras-chave foram ***reservadas***; ou seja, se não puderem ser usadas como identificadores. Neste caso, uma sequência de caracteres forma um identificador apenas se não for uma palavra-chave.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * Às vezes, os analisadores léxicos são divididos em uma cascata de dois processos:

  * **a)** O ***escandimento*** consiste no processo simples de varredura de entrada sem se preocupar, por exemplo, com a remoção de comentários e a compactação de caracteres de espaço em branco consecutivos em apenas um caractere.
  * **b)** A ***análise léxica*** propriamente dita é a parte mais complexa, onde o analisador léxico produz a sequência de tokens como saída.

* **O Papel do Analisador Léxico**
  * **Análise Léxica Versus Análise Sintática**
    * Existem vários motivos pelos quais a parte de análise de um compilador normalmente é separada em fases de análise léxica e análise sintática:

    * **1) *Simplicidade de projeto* é a consideração mais importante**. A separação das análises léxica e sintática normalmente nos permite simplificar pelo menos uma dessas tarefas.

      * Por exemplo, um analisador sintático que tivesse de lidar com comentários e espaço em branco como unidades sintáticas seria muito mais complexo do que um que pudesse assumir que comentários e espaço em branco já foram removidos pelo analisador léxico. ***Se estivermos projetando uma nova linguagem, a separação dos aspectos léxico e sintático pode levar a um projeto de linguagem mais limpo.***

* **O Papel do Analisador Léxico**
  * **Análise Léxica Versus Análise Sintática**
    * Existem vários motivos pelos quais a parte de análise de um compilador normalmente é separada em fases de análise léxica e análise sintática:

    * **2)** ***A eficiência do compilador é melhorada.*** Um analisador léxico separado nos permite aplicar técnicas especializadas, que servem apenas à tarefa léxica, e não à tarefa de análise sintática. Além disso, as técnicas especializadas de ***buffering*** para a leitura dos caracteres da entrada podem acelerar o compilador significativamente.

* **O Papel do Analisador Léxico**
  * **Análise Léxica Versus Análise Sintática**
    * Existem vários motivos pelos quais a parte de análise de um compilador normalmente é separada em fases de análise léxica e análise sintática:

    * **3)** ***A portabilidade do compilador é melhorada.*** As peculiaridades específicas do dispositivo de entrada podem ser restringidas ao analisador léxico.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Tokens, Padrões e Lexemas**
    * Ao discutir a análise léxica, usamos três termos relacionados, porém distintos:

      * Um ***token*** é um par consistindo em um nome e um valor de atributo opcional. O nome do token é um símbolo abstrato que representa um tipo de unidade léxica, por exemplo, uma palavra-chave em particular, ou uma sequência de caracteres de entrada denotando um identificador. Os nomes de tokens são os símbolos da entrada que o analisador sintático processa. Geralmente escrevemos o nome de um token em negrito. Vamos frequentemente nos referir a um token por seu nome de token.

* **O Papel do Analisador Léxico**
  * **Tokens, Padrões e Lexemas**
    * Ao discutir a análise léxica, usamos três termos relacionados, porém distintos:

      * Um ***padrão*** é uma descrição da forma que os lexemas de um token podem assumir. No caso de uma palavra-chave como um token, o padrão é uma estrutura mais complexa, que é *casada* por muitas sequências de caracteres.

      * Um ***lexema*** é uma sequência de caracteres no programa fonte que casa com o padrão para um token e é identificado pelo analisador léxico como uma instância desse token.

* **O Papel do Analisador Léxico**
  * **Tokens, Padrões e Lexemas**
    * A Fig. 2, abaixo, mostra alguns tokens típicos, seus padrões descritos informalmente, e alguns exemplos de lexemas.

TOKEN	DESCRIÇÃO INFORMAL	EXEMPLOS DE LEXEMAS
if	caracteres `i`, `f`	`if`
else	caracteres `e`, `l`, `s`, `e`	`else`
comparison	`<` `or` `>` ou `<=` ou `>=` ou `==` ou `!=`	`<=`, `!=`
id	letra seguida por letras e dígitos	`pi`, `score`, `D2`
number	qualquer constante numérica	`3.14159`, `0`, `6.02e23`
literal	qualquer caractere diferente de `"`, cercado por `"`s	`"core dumped"`

Exemplos de tokens.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Tokens, Padrões e Lexemas**
    * Os tokens são símbolos léxicos reconhecidos através de um padrão.
    * Os tokens podem ser divididos em dois grupos:
      * Tokens simples: são tokens que não têm valor associado pois a classe do token já a descreve. Exemplo: palavras reservadas, operadores, delimitadores: <if,>, <else>, <+,>.
      * Tokens com argumento: são tokens que têm valor associado e corresponde a elementos da linguagem definidos pelo programador. Exemplo: identificadores, constantes numéricas - <id, 3>, <numero, 10>, <literal, Olá Mundo> .
    * A tabela a seguir mostra outros exemplos de uso dos termos 'token', 'padrão' e 'lexema' durante a análise léxica:

Token	Padrão	Lexema	Descrição
<const, >	Sequência das palavras c, o, n, s, t	const	Palavra reservada
<while, >	Sequência das palavras w, h, i, l, e	while, While, WHILE	Palavra reservada
<if, >	Sequência das palavras i, f	If, IF, iF, If	Palavra reservada
<=, >	<, >, <=, >=, ==, !=	==, !=	
<numero, 18>	Dígitos numéricos	0.6, 18, 0.009	Constante numérica
<literal, "Olá Mundo">	Caracteres entre ""	“Olá Mundo”	Constante literal
<identificador, 1>	Nomes de variáveis, funções, parâmetros de funções.	nomeCliente, descricaoProduto, calcularPreco()	Nome de variável, nome de função
<=, >	=	=	Comando de atribuição
<{, >	{, }, [,]	{, }, [,]	Delimitadores de início e fim

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Tokens, Padrões e Lexemas**
    * Para ver como esses conceitos são usados na prática, no comando em C:

```
printf("Total = %d\n", score);
```

    * Tanto **`printf`** quanto **`score`** são lexemas casando com o padrão para o token **id**, e **`"Total = %d\n"`** é um lexema casando com **literal**.

* **O Papel do Analisador Léxico**
  * **Tokens, Padrões e Lexemas**
    * Em muitas linguagens de programação, as classes a seguir abrangem a maioria ou todos os tokens:

1) Um token para cada palavra-chave. O padrão para uma palavra-chave é o mesmo que a própria palavra-chave.

2) Tokens para os operadores, seja individualmente ou em classes, como o token **comparison** mencionado na Fig. 2.

3) Um token representado todos os identificadores.

4) Um ou mais tokens representando constantes, como números e cadeias literais.

5) Tokens para cada símbolo de pontuação, como parênteses esquerdo e direito, vírgula e ponto-e-vírgula.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Atributos de Tokens**
    * Quando mais de um lexema casar com um padrão, o analisador léxico precisa oferecer às fases subsequentes do compilador informações **adicionais** sobre qual foi o lexema em particular casado.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Atributos de Tokens**
    * Assim, em muitos casos, o analisador léxico retorna ao analisador sintático não apenas um nome de token, mas um valor de atributo que **descreve** o lexema representado pelo token.

