Universidade de Santa Cruz do Sul – UNISC
Departamento de informática

COMPILADORES

Análise léxica

Parte 02

Prof. Geovane Griesang
geovanegriesang@unisc.br

Análise léxica

Reconhecimento de *tokens*

Na aula anterior, padrões foram expressados com o uso de Expressões Regulares (ER).

Agora, tem-se que estudar os padrões para todos os *tokens* necessários e criar um trecho de código que examina a cadeia de entrada e encontra um prefixo que seja um lexema casando (ligando) com um dos padrões.

Um prefixo da cadeia S é qualquer cadeia obtida pela remoção de zero ou mais símbolos no final de S.

Exemplo: *ban*, *banana* e ε são prefixos de banana.

Análise léxica

Reconhecimento de *tokens*

Exemplo: O fragmento da gramática abaixo descreve uma forma simples de comandos de desvio e expressões condicionais.

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

Essa sintaxe é semelhante à da linguagem Pascal, pois ***then*** aparece explicitamente após as condições.

Análise léxica

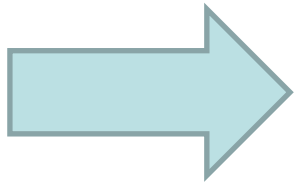
Reconhecimento de *tokens*

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

Continuando... Para ***relop***, usa-se os operadores de comparação de ling. como Pascal ou SQL, onde = é “igual” e <> é “não igual”.

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.

Sendo assim, os padrões para esses *tokens* são descritos por meio de definições regulares, conforme exibido no próximo exemplo.



Análise léxica

Reconhecimento de *tokens*

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	→	< | > | <= | >= | = | <>

Para esta linguagem, o analisador léxico reconhecerá as palavras-chave *if*, *then* e *else*, além dos lexemas que casam (combinam) com os padrões para *relop*, *id* e *number*.

Análise léxica

Reconhecimento de *tokens*

Continuação do exemplo...

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	→	< | > | <= | >= | = | <>

Para simplificar, supõem-se que as palavras-chave também são *palavras reservadas*: ou seja, não são identificadores, embora seus lexemas casem com o padrão para identificadores.

Além disso, atribui-se ao analisador léxico a tarefa de remover espaços em branco, reconhecendo o *token ws* definido por:

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

Análise léxica

Reconhecimento de *tokens*

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

Sendo assim, os símbolos *blank*, *tab* e *newline* são usados para expressar caracteres ASCII com os mesmos nomes.

O *token ws* é diferente dos demais *tokens* porque, quando é reconhecido, não é retornado ao analisador sintático, mas a análise léxica é reiniciada a partir do caractere seguinte ao espaço em branco.

token seguinte que é retornado ao analisador sintático.

ws = (***blank*** | ***tab*** | ***newline***)$^{+}$

Análise léxica

Lexemas	Nome *Token*	Valor do atributo
Qualquer *ws*	-	-
if	if	-
then	then	-
else	else	-
Qualquer *id*	id	Apontador para a entrada de tabela
Qualquer *number*	number	Apontador para a entrada de tabela
<	relop	LT *(Less Than)*
<=	relop	LE *(Less than or Equal)*
=	relop	EQ (*EQuals*)
<>	relop	NE (*Not Equals*)
>	relop	GT *(Greater Than)*
>=	relop	GE *(Greater than or Equal)*

Análise léxica

Lexemas	*Token*	Atributo
Q. ws	-	-
if	if	-
then	then	-
else	else	-
Q. id	id	Apontador p/ tabela
Q. number	number	Apontador p/ tabela
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Reconhecimento de *tokens*

O objetivo do analisador léxico está resumido na tabela, que mostra cada lexema ou família de lexemas, onde o nome de *token* e o valor do atributo são retornados ao analisador sintático.

Observações

As constantes simbólicas (LT, LE, EQ, NE, GT, GE) são usadas como valor de atributo para indicar qual instancia do *token* **relop** foi encontrada.

O operador encontrado irá influenciar o código gerado pelo compilador.

Análise léxica

Diagrama de transições

Esse diagrama é usado p/ determinar a sequência de ações executadas pelo analisador léxico no processo de reconhecimento de um *token*.

As posições no diagrama são representadas através de um círculo e são chamadas de estado.

Os estados são conectados por setas, denominadas lados ou arestas.

Os lados são rotulados com caracteres que indicam as possíveis entrada que podem aparecer após o diagrama de estado ter atingido um dado estado.

Análise léxica

Diagrama de transições

O rótulo outro se refere a qualquer caractere de entrada que não seja o indicado pelos demais rótulos que deixam o estado.

Um círculo duplo determina um estado de aceitação (estado final).

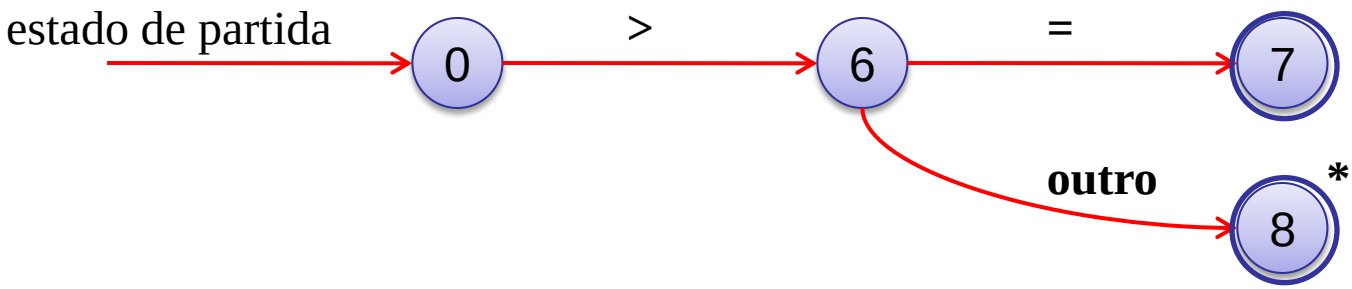
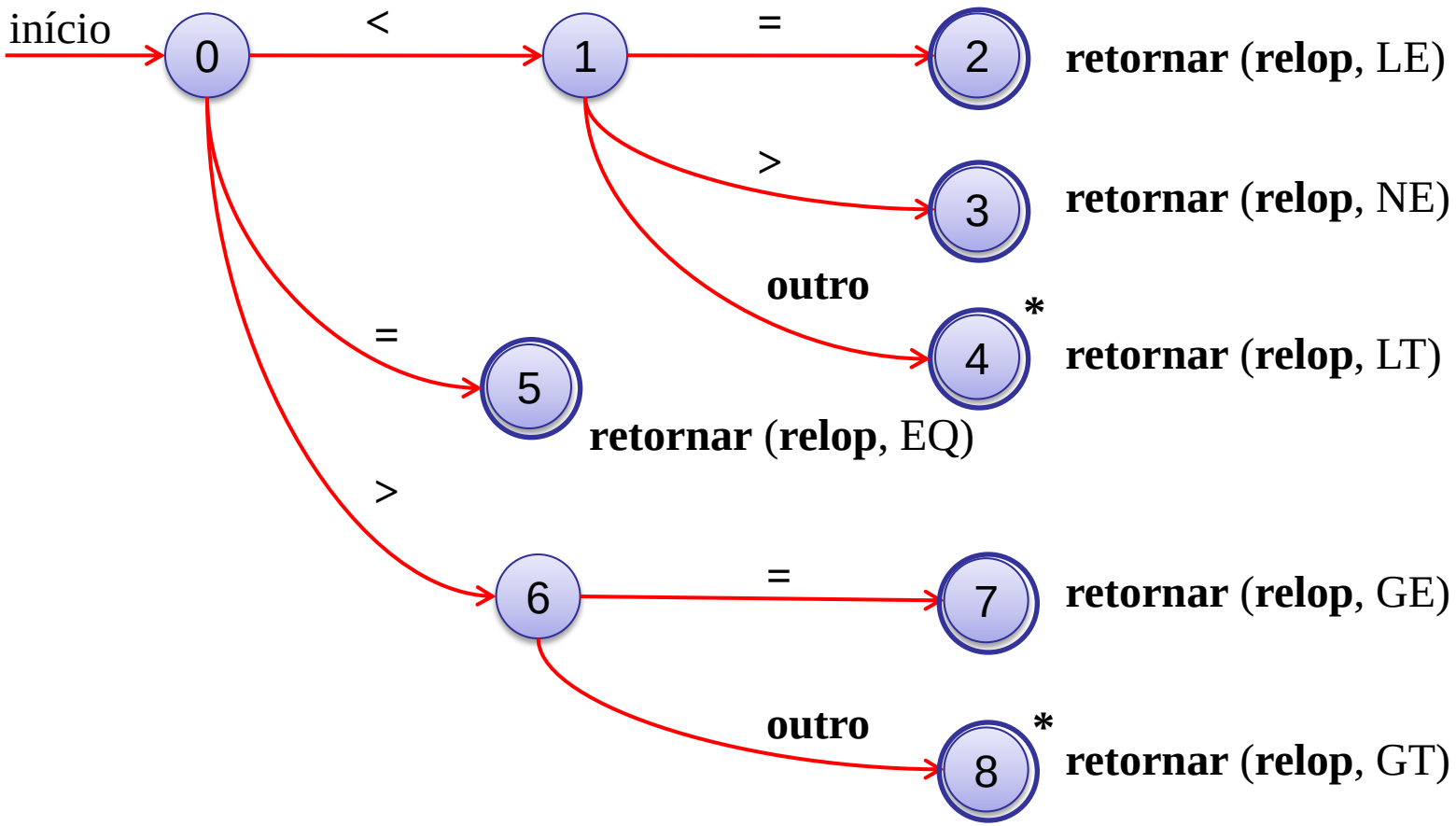