    * O nome do token influencia as decisões durante a análise sintática, enquanto o valor do atributo influencia a tradução dos tokens após o reconhecimento sintático.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Atributos de Tokens**
    * Vamos considerar que os tokens possuem no máximo um atributo associado, embora esse atributo possa ter uma estrutura que combine com várias peças da informação.
    * O exemplo mais importante é o **id**, onde precisamos associar muitas informações ao token.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Atributos de Tokens**
    * Normalmente, **as informações sobre um identificador** – por exemplo, seu lexema, seu tipo, e sua localização na entrada (caso seja preciso emitir uma mensagem de erro sobre esse identificador) – **são mantidas na tabela de símbolos.**
    * Assim, o valor de atributo apropriado para um identificador é um apontador para a sua **entrada na tabela de símbolos.**

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Erros Léxicos**
    * É difícil para um analisador léxico saber, sem o auxílio de outros componentes, que existe um erro no código fonte.
    * Por exemplo, se a cadeia de caracteres **`fi`** for encontrada pela primeira vez em um programa C no contexto:

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

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Erros Léxicos**
    * Um analisador léxico não tem como saber se `fi` é a **palavra-chave** `if` escrita errada ou um identificador de **função não declarada.**
    * Como `fi` é um lexema válido para o token **id**, o analisador léxico precisa retornar o token **id** ao analisador sintático e deixar que alguma outra fase do compilador – provavelmente o analisador sintático, neste caso – trate do erro devido à transposição das letras.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Erros Léxicos**
    * Suponha, porém, que ocorra uma situação na qual o analisador léxico seja **incapaz de prosseguir** porque nenhum dos padrões para tokens casa com nenhum prefixo da entrada restante.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Erros Léxicos**
    * A estratégia de recuperação mais simples é a recuperação no "modo de pânico".
    * Removemos os caracteres seguintes da entrada restante, até que o analisador léxico possa encontrar um token bem formado no início da entrada que resta.
    * Essa técnica de recuperação pode **confundir o analisador sintático**, mas, em um ambiente de computação interativo, pode ser bastante adequada.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Erros Léxicos**
    * Outras ações de recuperação de erro possíveis são:
      1) Remover um caractere da entrada restante.
      2) Inserir um caractere que falta na entrada restante.
      3) Substituir um caractere por outro caractere.
      4) Transpor dois caracteres adjacentes.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Erros Léxicos**
    * Transformações como estas podem ser experimentadas em uma tentativa de reparar a entrada.
    * A estratégia mais simples desse tipo é ver se um prefixo da entrada restante pode ser transformado em um lexema válido por uma única transformação.
    * Essa estratégia faz sentido, pois, na prática, a maior parte dos erros léxicos envolve um único caractere.

Analisador Léxico

* **O Papel do Analisador Léxico**
  * **Erros Léxicos**
    * Uma estratégia de correção mais geral é **encontrar o menor número de transformações necessárias** para converter o programa fonte em um que consista apenas em lexemas válidos, mas, na prática, essa técnica é considerada muito dispendiosa para compensar o esforço.

Analisador Léxico

* **Bufferes de Entrada**
  * Antes de discutirmos o problema de reconhecer os lexemas da entrada, vamos examinar as maneiras pelas quais a simples porém importante tarefa de ler o programa fonte pode ser **acelerada.**

  * Essa tarefa se torna difícil pelo fato de com frequência precisarmos examinar um ou mais caracteres além do próximo lexema para certificar-nos de ter o lexema correto.

Analisador Léxico

* **Bufferes de Entrada**
  * Por exemplo, não podemos ter certeza de que chegamos ao fim de um identificador até que vejamos um caractere que não é uma letra ou um dígito, e, portanto não faz parte do lexema para **id**.

  * Em C, operadores com um único caractere como = ou < também podem ser o início de um operador de dois caracteres, como == ou <=.

Analisador Léxico

* **Bufferes de Entrada**
  * Devido à quantidade de tempo necessária para processar caracteres e o grande número de caracteres que precisam ser processados durante a compilação de um programa fonte grande, foram desenvolvidas técnicas especializadas de *buffering* para **reduzir o custo exigido no processamento de um único caractere de entrada.**

Analisador Léxico

* **Bufferes de Entrada**
  * Cada buffer possui o mesmo tamanho *N*, e *N* normalmente corresponde ao tamanho de um bloco de disco, por exemplo, 4096 bytes.
  * Usando um comando de leitura do sistema, podemos ler ***N* caracteres para um buffer**, em vez de fazer uma chamada do sistema para cada caractere.

Analisador Léxico

* **Especificação de Tokens**
  * As ***expressões regulares*** são uma importante notação para especificar os padrões de lexemas.
  * Embora não possam expressar todos os padrões possíveis, elas são muito eficientes na especificação dos tipos de padrões de que realmente precisamos para os tokens.

Analisador Léxico

* **Especificação de Tokens**
  * Veremos a notação formal para as expressões regulares e como essas expressões são usadas em um gerador de analisador léxico.
  * Depois, veremos como construir o analisador léxico convertendo expressões regulares em autômatos que realizam o reconhecimento dos tokens especificados.

Analisador Léxico

* **Especificação de Tokens**
  * **Cadeias e Linguagens**
    * Um *alfabeto* é qualquer conjunto finito de símbolos. Os exemplos típicos de símbolos são letras, dígitos e sinais de pontuação.

    * O conjunto {0,1} é o *alfabeto binário*. ASCII é um importante exemplo de alfabeto; ele é usado em muitos sistemas de software. Unicode, que inclui aproximadamente 100.000 caracteres de alfabetos do mundo inteiro, é outro importante exemplo.

Analisador Léxico

* **Especificação de Tokens**
  * **Cadeias e Linguagens**
    * Uma cadeia (*string*) em um alfabeto é uma sequência finita de símbolos retirada desse alfabeto.
    * Em teoria de linguagem, os termos "sentença" e "palavra" são frequentemente usados como sinônimos para "cadeia". O tamanho de uma cadeia s, normalmente escrito como $|s|$, é o número de ocorrências de símbolos em s. A cadeia vazia, indicada por λ, é uma cadeia de tamanho zero.

Analisador Léxico

* **Especificação de Tokens**
  * **Cadeias e Linguagens**
    * Como termos para partes da cadeia, podemos utilizar os mesmos conceitos de ***prefixo***, ***sufixo*** e ***subpalavra*** já vistos em FTC.

Analisador Léxico

* **Especificação de Tokens**
  * **Cadeias e Linguagens**
    * Uma linguagem é qualquer conjunto contável de cadeias de algum alfabeto fixo. Essa definição é muito ampla.
    * Linguagens abstratas como Ø, o *conjunto vazio*, ou {λ}, o conjunto contendo apenas a cadeia vazia, são linguagens sob essa definição.

Analisador Léxico

* **Especificação de Tokens**
  * **Cadeias e Linguagens**
    * Também o são o conjunto de todos os programas C sintaticamente bem formados e o conjunto de todas as sentenças inglesas gramaticamente corretas, embora essas duas últimas linguagens sejam difíceis de especificar exatamente.
    * Observe que a definição de "linguagem" não exige nenhum significado atribuído às cadeias na linguagem.

Analisador Léxico

* **Especificação de Tokens**
  * **Operações sobre Linguagens**
    * Na análise léxica, as operações mais importantes sobre as linguagens são união (L_1 U L_2), concatenação (L_1L_2) e fechamento (*Fecho de Kleene* de L, definido formalmente como L^*).

Analisador Léxico

* **Especificação de Tokens**
  * **Operações sobre Linguagens**
    * A união é a conhecida operação sobre conjuntos.
    * A concatenação de linguagens são todas as cadeias formadas a partir de uma cadeia da primeira linguagem e uma cadeia da segunda linguagem, em todas as formas possíveis, e concatenando-as.
    * O *fecho de Kleene* de uma linguagem L, indicado por L^*, é o conjunto de cadeias obtidas concatenando L zero ou mais vezes.

Analisador Léxico

* **Especificação de Tokens**
  * **Operações sobre Linguagens**
    * O fechamento positivo, indicado por L^+, é o mesmo que o fecho de Kleene, porém sem o λ.
    * Dica: Revise o material de FTC para relembrar o conteúdo de linguagens formais.

Analisador Léxico

* **Especificação de Tokens**
  * **Operações sobre Linguagens**
    * Exemplo: Considere que L seja o conjunto de letras {A, B, ..., Z, a, b, ..., z} e D seja o conjunto de dígitos {0, 1, ..., 9}.
    * Podemos pensar em L e D de duas maneiras essencialmente equivalentes. Uma delas é que L e D são, respectivamente, os alfabetos de letras maiúsculas e minúsculas e de dígitos.
    * A segunda maneira é que L e D são linguagens, ambas com todas as cadeias de tamanho um.

Analisador Léxico

* **Especificação de Tokens**
  * **Operações sobre Linguagens**
    * Aqui estão algumas outras linguagens que podem ser construídas a partir das linguagens L e D, usando os operadores vistos:

      * 1) $L \cup D$ é o conjunto de letras e dígitos – estritamente falando, a linguagem com 62 cadeias de tamanho um, cada uma tendo uma letra ou um dígito.
      * 2) LD é o conjunto de 520 cadeias de tamanho dois, cada uma consistindo em uma letra seguida por um dígito.
      * 3) L^4 é o conjunto de todas as cadeias de 4 letras.
      * 4) L^* é o conjunto de todas as cadeias de letras, incluindo λ.

* **Especificação de Tokens**
  * **Operações sobre Linguagens**
    * Aqui estão algumas outras linguagens que podem ser construídas a partir das linguagens L e D, usando os operadores vistos:

      * 5) $L(L \text{ U } D)^*$ é o conjunto de todas as cadeias de letras e dígitos começando com uma letra.
      * 6) D^+ é o conjunto de todas as cadeias de um ou mais dígitos.

      * Observe atentamente o item 5.

Analisador Léxico

* **Especificação de Tokens**
  * **Expressões Regulares**
    * Suponha que quiséssemos descrever o conjunto de identificadores válidos em C.
    * Essa é quase exatamente a linguagem descrita no item 5 visto no exemplo anterior, a única diferença é que o caractere sublinhado (*underscore*) está incluído entre as letras.

Analisador Léxico

* **Especificação de Tokens**
  * **Expressões Regulares**
    * No exemplo visto, pudemos descrever identificadores dando nomes a conjuntos de letras e dígitos, e usando os operadores da linguagem: união, concatenação e fechamento.

    * Esse processo é tão importante que uma notação chamada ***expressões regulares*** foi criada para descrever todas as linguagens que podem ser formadas a partir desses operadores aplicados aos símbolos de algum alfabeto.

Analisador Léxico

* **Especificação de Tokens**
  * **Expressões Regulares**
    * Nessa notação, se estabelecermos que ***letra_*** significa qualquer letra ou o sublinhado, e que ***dígito*** significa qualquer dígito, então poderíamos descrever os identificadores da linguagem C por:

letra_ **(*letra_* | *dígito*)***

Analisador Léxico

* **Especificação de Tokens**
  * **Expressões Regulares**
    * A barra vertical anterior significa união, os parênteses são usados para agrupar subexpressões, o asterisco significa "zero ou mais ocorrências de", e a justaposição de *letra* com o restante da expressão significa concatenação.

    * Dica: Revise o material de FTC para relembrar o conteúdo de expressões regulares.

Analisador Léxico

* **Especificação de Tokens**
  * **Expressões Regulares**
    * Uma linguagem que pode ser definida por uma expressão regular é chamada de *conjunto regular*.
    * Se duas expressões regulares denotam o mesmo conjunto regular, dizemos que elas são *equivalentes*

Analisador Léxico

* **Especificação de Tokens**
  * **Definições Regulares**
    * Por conveniência de notação, podemos dar nomes a certas expressões regulares e usar esses nomes em expressões subsequentes, como se os nomes fossem os próprios símbolos.

* **Especificação de Tokens**
  * **Definições Regulares**
    * Exemplo 1: Os identificadores da linguagem C são cadeias de letras, dígitos e sublinhados. Aqui está uma definição regular para os identificadores da linguagem C. Por convenção, vamos usar itálico para os símbolos definidos em definições regulares.

$$
\begin{aligned}
\textit{letter\_} &\rightarrow \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \mid \texttt{\_} \\
\textit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\
\textit{id} &\rightarrow \textit{lettter\_} \, (\, \textit{letter\_} \mid \textit{digit} \,)^{*}
\end{aligned}
$$

* **Especificação de Tokens**
  * **Definições Regulares**
    * Exemplo 2: Números sem sinal (inteiros ou ponto flutuante) são cadeias como 5280, 0.01234, 6.336E4 ou 1.89E-4.
    * Definição regular:

$$
\begin{aligned}
\textit{digit} &\rightarrow 0 \mid 1 \mid \ldots \mid 9 \\
\textit{digits} &\rightarrow \textit{digit}\ \textit{digit}^{*} \\
\textit{optionalFraction} &\rightarrow .\ \textit{digits} \mid \lambda \\
\textit{optionalExponent} &\rightarrow (\mathrm{E}\ (+ \mid - \mid \lambda)\ \textit{digits}) \mid \lambda \\
\textit{number} &\rightarrow \textit{digits}\ \textit{optionalFraction}\ \ \textit{optionalExponent}
\end{aligned}
$$

Analisador Léxico

* **Especificação de Tokens**
  * **Definições Regulares**
    * A definição regular vista anteriormente é uma especificação precisa para esse conjunto de cadeias.
    * Ou seja, uma *optionalFraction* é um ponto decimal seguido por um ou mais dígitos ou não existe (a cadeia vazia).
    * Um *optionalExponent*, se não estiver faltando, é a letra E seguida por um sinal de + ou - opcional, seguido por um ou mais dígitos.
    * Observe que pelo menos um dígito precisa vir após o ponto, de modo que um número não casa com 1., mas casa com 1.0.

Analisador Léxico

* **Especificação de Tokens**
  * **Extensões de Expressões Regulares**
    * Desde que *Kleene* introduziu as expressões regulares com os operadores básicos para união, concatenação e *fecho de Kleene*, na década de 1950, muitas extensões foram acrescentadas às expressões regulares para melhorar sua capacidade de especificar os padrões da cadeia.

* **Especificação de Tokens**
  * **Extensões de Expressões Regulares**
    * Veremos algumas extensões importantes que foram incorporadas inicialmente em utilitários Unix, como *Lex* (ou *Flex*, em uma versão mais recente), que são particularmente úteis na especificação de analisadores léxicos.

    * 1) **Uma ou mais instâncias**: O operador unário pós-fixado + representa o fechamento positivo de uma expressão regular e sua linguagem. Ou seja se r é uma expressão regular, então $(r)^+$ denota a linguagem $(L(r))^+$. O operador + tem a mesma precedência e associatividade do operador *. Duas leis algébricas úteis, $r^* = r^+ \mid \lambda$ e $r^+ = rr^* = r^*r$, se relacionam ao fecho e ao fecho positivo de *Kleene*, respectivamente.

* **Especificação de Tokens**
  * **Extensões de Expressões Regulares**
    * Veremos algumas extensões importantes que foram incorporadas inicialmente em utilitários Unix, como *Lex* (ou *Flex*, em uma versão mais recente), que são particularmente úteis na especificação de analisadores léxicos.

    * 2) **Zero ou uma instância:** O operador unário pós-fixado ? significa "zero ou uma ocorrência". Ou seja, $r?$ é equivalente a $r \mid \lambda$ ou, de outra maneira, $L(r?) = L(r) \cup \lambda$. O operador ? Tem a mesma precedência e associatividade de * e +.

Analisador Léxico

* **Especificação de Tokens**
  * **Extensões de Expressões Regulares**
    * Veremos algumas extensões importantes que foram incorporadas inicialmente em utilitários Unix, como *Lex* (ou *Flex*, em uma versão mais recente), que são particularmente úteis na especificação de analisadores léxicos.

    * 3) **Classes de caracteres**: Uma expressão regular $a_1|a_2|...|a_n$, onde os a's são cada um dos símbolos do alfabeto, pode ser substituída pela abreviação $[a_1a_2...a_n]$. Mais importante, quando a_1, a_2,...,a_n formam uma sequência lógica, por exemplo, letras maiúsculas consecutivas, letras minúsculas ou dígitos, podemos substituí-las por a_1-a_n, ou seja, apenas a primeira e a última separadas por um hífen. Assim, $[abc]$ é a abreviação de $a|b|c$, e $[a\text{-}z]$ é a abreviação de $a|b|...|z$.