Diagrama de transição para >=

Análise léxica

Diagrama de transições para o ***relop***



Análise léxica

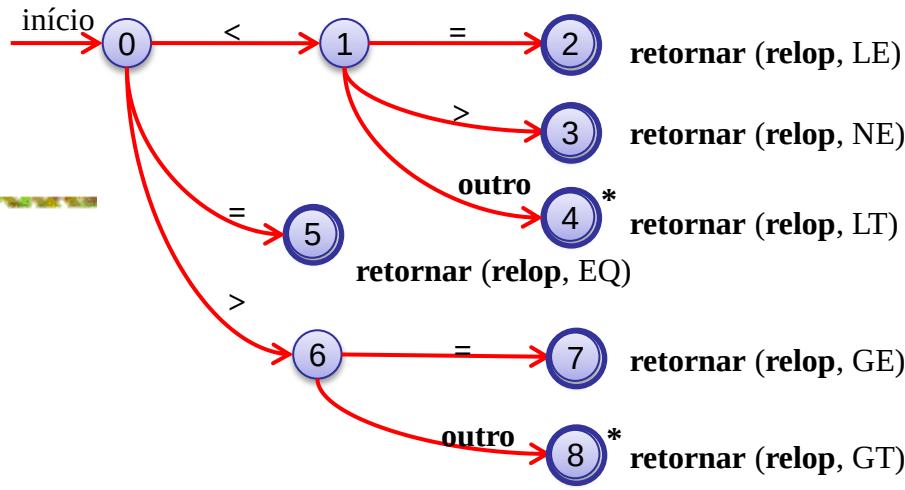


Diagrama de transições para o ***relop***

Reconhece os lexemas casando com o *token* **relop**.

Começa no estado 0 (inicial). Se for encontrado o < como o 1º símbolo de entrada, então entre os lexemas que casam com o padrão **relop**, só pode-se estar procurando <, <> ou <=.

Portanto, vai-se para o estado 1 e o caractere seguinte é examinado. Se for =, então reconhece-se o lexema <=.

Entra-se no estado 2, retornando o *token* **relop** com o atributo LE, que é a constante que representa esse operador de comparação.

Análise léxica

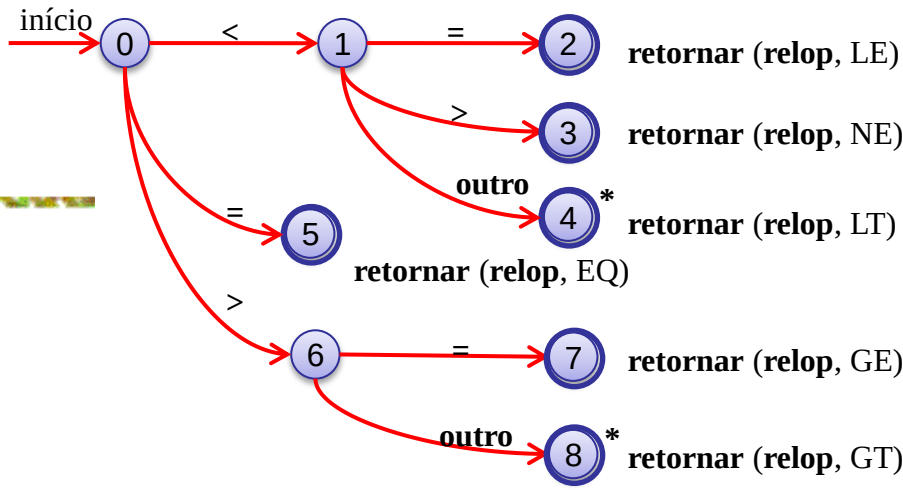


Diagrama de transições para o ***relop***

Se no estado 1 o caractere seguinte for >, então temos o lexema <>. Desta forma, entra-se no estado 3 para retornar uma indicação de que o operador "não-igual" foi encontrado.

Com outro caractere, o lexema é <. Entra-se no estado 4 para retornar essa informação.

Observação: No estado 4 tem um * para indicar que deve-se recuar uma posição.

Análise léxica

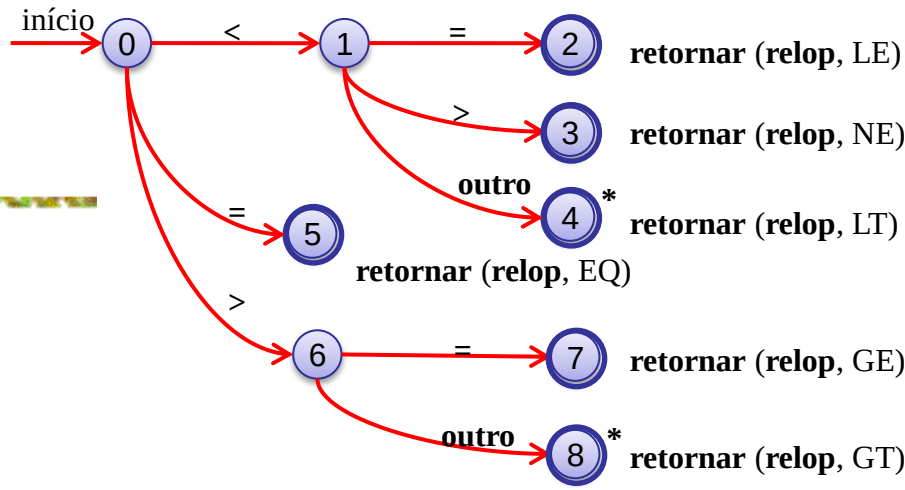


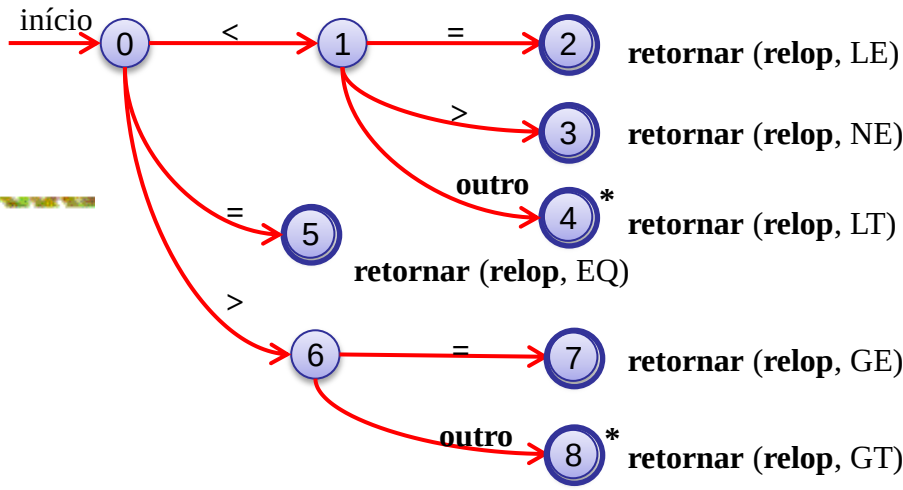
Diagrama de transições para o ***relop***

Por outro lado, se no estado 0 o primeiro caractere for =, então esse único caractere precisa ser o lexema. Imediatamente retorna-se esse fato do estado 5.

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

Análise léxica

Diagrama de transições para o ***relop***



Pode-se observar que, se no estado 0, for visto qualquer caractere além de <, = ou >, é possível que não se esteja vendo o lexema de um relop.

Sendo assim, este diagrama de transição não poderia ser usado.

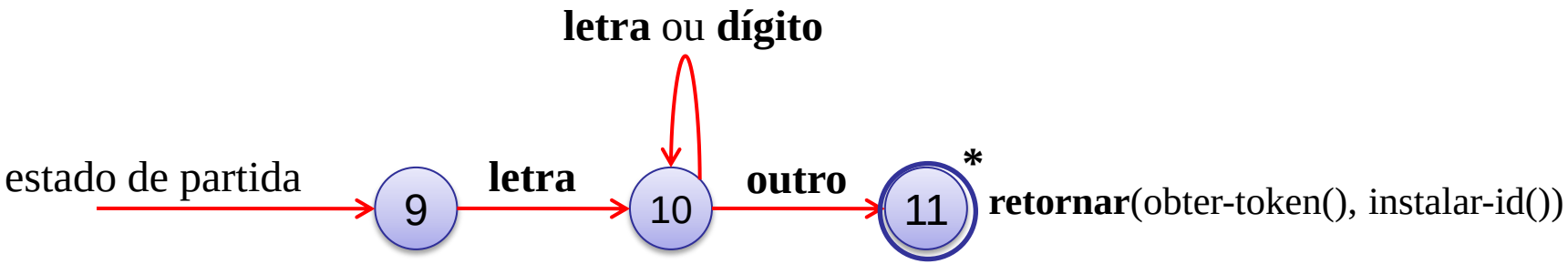
Análise léxica

Reconhecimento de palavras reservadas e identificadores

Reconhecer palavras-chave e identificadores implica um problema.

No geral, palavras-chave como *if* ou *then* são reservadas, de modo que as palavras-chave não são identificadores embora se pareçam com eles.

Exemplo: O diagrama abaixo é usado para procurar identificadores, mas também pode reconhecer palavras-chave (*if*, *then*, *else*).



Análise léxica

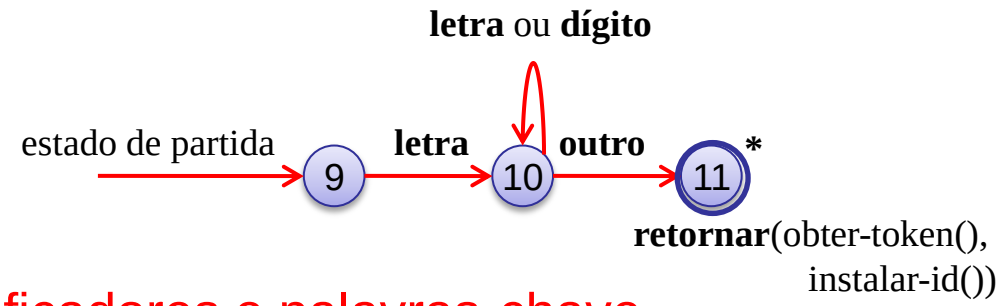


Diagrama de transição para identificadores e palavras-chave

Pode-se lidar com palavras reservadas que se parecessem com id's de duas formas:

1) Instala-se inicialmente as palavras reservadas na tabela de símbolos.

   Um campo da entrada da tabela de símbolos indica que essas cadeias de caracteres nunca são identificadores comuns, e diz qual *token* eles representam.

   Ao encontrar um id, um chamada de *installID* (instalar-id) o coloca na tab. de símbolos caso ele ainda não esteja lá, e retorna um apontador p/ a entrada da tab. de símbolos referente ao lexema encontrado.

Análise léxica

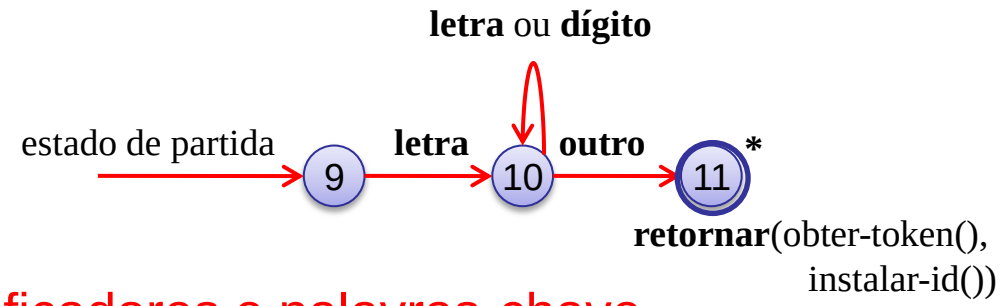


Diagrama de transição para identificadores e palavras-chave

Continuação...

Qualquer identificador que não esteja na tab. de símbolos durante a análise léxica não pode ser uma palavra reservada, portanto, o *token* é **id**.

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

Análise léxica

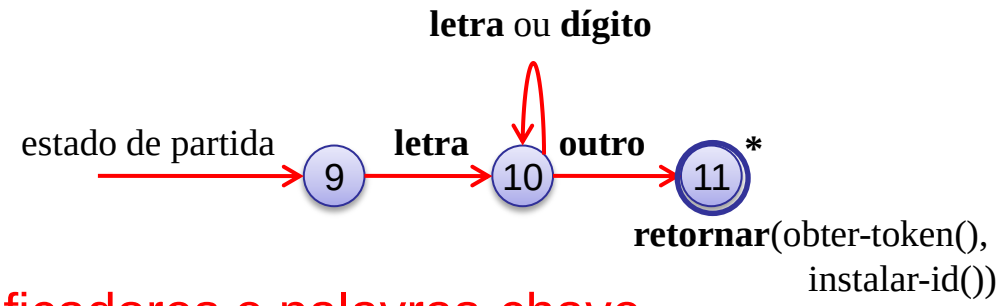


Diagrama de transição para identificadores e palavras-chave

Obter-*token*()

- procura o *lexema* na tabela de símbolos.
- se o *lexema* for uma palavra-chave o *token* correspondente é retornado, caso contrário, é retornado **id.**

Instalar-id()

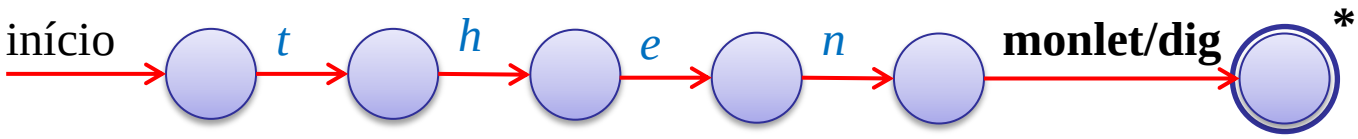
- procura *lexema* na tabela de símbolos.
- se o *lexema* for uma palavra-chave é retornado zero, caso contrário, é retornado um ponteiro para a tabela de símbolos.
- se o *lexema* não for encontrado ele é instalado como uma variável e é retornado um apontador para entrada recém criada.

Análise léxica

Diagrama de transição para identificadores e palavras-chave

2) Cria-se diagramas de transição separados para cada palavra-chave.

Exemplo: diagrama hipotético para a palavra-chave *then*:



Esse diagrama de transição consiste nos estados que representam a situação logo após cada letra sucessiva da palavra vista, seguido por uma “não-letra-ou-dígito”.

Ou seja, qualquer caractere que não possa ser a continuação de um identificador.

Análise léxica

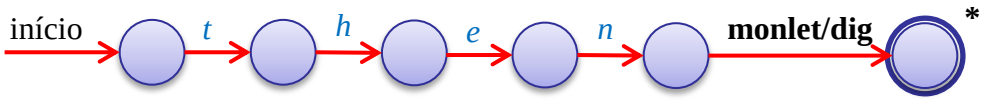


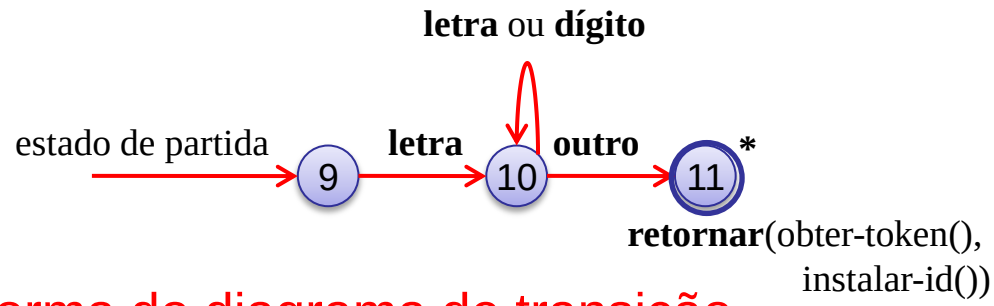
Diagrama de transição para identificadores e palavras-chave

Continuação...

Verifica-se se o identificador terminou, ou, do caso contrário, retorna-se o *token then* em situações nas quais o *token* correto é *id*, com um lexema como *thenextvalue* que possui *then* como prefixo próprio.

Se essa técnica for adotada, será preciso priorizar os *tokens* de modo que os *tokens* de palavra reservada sejam reconhecidos em preferência ao id, quando o lexema casa com os dois padrões.

Análise léxica



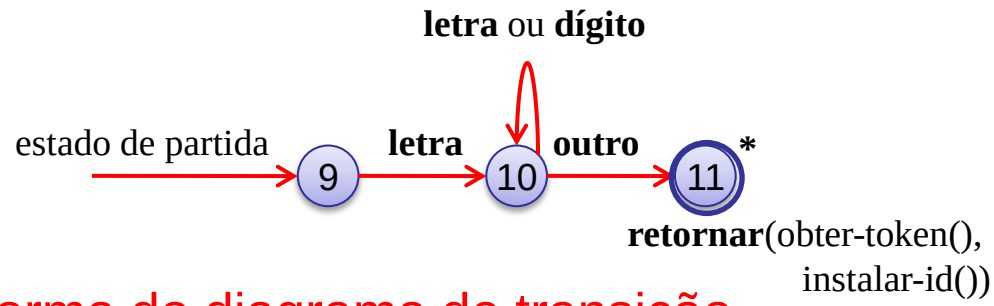
Término do exemplo da primeira forma do diagrama de transição para identificadores e palavras-chave

O diagrama de transição em questão, possui uma estrutura simples, onde a partir do estado 9, verifica-se se o lexema começa por uma letra e vai para o estado 10 nesse caso.

Permanece-se no estado 10 enquanto a entrada tiver letras e dígitos.

A partir do momento que for encontrado qualquer símbolo diferente de uma letra e/ou dígito, passa-se para o estado 11 e aceita-se o lexema encontrado.

Análise léxica



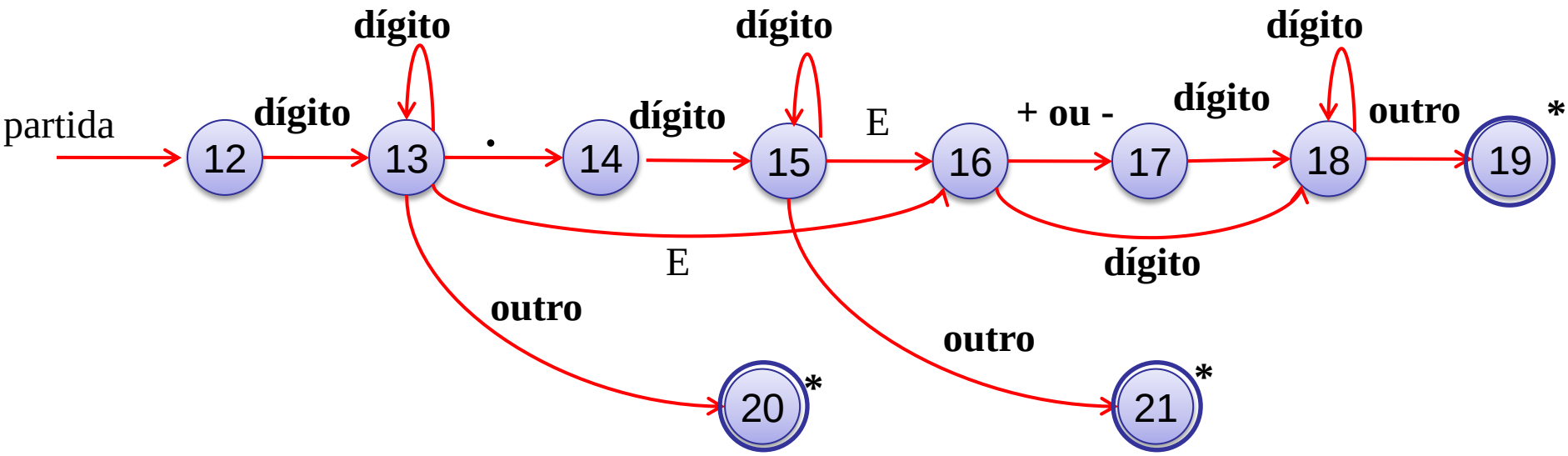
Término do exemplo da primeira forma do diagrama de transição para identificadores e palavras-chave

Como o último caractere não faz parte do identificador, tem-se que recuar na entrada uma posição, inserir o que foi encontrado na tab. símbolos e determinar se o que foi encontrado é uma palavra-chave ou um verdadeiro identificador.