Analisador Léxico

* **Especificação de Tokens**
  * **Extensões de Expressões Regulares**
    * Usando essas abreviações, podemos reescrever a definição regular do Exemplo 1 (Identificadores da linguagem C) como:

$$
\begin{aligned}
\textit{letter\_} &\rightarrow [\texttt{A-Za-z\_}] \\
\textit{digit} &\rightarrow [\texttt{0-9}] \\
\textit{id} &\rightarrow \textit{letter\_}\ (\ \textit{letter} \mid \textit{digit}\ )^*
\end{aligned}
$$

Analisador Léxico

* **Especificação de Tokens**
  * **Extensões de Expressões Regulares**
    * A definição regular do Exemplo 2 também pode ser simplificada para:

$$
\begin{aligned}
\mathit{digit} &\rightarrow [0\text{-}9] \\
\mathit{digits} &\rightarrow \mathit{digit}^{+} \\
\mathit{number} &\rightarrow \mathit{digits}\ (.\ \mathit{digits})?\ (\mathrm{E}\ [+-]?\ \mathit{digits}\ )?
\end{aligned}
$$

Descrição de Símbolos com Gramáticas Regulares

* **Reconhecimento de Tokens**
  * Até aqui, aprendemos a expressar padrões usando expressões regulares.
  * Agora, temos de estudar os padrões para todos os tokens necessários e construir um trecho de código que examina a cadeia de entrada e encontra um prefixo que seja um lexema casando com um dos padrões.

Descrição de Símbolos com Gramáticas Regulares

* **Reconhecimento de Tokens**
  * Para isso, vamos utilizar o seguinte exemplo:
  * O fragmento de gramática da Fig. 3 a seguir, descreve uma forma simples de comandos de desvio e expressões condicionais.
  * Esta sintaxe é semelhante à da linguagem Pascal, pois **`then`** aparece explicitamente após as condições. Para **`relop`**, usamos os operadores de comparação de linguagens como Pascal ou SQL, onde = é "igual" e <> é "não igual", pois apresenta uma estrutura interessante de lexemas.

Descrição de Símbolos com Gramáticas Regulares

* **Reconhecimento de Tokens**
  * Abaixo, Fig. 3

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	**λ**
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**number**

Uma gramática para comandos de desvio.

  * Dica: Revise o material de FTC para relembrar o conteúdo de gramáticas regulares.

Descrição de Símbolos com Gramáticas Regulares

* **Reconhecimento de Tokens**
  * Os terminais da gramática, que são **`if`**, **`then`**, **`else`**, **`relop`**, **`id`** e **`number`**, são os nomes dos tokens no que se refere ao analisador léxico. Os padrões para esses tokens são descritos por meio de definições regulares, como na Fig. 4 abaixo.

digit	→	[0-9]
digits	→	*digit*$^+$
number	→	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	→	[A-Za-z]
id	→	*letter* (*letter* | *digit*)*
if	→	if
then	→	then
else	→	else
relop	→	< | > | <= | >= | = | <>

Padrões para os tokens

Descrição de Símbolos com Gramáticas Regulares

* **Reconhecimento de Tokens**
  * Para esta linguagem, o analisador léxico reconhecerá as palavra-chave **`if`**, **`then`** e **`else`**, além dos lexemas que casam com os padrões para *relop*, *id* e *number*.
  * Para simplificar, vamos supor que as palavras-chave também são *palavras reservadas*: ou seja, não são identificadores, embora seus lexemas casem com o padrão para identificadores.
  * Além disso, atribuímos ao analisador léxico a tarefa de remover espaços em branco, reconhecendo o "token" *ws* definido por:

$$ws \rightarrow (\textbf{blank} \mid \textbf{tab} \mid \textbf{newline})^{+}$$

Descrição de Símbolos com Gramáticas Regulares

* **Reconhecimento de Tokens**
  * Aqui, **blank**, **tab** e **newline** são símbolos abstratos que usamos para expressar os caracteres ASCII com os mesmos nomes.
  * O token *ws* é diferente dos demais tokens porque, quando o reconhecemos, não o retornamos ao analisador sintático, mas reiniciamos a análise léxica a partir do caractere seguinte ao espaço em branco.
  * É o token seguinte que é retornado ao analisador sintático.

Descrição de Símbolos com Gramáticas Regulares

* **Reconhecimento de Tokens**
  * Nosso objetivo para o analisador léxico é resumido na Fig. 5 a seguir.
  * Essa tabela mostra para cada lexema ou família de lexemas, qual nome de token é retornado ao analisador sintático e qual valor de atributo é retornado.
  * Observe que, para os seis operadores relacionais, as constantes simbólicas LT, LE e assim por diante são usadas como valor de atributo, a fim de indicar qual instância do token **relop** encontramos. O operador em particular encontrado influenciará o código que é gerado pelo compilador.

Descrição de Símbolos com Gramáticas Regulares

* **Reconhecimento de Tokens**
  * Abaixo, Fig. 5

LEXEMAS	NOME DO TOKEN	VALOR DO ATRIBUTO
Qualquer *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Qualquer *id*	**id**	Apontador para entrada de tabela
Qualquer *number*	**number**	Apontador para entrada de tabela
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Tokens, seus padrões e valores de atributo.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Diagramas de Transição**
    * Como um passo intermediário na construção de um analisador léxico, primeiro convertermos padrões em fluxogramas estilizados, chamados "***diagramas de transição***".

    * Realizaremos manualmente a conversão dos padrões de expressão regular para diagramas de transição, mas, veremos mais adiante que existe um modo mecânico de construir esses diagramas a partir de coleções de expressões regulares.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Diagramas de Transição**
    * ***Diagramas de transição*** possuem uma coleção de nós ou círculos, denominados ***estados***.
    * Cada estado representa uma condição que poderia ocorrer durante o processo de escandimento da entrada procurando por um lexema que case um dos vários padrões.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Diagramas de Transição**
    * Podemos pensar em um estado como resumindo tudo aquilo de que precisamos saber sobre quais caracteres vimos entre os apontadores do buffer de entrada.

    * ***Arestas*** são direcionadas de um estado do diagrama de transição para outro. Cada aresta é *rotulada* por um símbolo ou conjunto de símbolos.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Diagramas de Transição**
    * Se estivermos em algum estado s, e o próximo símbolo de entrada for a, procuramos por uma aresta saindo do estado s rotulada por a (e talvez por outros símbolos também).
    * Se encontrarmos tal aresta, avançamos o apontador no buffer e entramos no estado do diagrama de transição ao qual essa aresta leva.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Diagramas de Transição**
    * Vamos considerar que todos os nossos diagramas de transição são *determinísticos*, significando que nunca existe mais de uma aresta saindo de determinado estado com determinado símbolo entre seus rótulos.
    * Os diagramas de transição *não determinísticos* tornam a vida muito mais fácil para o projetista de um analisador léxico, embora mais complicada para o implementador.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Diagramas de Transição**
    * Algumas convenções importantes sobre os diagramas de transição são:

    * 1) Certos estados são considerados estados de *aceitação*, ou *finais*. Esses estados indicam que um lexema foi encontrado, embora o lexema corrente possa não consistir em todas as posições entre os apontadores do buffer de entrada. Sempre indicamos um estado de aceitação pelo círculo duplo e, se houver uma ação a ser tomada – normalmente retornando um token e um valor de atributo ao analisador sintático – conectaremos essa ação ao estado de aceitação.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Diagramas de Transição**
    * Algumas convenções importantes sobre os diagramas de transição são:

    * 2) Além disso, se for necessário recuar o apontador uma posição no buffer (ou seja, o lexema não inclui o símbolo que nos levou ao estado de aceitação), então incluiremos um * ao lado desse estado de aceitação. No nosso caso, nunca é necessário recuar o apontador por mais de uma posição, mas, se fosse, poderíamos conectar qualquer quantidade de *s ao estado de aceitação.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Diagramas de Transição**
    * Algumas convenções importantes sobre os diagramas de transição são:

    * 3) Um estado é designado como o *estado inicial*; ele é iniciado por uma aresta, rotulada como "início", e não há arestas chegando ao estado inicial. O diagrama de transição sempre começa no estado inicial antes que qualquer símbolo de entrada tenha sido lido.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Diagramas de Transição**
    * A Fig. 6, a seguir, é um diagrama de transição que reconhece os lexemas casando com o token **relop**.
    * Começamos no estado 0, o estado inicial. Se encontrarmos < como o primeiro símbolo de entrada, então entre os lexemas que casam com o padrão para **relop**, só podemos estar procurando <, <>, ou <=.
    * Portanto, vamos para o estado 1 e examinamos o caractere seguinte. Se ele for =, então reconhecemos o lexema <=, entramos no estado 2 e retornamos o token **relop** com o atributo LE, a constante simbólica representando esse operador de comparação em particular.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Diagramas de Transição**
    * Se no estado 1, o caractere seguinte for >, então temos o lexema <>, e entramos no estado 3 para retornar uma indicação de que o operador não-igual foi encontrado.
    * Com qualquer outro caractere, o lexema é <, e entramos no estado 4 para retornar essa informação.
    * Observe, porém, que o estado 4 tem um * para indicar que devemos recuar na entrada uma posição.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Diagramas de Transição** - Abaixo, Fig. 6

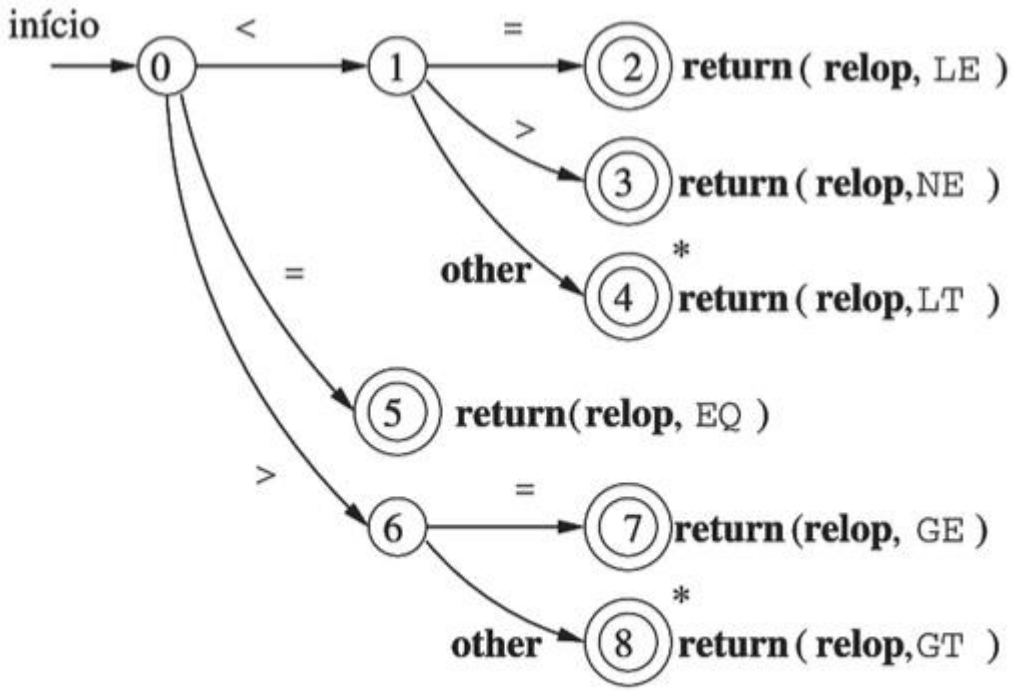


Diagrama de transição para **relop**.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Diagramas de Transição**
    * Por outro lado, se no estado 0 o primeiro caractere que vemos for =, então esse único caractere precisa ser o lexema. Imediatamente retornamos esse fato do estado 5.

    * A possibilidade restante é que o primeiro caractere seja >. Depois, temos de entrar no estado 6 e decidir, com base no próximo caractere, se o lexema é >= (se em seguida virmos o sinal =) ou apenas > (com qualquer outro caractere).

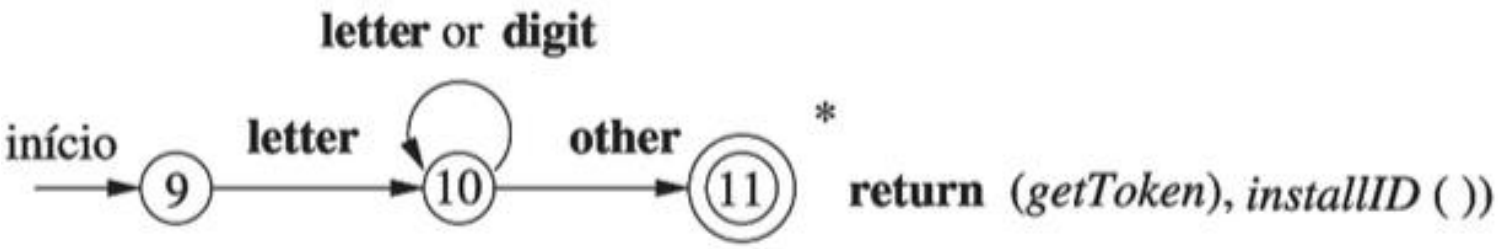
    * Observe que se, no estado 0, virmos qualquer caractere além de <, = ou >, possivelmente não podemos estar vendo um lexema relop, **de modo que esse diagrama de transição não será usado**.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Reconhecimento de Palavras Reservadas e Identificadores**
    * Reconhecer palavras-chave e identificadores implica um **problema**.
    * Usualmente, palavras-chave como **`if`** ou **`then`** são reservadas, de modo que as palavras-chave não são identificadores embora se *pareçam* com eles.
    * Portanto, muito embora usemos um diagrama de transição como o da Fig. 7 a seguir para procurar lexemas identificadores, esse diagrama também reconhecerá as palavras-chave **`if`**, **`then`** e **`else`** do exemplo em andamento.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Reconhecimento de Palavras Reservadas e Identificadores**
    * Abaixo, Fig. 7



Um diagrama de transição para id's e palavras-chave.

    * Podemos lidar com palavras reservadas que se parecem com identificadores de **duas formas:**

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Reconhecimento de Palavras Reservadas e Identificadores**
    * 1) **Instale inicialmente as palavras reservadas na tabela de símbolos.** Um campo de entrada da tabela de símbolos indica que essas cadeias de caracteres nunca são identificadores comuns, e diz qual token elas representam.

    * Consideramos que esse método esteja em uso na Fig. 7. Quando encontramos um identificador, uma chamada a ***installID*** o coloca na tabela de símbolos caso ele ainda não esteja lá, e retorna um apontador para a entrada da tabela de símbolos referente ao lexema encontrado.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Reconhecimento de Palavras Reservadas e Identificadores**
    * Naturalmente, qualquer identificador que não esteja na tabela de símbolos durante a análise léxica não pode ser uma palavra reservada, portanto seu token é **id**.

    * A função ***getToken*** examina a entrada da tabela de símbolos em busca do lexema encontrado e retorna qualquer que seja o nome de token que a tabela de símbolos diz que esse lexema representa – ou **id** ou um dos tokens de palavra-chave que foi inicialmente instalado na tabela.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Reconhecimento de Palavras Reservadas e Identificadores**
    * 2) **Crie diagramas de transição separados para cada palavra-chave**. Um exemplo para a palavra-chave **`then`** aparece na Fig. 8, a seguir.