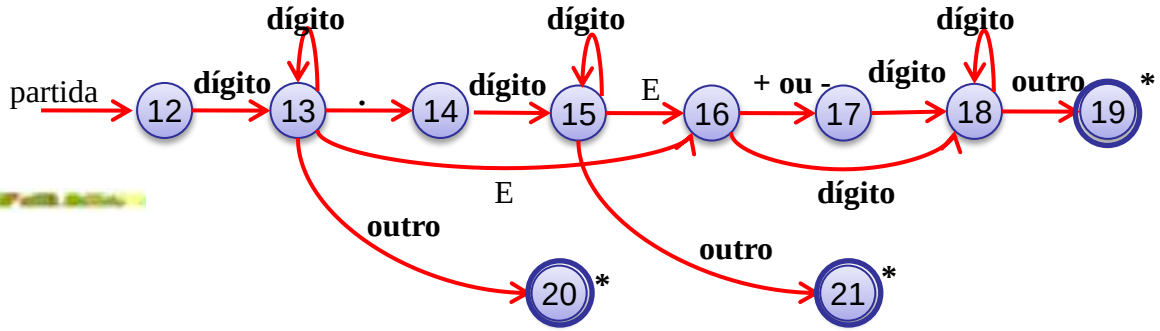
Análise léxica

Término do exemplo da primeira forma do diagrama de transição para identificadores e palavras-chave

Ex.: Diagrama de transição para o *token number* (números sem sinal).



Análise léxica



Término do exemplo

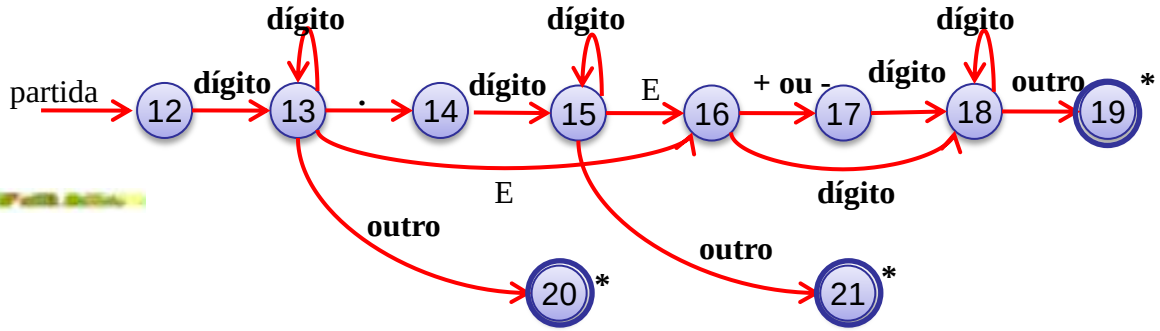
A partir do estado 12, se for encontrado um dígito, vai-se p/ o estado 13.

No estado 13, pode-se ler qualquer quantidade de dígitos adicionais. Mas se for encontrado qualquer símbolo diferente de um dígito ou ponto, vê-se um número na forma de um inteiro (ex.: 123).

Esse caso é tratado pela entrada do estado 20, onde é retornado o *token number* e um apontador para tabela de constantes, na qual o lexema encontrado é inserido.

Se um ponto for lido no estado 13, tem-se um *optionalFraction*.

Análise léxica



Término do exemplo

Entra-se no estado 14 e procura-se um ou mais dígitos adicionais.

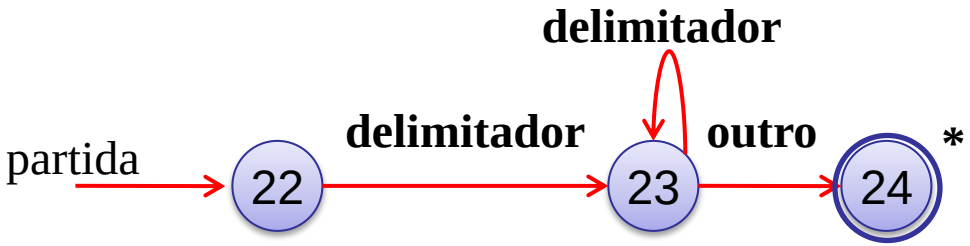
Estado 15 é usado p/ esta finalidade. Se for encontrado um E, então tem-se um *optionExponent*, cujo reconhecimento é uma tarefa dos estados de 16 a 19.

Se, no estado 15, for encontrado algo diferente de um E ou dígito, então, chega-se ao fim da fração. Não existe expoente e é retornado o lexema encontrado por meio do estado 21.

Análise léxica

Término do exemplo

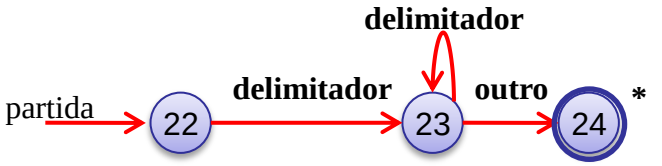
O último diagrama de transição é para espaços em branco.



Portanto, neste diagrama, procura-se um ou mais “caracteres-em-branco”, representados por delimitador.

No geral, esses caracteres são espaços em branco, tabulações, quebra de linha ou outros caracteres que não são considerados pelo projeto da linguagem por parte de um *token*.

Análise léxica



Término do exemplo

No estado 24, é encontrado um bloco de caracteres de espaço em branco consecutivos, seguidos por um caractere diferente de espaços em branco.

Desta forma, recua-se na entrada para começar em um símbolo que não seja espaço em branco, mas não retorna ao analisador sintático.

Em vez disso, tem-se que reiniciar o processo de análise léxica após o espaço em branco.

Análise léxica

Arquitetura de um analisador léxico baseado em um diagrama transição

Existem várias formas de usar uma coleção de diagramas de transição para construir um analisador léxico.

Independente da estratégia usada, cada estado é representado por um trecho de código.

Pode-se imaginar uma variável *state* (estado) contendo o nº do estado corrente para um diagrama de transição.

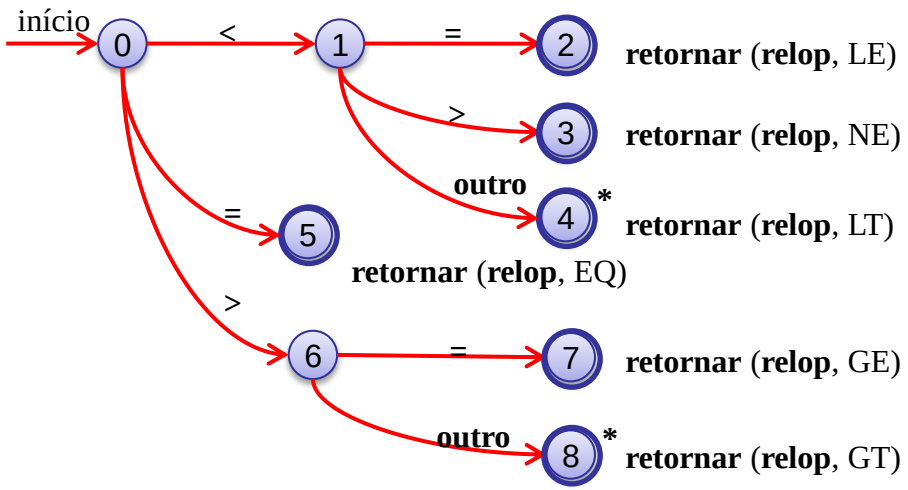
Um comando *switch* baseado no valor de *state* nos leva ao código para um dos possíveis estados, onde encontra-se a ação desse estado.

Análise léxica

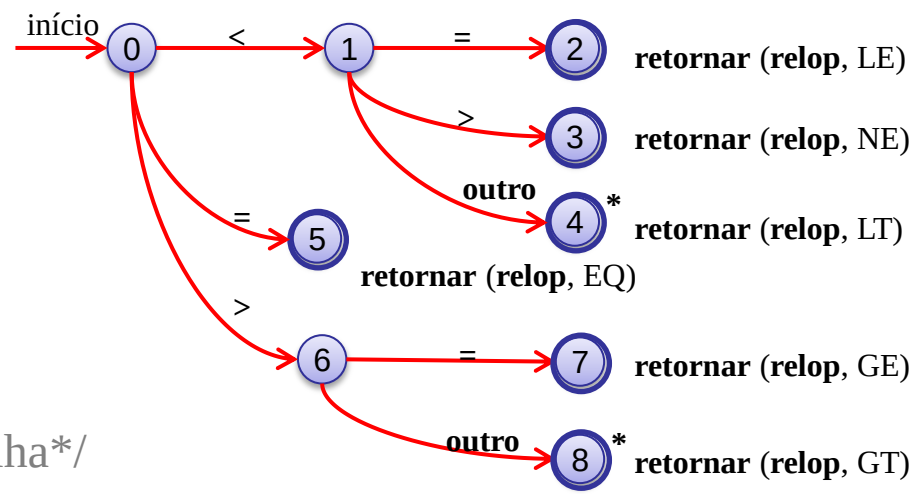
Arquitetura de um analisador léxico baseado em um diagrama transição

No geral, 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.

Exemplo: Vamos criar um esboço de um função em *C++* *getRelop()*, cuja tarefa é simular o seguinte diagrama de transição :



Análise léxica

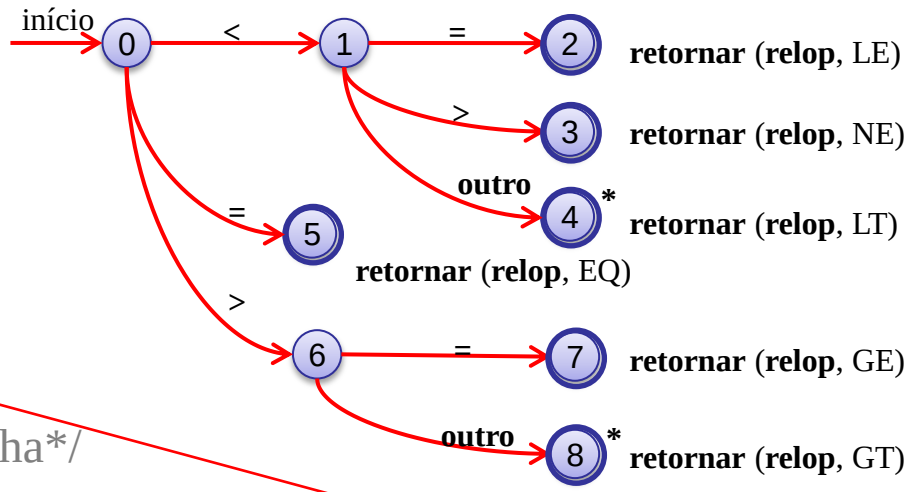



```
TOKEN getRelop{
    TOKEN retToken = new(RELOP);
    while(1){ /* até ocorrer um retorno ou falha*/
        switch(state){
            case 0:   c = nextChar();
                      if (c == '<') state = 1;
                      else if (c == '=') state = 5;
                      else if (c == '>')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

Objetivo da função

Simular o diagrama de transição e retornar um objeto do tipo TOKEN, ou seja, um par consistindo no nome do *token* (que precisa ser **relop** neste caso) e no valor de um atributo (o código para um dos seis operadores de comparação neste caso).

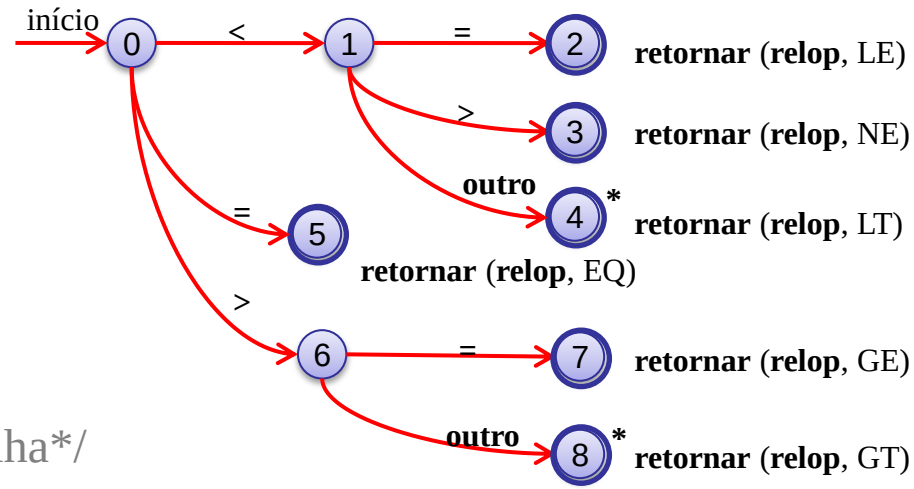
Análise léxica




```
TOKEN getRelop{
    TOKEN retToken = new(RELOP);
    while(1){ /* até ocorrer um retorno ou 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 relop */
            case 1:   ...
            ...
            case 8:   retract();
                      retToken.attribute = GT;
                      return(retToken);
    }}}
```

A função *getRelop()* 1º cria um novo objeto *retToken* e inicia seu 1º componente como *RELOP*, o código simbólico para o *token* **relop**.

Análise léxica

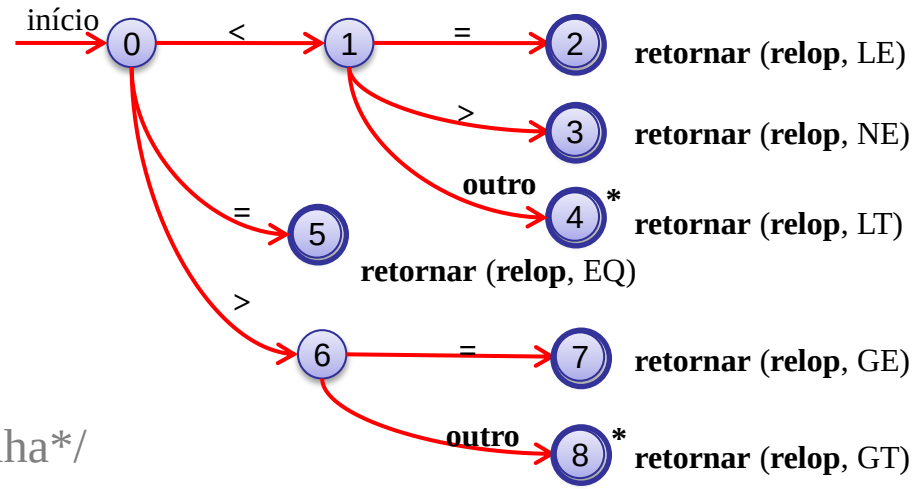



```
TOKEN getRelop{
    TOKEN retToken = new(RELOP);
    while(1){ /* até ocorrer um retorno ou falha*/
        switch(state){
            case 0:   c = nextChar();
                      if (c == '<') state = 1;
                      else if (c == '=') state = 5;
                      else if (c == '>')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

Case 0: o estado corrente é 0.

Uma função *nextChar()* obtém o próximo caractere de entrada e o atribui a variável c.

Análise léxica

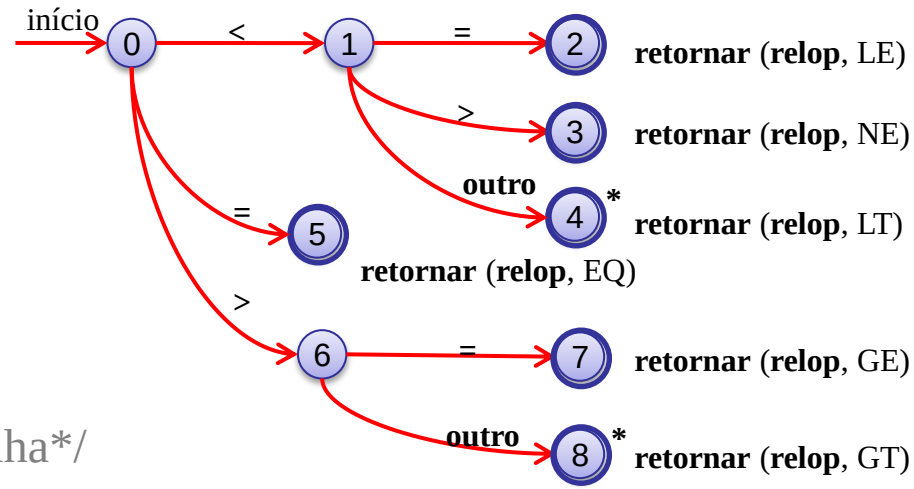



```
TOKEN getRelop{
    TOKEN retToken = new(RELOP);
    while(1){ /* até ocorrer um retorno ou falha*/
        switch(state){
            case 0:   c = nextChar();
                      if (c == '<') state = 1;
                      else if (c == '=') state = 5;
                      else if (c == '>')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

Depois, verifica-se se a variável c casa com um dos três caracteres que esperamos encontrar, fazendo a transição de estado especificada no diagrama.

Exemplo: se o próximo caractere da entrada for =, vai-se p/ o estado 5.

Análise léxica

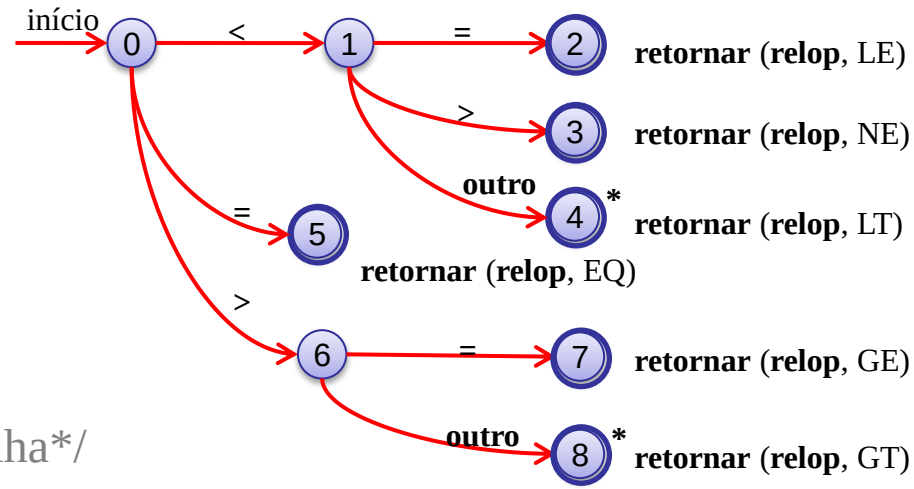



```
TOKEN getRelop{
    TOKEN retToken = new(RELOP);
    while(1){ /* até ocorrer um retorno ou falha*/
        switch(state){
            case 0:   c = nextChar();
                      if (c == '<') state = 1;
                      else if (c == '=') state = 5;
                      else if (c == '>')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

Se o próximo caractere não for um que possa iniciar um operador de comparação, então uma função *fail()* é chamada.

O que a função *fail()* faz depende da estratégia global de recuperação de erro do analisador léxico.