    * Observe que esse diagrama de transição consiste nos estados representando a situação logo após cada letra sucessiva da palavra-chave ser vista, seguido por um teste para uma "*não-letra-ou-dígito*", ou seja, qualquer caractere que não possa ser a continuação de um identificador.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Reconhecimento de Palavras Reservadas e Identificadores**
    * É preciso verificar se o identificador terminou, ou, do contrário, retornaríamos o token **`then`** em situações nas quais o token correto é **id**, com um lexema como **`thenextvalue`** que possui **`then`** como prefixo próprio.

    * Se adotarmos essa técnica, temos de priorizar os tokens de modo que os tokens de palavra reservada sejam reconhecidos em preferência a **id**, quando o lexema casa com os dois padrões. Não vamos utilizar essa técnica no exemplo em andamento, por esse motivo os estados da Fig. 8 não são numerados.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Reconhecimento de Palavras Reservadas e Identificadores**
    * Abaixo, Fig. 8

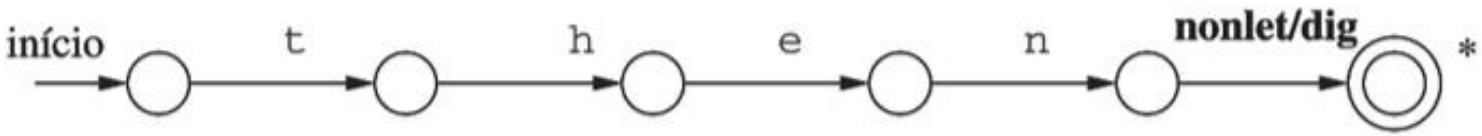


Diagrama de transição hipotético para a palavra-chave `then`.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Término do Exemplo em Andamento**
    * O diagrama de transição para os **id**'s que vimos na Fig. 7 possui uma estrutura simples.
    * A partir do estado 9, ele verifica se o lexema começa com uma letra e vai para o estado 10 nesse caso. Permanecemos no estado 10 enquanto a entrada tiver letras e dígitos.
    * A partir do momento que encontramos qualquer símbolo diferente de uma letra ou dígito, passamos para o estado 11 e aceitamos o lexema encontrado.

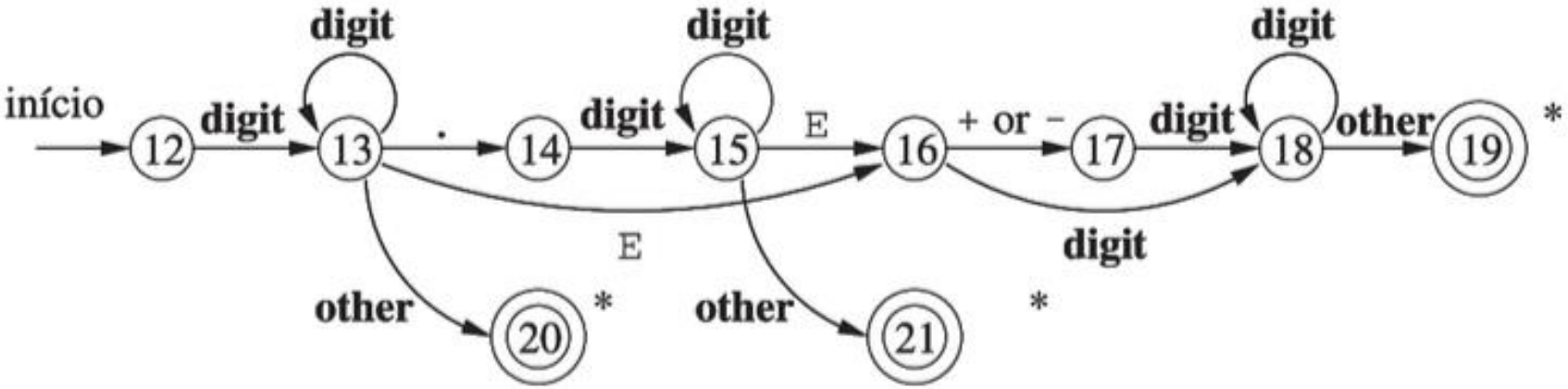
Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Término do Exemplo em Andamento**
    * Como o último caractere não faz parte do identificador, temos de recuar na entrada uma posição e, conforme foi visto, inserimos o que encontramos na tabela de símbolos e determinamos se temos uma palavra-chave ou um verdadeiro identificador.

    * O diagrama de transição para o token **number** aparece na Fig. 9 a seguir, e até aqui é o diagrama mais complexo que já vimos.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Término do Exemplo em Andamento**
    * Abaixo, Fig. 9



Um diagrama de transição para números sem sinal.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Término do Exemplo em Andamento**
    * A partir do estado 12, se encontrarmos um dígito, vamos para o estado 13. Neste estado, podemos ler qualquer quantidade de dígitos adicionais. Porém, se encontrarmos qualquer símbolo diferente de um dígito ou um ponto, então vemos um número na forma de um inteiro; 123 é um exemplo.

    * Esse caso é tratado pela entrada no estado 20, em que retornamos o token **number** e um apontador para uma tabela de constantes na qual o lexema encontrado é inserido. Essa parte não aparece no diagrama, mas é semelhante ao modo como tratamos os identificadores.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Término do Exemplo em Andamento**
    * Se, em vez disso, virmos um ponto no estado 13, então temos uma "*optionalFraction*". Entramos no estado 14 e procuramos um ou mais dígitos adicionais; o estado 15 é usado para essa finalidade.
    * Se encontrarmos um E, então temos um "*optionalExponent*", cujo reconhecimento é tarefa dos estados 16 até 19.
    * Se no estado 15, em vez disso encontrarmos algo diferente de E ou um dígito, então chegamos ao fim da fração, não existe expoente e retornamos o lexema encontrado por meio do estado 21.

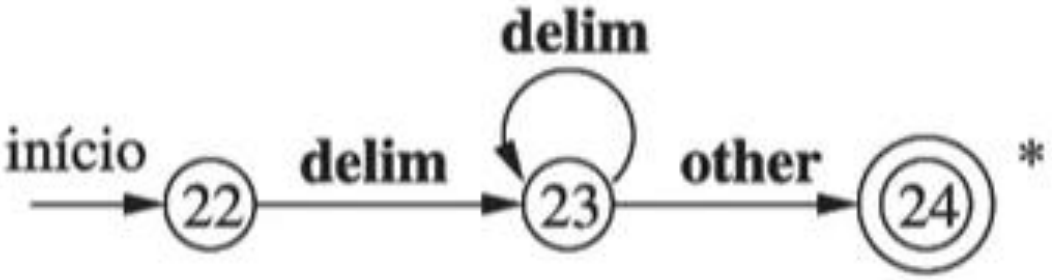
Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Término do Exemplo em Andamento**
    * O último diagrama de transição, mostrado na Fig. 10 a seguir, é para espaços em branco.

    * Nesse diagrama, procuramos por um ou mais caracteres de "espaço em branco", representado por **delim** nesse diagrama – tipicamente, esses caracteres são espaços em branco, tabulações, quebras de linha e talvez outros caracteres que não são considerados pelo projeto da linguagem como parte de um token.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Término do Exemplo em Andamento**
    * Abaixo, Fig. 10



Um diagrama de transição para espaços em branco.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Término do Exemplo em Andamento**
    * Observe que, no estado 24, encontramos um bloco de caracteres de espaço em branco consecutivos, seguidos por um caractere diferente de espaço em branco.

    * Recuamos na entrada para começar em um símbolo que não seja espaço em branco, mas **não retornamos ao analisador sintático**. Em vez disso, temos de reiniciar o processo de análise léxica após o espaço em branco.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Arquitetura de um Analisador Léxico Baseado em Diagrama de Transição**
    * Existem várias formas de usar uma coleção de diagramas de transição para construir um analisador léxico.
    * Independentemente da estratégia geral adotada, cada estado é representado por um trecho de código.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Arquitetura de um Analisador Léxico Baseado em Diagrama de Transição**
    * Podemos imaginar uma variável **`state`** contendo o número do estado corrente para um diagrama de transição. Um comando *switch* baseado no valor de **`state`** nos leva ao código para cada um dos possíveis estados, onde encontramos a ação deste estado.