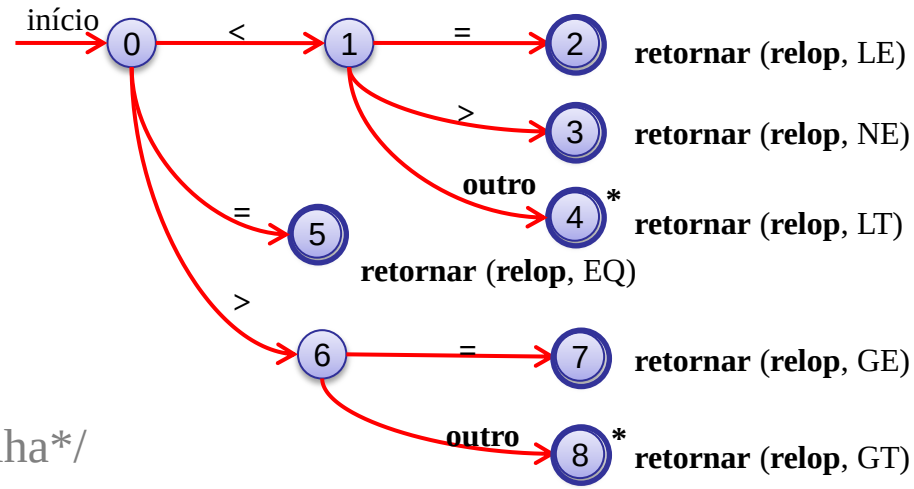
Análise léxica




```
TOKEN getRelop{
    TOKEN retToken = new(RELOP);
    while(1){ /* até ocorrer um retorno ou 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 relop */
            case 1:   ...
            ...
            case 8:   retract();
                      retToken.attribute = GT;
                      return(retToken);
    }}}
```

Ele deverá inicializar o apontador *forward* para *lexemaBegin*, a fim de permitir que outro diagrama de transição seja aplicado ao verdadeiro início da entrada não processada.

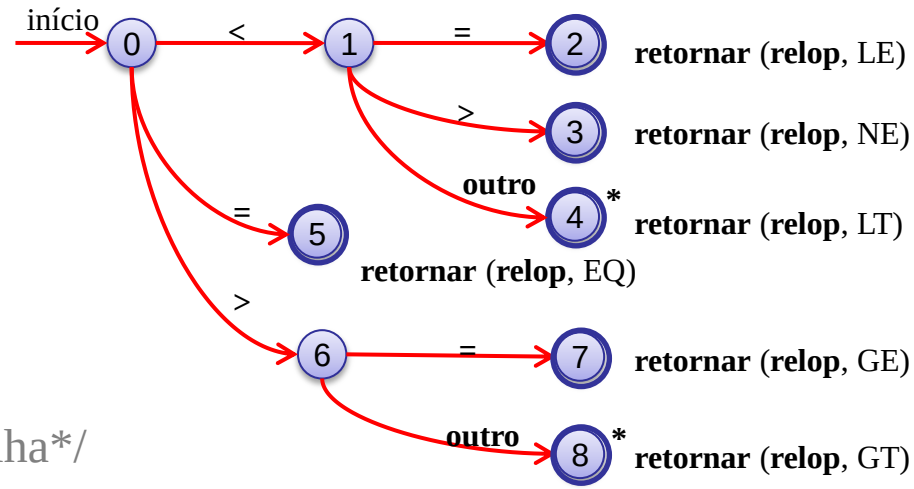
Análise léxica




```
TOKEN getRelop{
    TOKEN retToken = new(RELOP);
    while(1){ /* até ocorrer um retorno ou 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 relop */
            case 1:   ...
            ...
            case 8:   retract();
                      retToken.attribute = GT;
                      return(retToken);
}}}
```

Ele poderia, então, mudar o valor de *state* para ser o estado inicial p/ outro diagrama de transição, que pesquisará outro *token*.

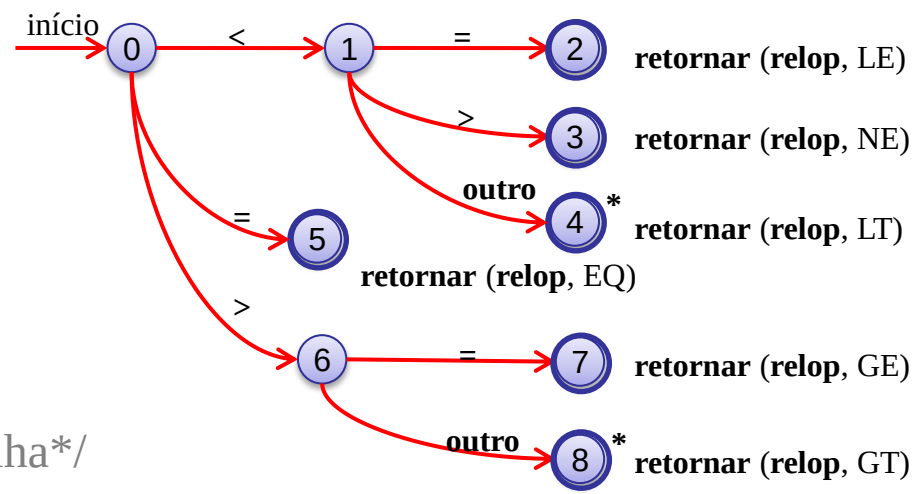
Análise léxica




```
TOKEN getRelop{
    TOKEN retToken = new(RELOP);
    while(1){ /* até ocorrer um retorno ou 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 relop */
            case 1:   ...
            ...
            case 8:   retract();
                      retToken.attribute = GT;
                      return(retToken);
    }}}
```

Outra alternativa, se não houver outro diagrama de transição que permaneça não utilizado, *fail()* poderia iniciar uma fase de correção de erro que tentará recuperar a entrada e encontrar um lexema.

Análise léxica

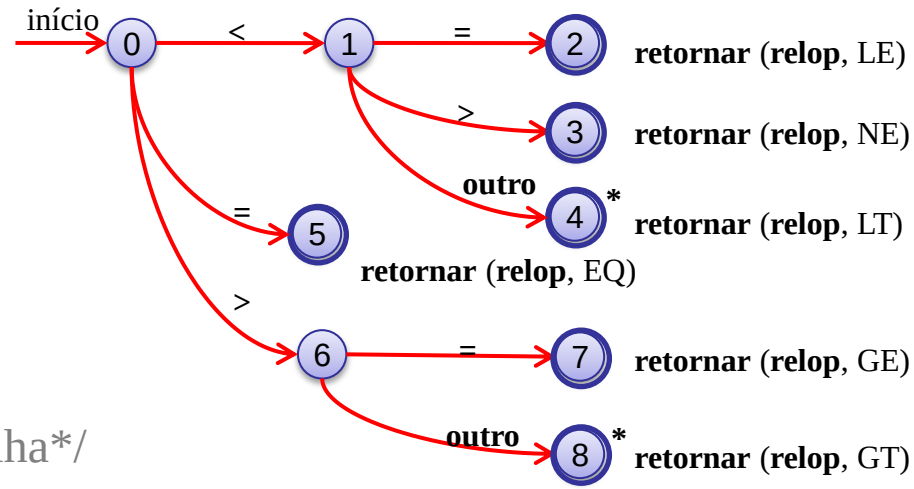



```
TOKEN getRelop{
    TOKEN retToken = new(RELOP);
    while(1){ /* até ocorrer um retorno ou falha*/
        switch(state){
            case 0:   c = nextChar();
                      if (c == '<') state = 1;
                      else if (c == '=') state = 5;
                      else if (c == '>')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

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()*.

Análise léxica