    * Normalmente, o código para um estado é, ele mesmo, um comando *switch* ou um desvio de múltiplos caminhos que determina o próximo estado lendo e examinando o próximo caractere da entrada.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Arquitetura de um Analisador Léxico Baseado em Diagrama de Transição**
    * Na Fig. 11, vemos um esboço de uma função em C++ **`getRelop()`**, cuja tarefa é simular o diagrama de transição da Fig. 6 e retornar um objeto do tipo `TOKEN`, ou seja, um par consistindo no nome do **token** (que precisa ser relop neste caso) e um valor de atributo (o código para um dos seis operadores de comparação neste caso).

    * A função **`getRelop()`** primeiro cria um novo objeto **`retToken`** e inicia seu primeiro componente como `RELOP`, o código simbólico para o token **relop**.

Descrição de Símbolos com Autômatos Finitos

* Abaixo, Fig.11

```
TOKEN getRelop()
{
     TOKEN retToken = new(RELOP);
     while(1) { /* repete o processamento de caractere até que ocorra
                   um retorno ou uma falha */
          switch(state) {
               case 0: c = nextChar();
                       if ( c == '<' ) state = 1;
                       else if ( c == '=' ) state = 5;
                       else if ( c == '>' ) state = 6;
                       else fail(); /* lexema não é um relop */
                       break;
               case 1: ...
               ...
               case 8: retract();
                       retToken.attribute = GT;
                       return(retToken);
          }
     }
}
```

Esboço da implementação do diagrama de transição de **relop**.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Arquitetura de um Analisador Léxico Baseado em Diagrama de Transição**
    * Vemos o comportamento típico de um estado em *case 0*, uma situação na qual o estado corrente é 0. Uma função **`nextChar()`** obtém o próximo caractere de entrada e o atribui à variável local ***c***.
    * Depois, verificamos se a variável ***c*** casa com um dos três caracteres que esperamos encontrar, fazendo a transição de estado especificada pelo diagrama da Fig. 6 em cada caso.
    * Por exemplo, se o próximo caractere de entrada for =, vamos para o estado 5.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Arquitetura de um Analisador Léxico Baseado em Diagrama de Transição**
    * Se o próximo caractere de entrada não for um que possa iniciar um operador de comparação, então uma função **`fail()`** é chamada. O que **`fail()`** faz depende da estratégia global de recuperação de erro do analisador léxico.
    * Ele deverá reiniciar o apontador do buffer, a fim de permitir que outro diagrama de transição seja aplicado ao verdadeiro início da entrada não processada.
    * Ele poderia, então, mudar o valor de **`state`** para ser o estado inicial para outro diagrama de transição, que pesquisará outro token.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Arquitetura de um Analisador Léxico Baseado em Diagrama de Transição**
    * Alternativamente, se não houver outro diagrama de transição que permaneça não utilizado, **`fail()`** poderia iniciar uma fase de correção de erro que tentará reparar a entrada e encontrar um lexema, conforme vimos em "*Erros Léxicos*".
    * No código da Fig. 11, também podemos ver a ação para o estado 8. Como o estado 8 contém um *, temos de recuar o apontador da entrada uma posição (ou seja, colocar ***`c`*** de volta no fluxo de entrada).
    * Essa tarefa é feita pela função **`retract()`**. Como o estado 8 representa o reconhecimento do lexema >=, atribuímos o segundo componente do objeto retornado, que supomos ser chamado **`atribute`**, para GT, o código para esse operador.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Arquitetura de um Analisador Léxico Baseado em Diagrama de Transição**
    * Para exemplificar a simulação de um diagrama de transição, vamos considerar as formas pelas quais o código da Fig. 11 poderia encaixar-se em um **analisador léxico completo**.

    * **1)** Poderíamos arrumar os diagramas de transição para cada token ser testado sequencialmente. Depois, a função **`fail()`** reinicia o apontador do buffer e inicia o próximo diagrama de transição, cada vez que for chamado.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Arquitetura de um Analisador Léxico Baseado em Diagrama de Transição**
    * Esse método nos permite usar diagramas de transição para cada uma das palavras-chave, como aquele sugerido na Fig. 8. A única restrição é que temos de utilizá-los antes de usarmos o diagrama para **id**, a fim de que as palavras-chave sejam palavras reservadas.

    * **2)** Poderíamos executar os diversos diagramas de transição "em paralelo", alimentando o próximo caractere de entrada em todos eles e permitindo que cada um faça quaisquer transições necessárias.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Arquitetura de um Analisador Léxico Baseado em Diagrama de Transição**
    * Se usarmos essa estratégia, precisamos estar atentos para resolver a situação em que um diagrama encontra um lexema que casa com seu padrão, enquanto um ou mais diagramas ainda são capazes de processar a entrada. A estratégia normal é pegar o prefixo mais longo da entrada que casa com qualquer padrão.

    * Essa regra nos permite, por exemplo, preferir o identificador **`thenext`** à palavra-chave **`then`**, ou o operador **`->`** ao operador **`-`**.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Arquitetura de um Analisador Léxico Baseado em Diagrama de Transição**
    * **3)** A técnica preferida, e a que adotaremos, é combinar todos os diagramas de transição em um único.

    * Permitiremos que o diagrama de transição leia a entrada até que não haja mais estado seguinte possível, e depois pegamos o lexema mais longo que casou com qualquer padrão, conforme discutimos anteriormente no item 2).

    * Em nosso exemplo, essa combinação é fácil, pois dois tokens não podem começar com o mesmo caractere; ou seja, o primeiro caractere nos diz imediatamente qual token estamos procurando.

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Arquitetura de um Analisador Léxico Baseado em Diagrama de Transição**
    * Assim, podemos simplesmente combinar os estados 0, 9, 12 e 22 em um único estado inicial, deixando as outras transições intactas.
    * Em geral, contudo, o problema de combinar os diagramas de transição para vários tokens é mais complexo, conforme veremos ao longo do curso.
    * Vejamos na Fig. 12, a seguir, o exemplo da junção dos diagramas de transição das Figuras 6, 7, 9 e 10:

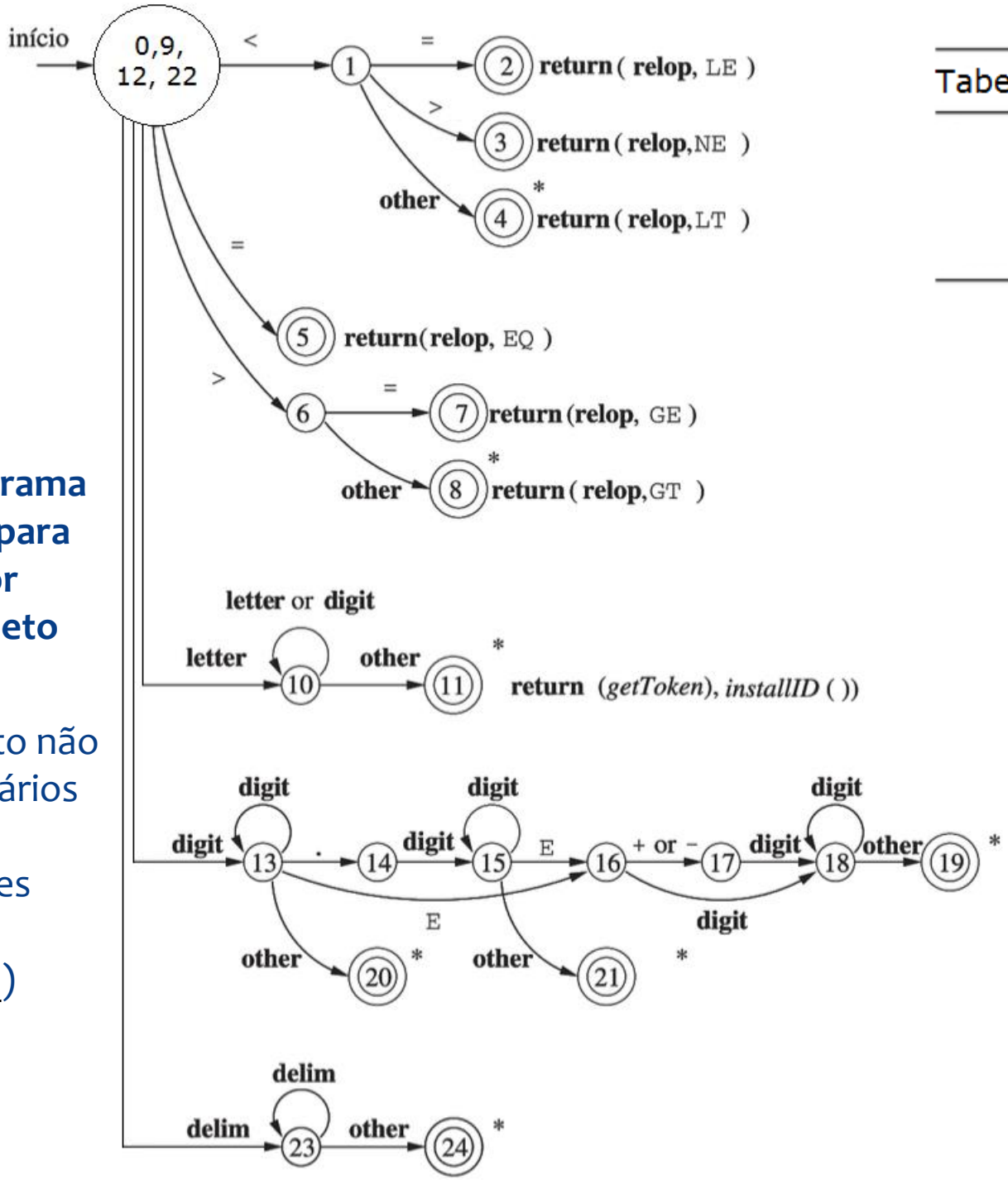


Tabela de Símbolos

```
if
then
else
...
```

* Fig. 12 - **Diagrama de Transição para um Analisador Léxico Completo**

* Obs.: O projeto não inclui comentários (Ex.: // e /**/) nem caracteres especiais (Ex.: $#@&%_)

Descrição de Símbolos com Autômatos Finitos

* **Reconhecimento de Tokens**
  * **Arquitetura de um Analisador Léxico Baseado em Diagrama de Transição**
    * O diagrama da Fig. 12 serve como base para a construção de outros tipos de analisadores léxicos.
    * Dica: Revise o material de FTC para relembrar o conteúdo de autômatos finitos.

Resumo

* ***Tokens***: O analisador léxico lê o programa fonte e produz, como saída, uma sequência de tokens que normalmente são passados, um de cada vez, para o analisador sintático. Alguns tokens podem consistir apenas em um nome de token, enquanto outros também podem ter um valor léxico associado que dê informações sobre a instância em particular do token que foi encontrado na entrada.

* ***Lexemas***: Toda vez que o analisador léxico retorna um token ao analisador sintático, ele possui um lexema associado – a sequência de caracteres de entrada que o token representa.

* ***Buffering***: Como normalmente é necessário ler adiante na entrada a fim de ver onde o próximo lexema termina, em geral é necessário que o analisador léxico coloque sua entrada em buffer. Técnicas especializadas de *buffering* para a leitura dos caracteres da entrada podem acelerar o compilador significativamente.

Resumo

* ***Padrões***: Cada token possui um padrão que descreve quais sequências de caracteres podem formar os lexemas correspondentes a esse token. O conjunto de palavras, ou cadeias de caracteres, que casam com determinado padrão é chamado de linguagem.

* ***Expressões Regulares***: Essas expressões normalmente são usadas para descrever padrões. Expressões regulares são construídas a partir de caracteres unitários, usando os operadores de união, concatenação e o de *fecho de Kleene*.

* ***Definições Regulares***: Coleções complexas de linguagens, como os padrões que descrevem os tokens de uma linguagem de programação, normalmente são definidos por uma definição regular, que é uma sequência de comandos que definem, cada um, uma variável para representar alguma expressão regular. A expressão regular para uma variável pode usar variáveis previamente definidas em sua expressão regular.

Resumo

* ***Notação de expressão regular estendida***: Diversos operadores adicionais podem aparecer como abreviações em expressões regulares, para facilitar a expressão de padrões. Alguns exemplo incluem o operador + (um-ou-mais), ? (zero-ou-um) e classes de caracteres (a união das cadeias, cada uma consistindo em um caractere).

* ***Diagramas de Transição***: O comportamento de um analisador léxico normalmente pode ser descrito por um diagrama de transição. Esses diagramas possuem estados, cada um representando algo sobre a história dos caracteres vistos durante a pesquisa corrente por um lexema que casa com um dos possíveis padrões. Existem setas, ou transições, de um estado para outro, cada uma indicando os próximos caracteres possíveis da entrada que fazem com que o analisador léxico mude de estado.

Resumo

* ***Autômatos Finitos***: Estes são uma formalização dos diagramas de transição que incluem uma designação de um estado inicial e um ou mais estados de aceitação, além do conjunto de estados, caracteres de entrada e transições entre estados. Os estados de aceitação indicam que o lexema para algum token foi encontrado. Diferente dos diagramas de transição, os autômatos finitos podem fazer transições sob a entrada vazia e também sob os caracteres da entrada.

* ***Autômatos Finitos Determinísticos***: Um AFD é um tipo especial de autômato finito que tem exatamente uma transição saindo de cada estado para cada símbolo da entrada. Além disso, as transições sob a entrada vazia não são permitidas. O AFD é facilmente simulado e permite uma boa implementação de um analisador léxico, semelhante a um diagrama de transição.

Resumo

* ***Autômatos Finitos Não Determinísticos***: Autômatos que não são AFDs são chamados de não determinísticos. Os AFNs normalmente são mais fáceis de projetar do que os AFDs. Outra arquitetura possível para um analisador léxico é tabular todos os estados em que os AFNs para cada um dos padrões possíveis podem estar, na medida em que lemos os caracteres de entrada.

* ***Conversão entre representações de padrões***: É possível converter qualquer expressão regular para um AFN aproximadamente do mesmo tamanho, reconhecendo a mesma linguagem que a expressão regular define. Além do mais, qualquer AFN pode ser convertido para um AFD para o mesmo padrão, embora, no pior caso (nunca encontrado nas linguagens de programação comuns), o tamanho do autômato pode crescer exponencialmente. Também é possível converter qualquer AFN ou AFD em uma expressão regular que define a mesma linguagem reconhecida pelo autômato finito.

Resumo

* ***Lex***: Existe uma família de sistemas de software, incluindo *Lex* e *Flex*, que são geradores de analisador léxico. O usuário especifica os padrões para os tokens usando uma notação de expressão regular estendida. O *Lex* converte essas expressões para um analisador léxico que é essencialmente um autômato finito determinístico que reconhece qualquer um dos padrões.

* ***Minimização dos autômatos finitos***: Para cada AFD, existe um AFD mínimo aceitando a mesma linguagem. Além disso, AFD mínimo para determinada linguagem é único, exceto para os nomes dados aos diversos estados.

* ***Tabela de Símbolos***: São estruturas de dados que mantêm informações sobre identificadores. As informações são colocadas na tabela de símbolos quando a declaração de um identificador é analisada. Uma ação semântica obtém informações da tabela de símbolos sempre que o identificador for usado subsequentemente, por exemplo, como um fator em uma expressão.

Obrigado.

joaopauloaramuni@gmail.com
joaopauloaramuni@fumec.br

UNIVERSIDADE FUMEC
50 Anos