```
TOKEN getRelop{
    TOKEN retToken = new(RELOP);
    while(1){ /* até ocorrer um retorno ou 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 relop */
            case 1:   ...
            ...
            case 8:   retract();
                      retToken.attribute = GT;
                      return(retToken);
}}}
```

Como o estado 8 representa o reconhecimento do lexema >, atribuímos o segundo componente do objeto retornado, que supomos ser chamado *attribute*, para GT, o código para esse apontador.

Análise léxica

Arquitetura de um analisador léxico baseado em um diagrama transição

Para exemplificar a simulação de um diagrama de transição, pode-se usar uma técnica que combina todos os diagramas de transição em um único.

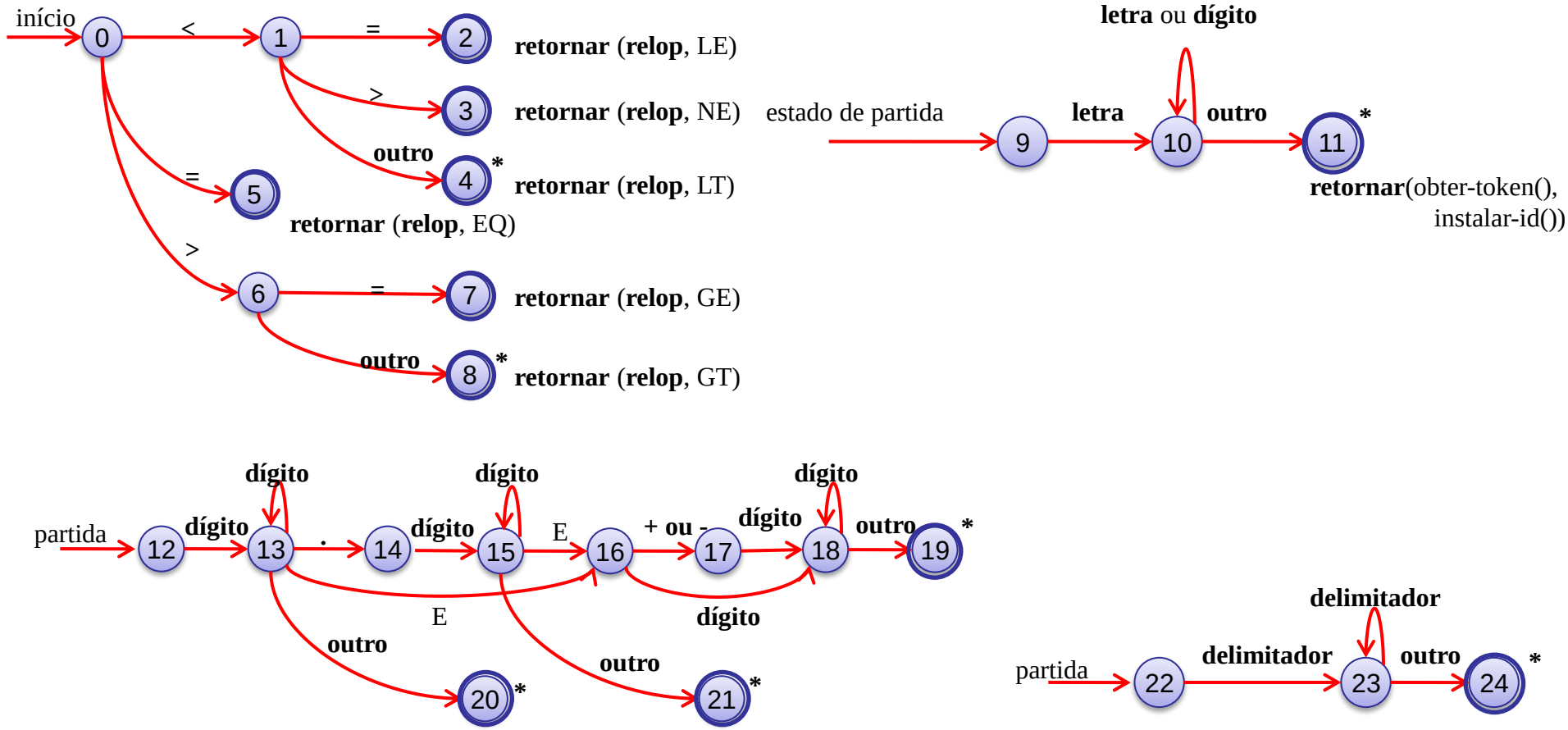
Nesta técnica, permite-se que o diagrama de transição leia a entrada até que não haja mais estado seguinte possível, e depois pega-se o lexema mais longo que casou com qualquer padrão. Exemplo: 12.3E14

Nos exemplos anteriores, essa combinação é fácil, pois dois *tokens* não podem começar com o mesmo caractere. Ou seja, o primeiro caractere diz imediatamente qual *token* estamos procurando.

Assim, pode-se simplesmente combinar os estados 0, 9, 12 e 22 em um único estado inicial, deixando as outras transições intactas.

Análise léxica

Arquitetura de um analisador léxico baseado em um diagrama transição



Análise léxica

O gerador de analisador léxico

Uma *ferramenta Lex* (ou uma mais recente: *Flex*) permite especificar um analisador léxico definindo expressões regulares para descrever padrões para *tokens*.

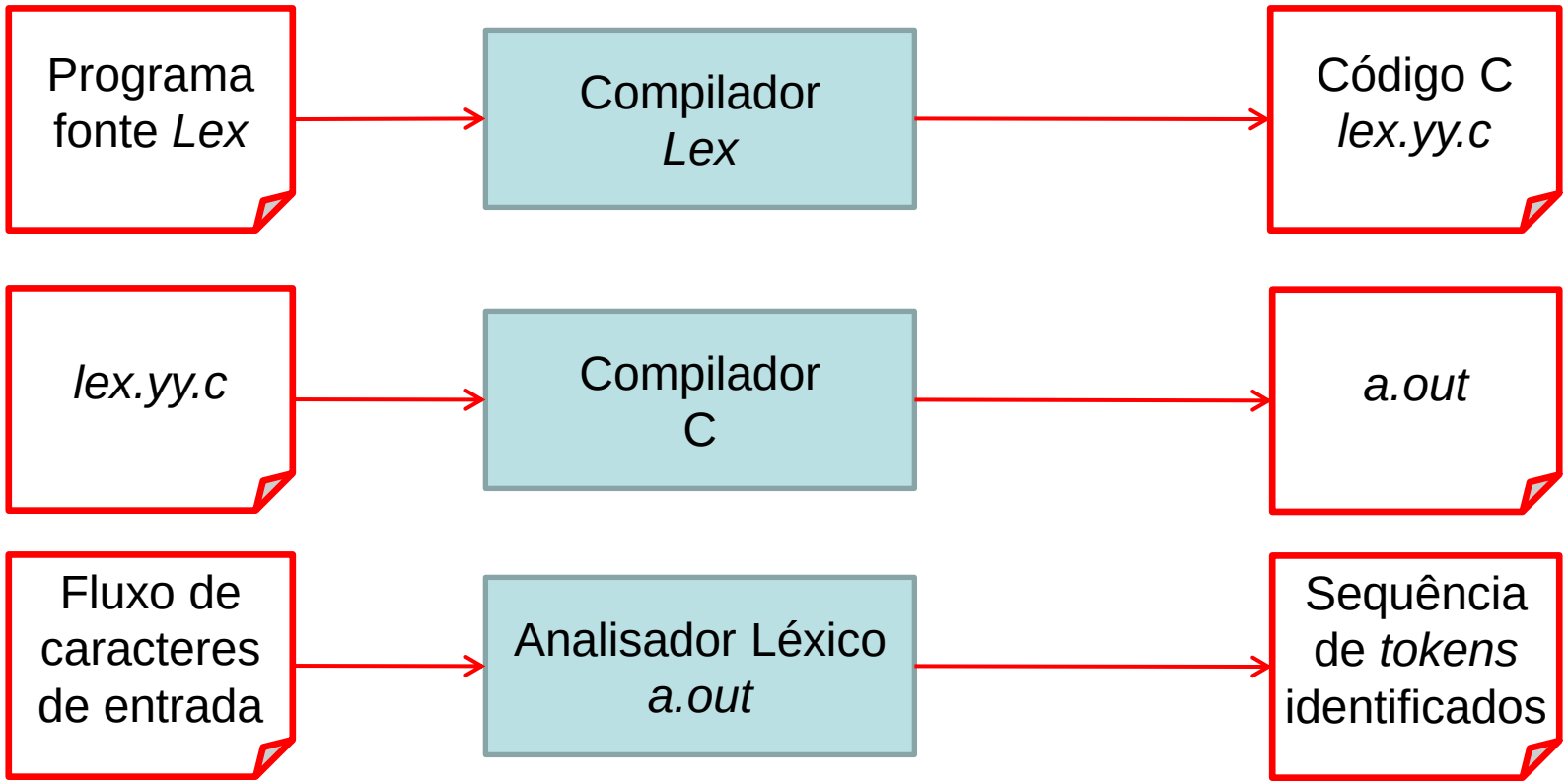
A notação de entrada para a *ferramenta Lex* é chamada *linguagem Lex*, e a ferramenta em si é o *compilador Lex*.

Internamente, o *compilador Lex* transforma os padrões em um diagrama de transição e gera código em um arquivo chamado *lex.yy.c*, que simula esse diagrama de transição.

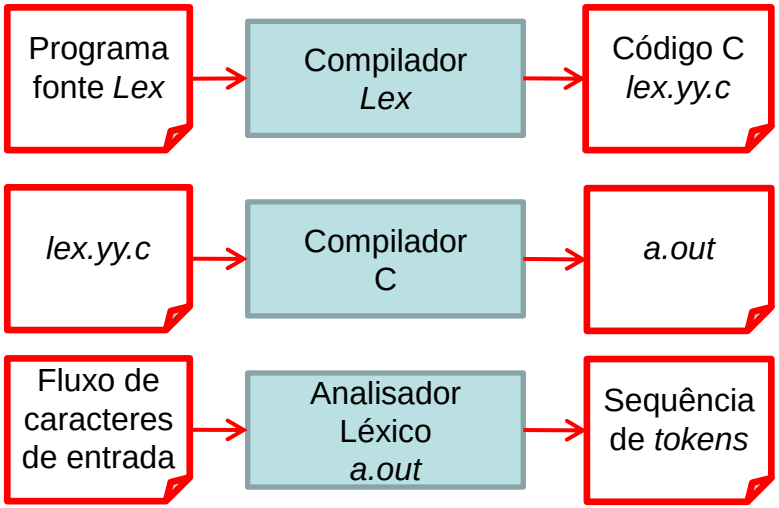
Análise léxica

O gerador de analisador léxico – Uso do *Lex*

Criação de um analisador léxico com o *Lex*

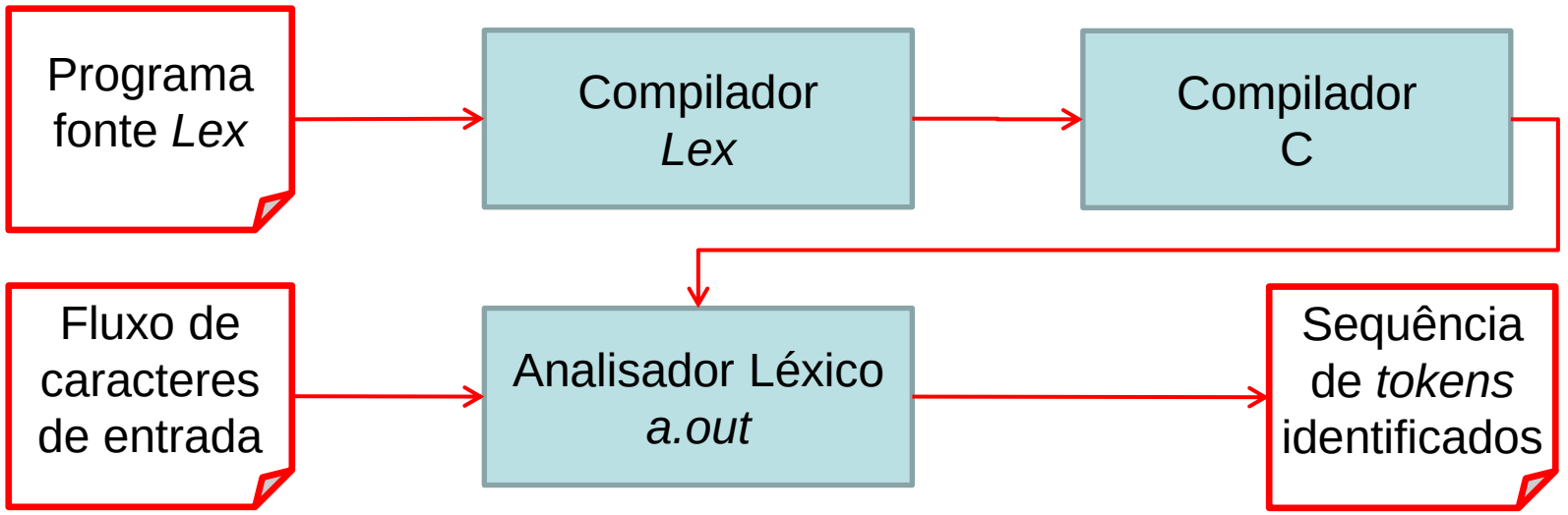


Análise léxica

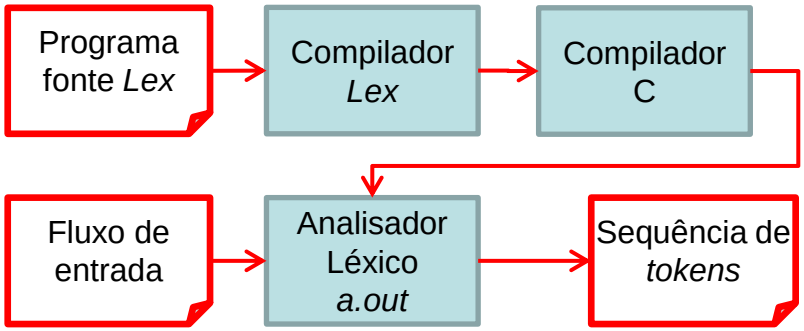


O gerador de analisador léxico – Uso do *Lex*

Criação de um analisador léxico com o *Lex*



Análise léxica



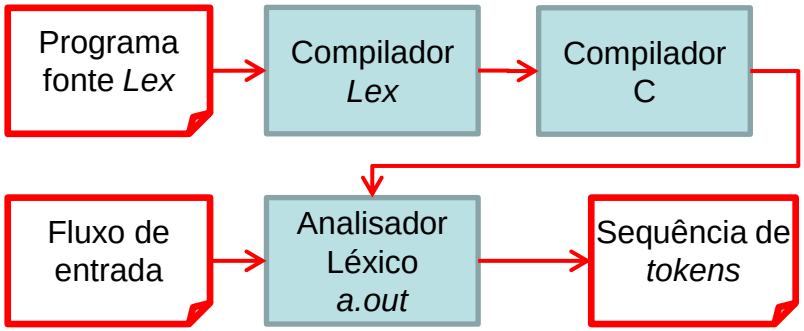
O gerador de analisador léxico – Uso do *Lex*

Um arquivo de entrada, por exemplo *lex.l*, é escrito em *linguagem Lex* e descreve o analisador léxico a ser gerado.

O *compilador Lex* transforma *lex.l* em um programa C, e o armazena em um arquivo que sempre se chama *lex.yy.c*. Esse arquivo é compilado pelo compilador C em um arquivo sempre chamado *a.out*.

A saída do compilador é o analisador léxico gerado, que pode receber como entrada um fluxo de caracteres e produzir como saída um fluxo de *tokens*.

Análise léxica



O gerador de analisador léxico – Uso do *Lex*

Normalmente, usa-se o programa em C compilado, chamado *a.out* como sub-rotina do analisador sintático.

Essa é uma função em C que retorna um inteiro, representando o código de um dos possíveis nomes de *token*.

Já o valor do atributo é colocado em uma variável global *yylval*, que é compartilhada entre o analisador léxico e o analisador sintático, tornando mais simples o retorno de ambos (nome e valor de atributo de um *token*).

Análise léxica

O gerador de analisador léxico – Estrutura de programas *Lex*

Um *programa Lex* possui o seguinte formato:

```
declarações
%%
regras de tradução
%%
funções auxiliares
```

Logo, a seção de declarações inclui a declaração de:

- variáveis,
- *constantes manifestas* (identificadores declarados para significar uma constante, por exemplo, o nome de um *token*)
- e definições regulares.

Exemplo:

*letter_*	→	A | B | ... | Z | a | b | ... | z | _
digit	→	0 | 1 | ... | 9
id	→	*letter_* (*letter_* | *_digit*) *

Análise léxica

declarações
%%
regras de tradução
%%
funções auxiliares

O gerador de analisador léxico – Estrutura de programas *Lex*

Cada regra de tradução possui o formato: Padrão { Ação }

Cada padrão é uma expressão regular que pode usar definições regulares da seção de declaração.

As ações são fragmentos de código, normalmente escritos em C.

A terceira seção contém quaisquer funções auxiliares usadas nas ações.

Tais funções podem ser compiladas separadamente e carregadas com o analisador léxico.

Análise léxica

declarações
%%
regras de tradução
%%
funções auxiliares

O gerador de analisador léxico – Estrutura de programas *Lex*

Quando chamado pelo analisador sintático, o analisador léxico criado pelo *Lex* começa a ler sua entrada restante, um caractere de cada vez, até encontrar o maior prefixo da entrada que case com os padrões P_i.

Depois ele executa a ação associativa A_i.

Normalmente, A_i retorna ao analisador sintático, mas, se não retornar (por ex., porque P_i descreve um espaço em branco ou comentários), então o analisador léxico prossegue a leitura para encontrar lexema adicionais, até que uma das ações correspondentes cause um retorno ao analisador sintático.

Análise léxica

declarações
%%
regras de tradução
%%
funções auxiliares

O gerador de analisador léxico – Estrutura de programas *Lex*

O analisador léxico retorna um único valor (nome do *token*) ao analisador sintático, mas usa uma variável compartilhada inteira "*yylval*" para passar informações adicionais sobre o lexema encontrado, se necessário.

A seguir é apresentado um *programa Lex* que reconhece e retorna os *tokens* da tabela:

Lexemas	*Token*	Atributo
Q. ws	-	-
if	if	-
then	then	-
else	else	-
Q. i*d*	id	Apontador p/ tabela
Q. number	number	Apontador p/ tabela
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Análise léxica

Lexemas	*Token*	Atributo
Q. ws	-	-
if	if	-
then	then	-
else	else	-
Q. *id*	id	Tabela
Q. number	number	tabela

Lexemas	*Token*	Atributo
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

```
%{
/* definições de constantes manifestas
      LT, LE, EQ, NE, GT, GE
      IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
      /* definições regulares */
      delim     [ \t\n]
      ws        {delim}+
      letter    [A-Za-z]
      digit     [0-9]
      id        {letter} ({letter}|{digit})*
      number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
      ...
```

Análise léxica

Lexemas	*Token*	Atributo
Q. ws	-	-
if	if	-
then	then	-
else	else	-
Q. *id*	id	Tabela
Q. number	number	tabela

Lexemas	*Token*	Atributo
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

```
%{
/* definições de constantes manifestas
      LT, LE, EQ, NE, GT, GE
      IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
      /* definições regulares */
      delim     [ \t\n]
      ws        {delim}+
      letter    [A-Za-z]
      digit     [0-9]
      id        {letter} ({letter}|{digit})*
      number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
      ...
```

Qualquer dado dentro dos delimitadores %{ e %} é copiado diretamente para o arquivo *lex.yy.c* e não é tratado como uma definição regular.

É comum colocar as definições das constantes manifestas lá, usando comandos #define da linguagem C para associar um único código inteiro a cada uma das constantes manifestas.

Análise léxica

Lexemas	*Token*	Atributo
Q. ws	-	-
if	if	-
then	then	-
else	else	-
Q. *id*	id	Tabela
Q. number	number	tabela

Lexemas	*Token*	Atributo
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

```
%{
/* definições de constantes manifestas
      LT, LE, EQ, NE, GT, GE
      IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
      /* definições regulares */
      delim     [ \t\n]
      ws        {delim}+
      letter    [A-Za-z]
      digit     [0-9]
      id        {letter} ({letter}|{digit})*
      number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
      ...
```

Na seção de declarações está a seq. de definições regulares.

As definições regulares usadas em outras definições ou padrões das regras de transição aparecem entre chaves “{ }”.

Ex., *delim* é definido como abreviação para a classe de caracteres consistindo em espaço, tabulação (\t) e quebra de linha (\n).

Análise léxica

Lexemas	*Token*	Atributo
Q. ws	-	-
if	if	-
then	then	-
else	else	-
Q. *id*	id	Tabela
Q. number	number	tabela

Lexemas	*Token*	Atributo
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

```
%{
/* definições de constantes manifestas
      LT, LE, EQ, NE, GT, GE
      IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
      /* definições regulares */
      delim     [ \t\n]
      ws        {delim}+
      letter    [A-Za-z]
      digit     [0-9]
      id        {letter} ({letter}|{digit})*
      number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
      ...
```

Logo, *ws* é definido como sendo um ou mais delimitadores pra a expressão regular *{delim}+*.

Na definição de *id* e *number*, os parênteses "()"são usados como metassímbolos de agrupamento e não representam a si mesmos.

Entretanto, E da definição *number*, representa a si mesmo.

Para representar a si mesmo, pode-se usar "\" barra invertida. Ex: .\

Análise léxica

Lexemas	*Token*	Atributo
Q. ws	-	-
if	if	-
then	then	-
else	else	-
Q. *id*	id	Tabela
Q. number	number	tabela

Lexemas	*Token*	Atributo
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

```
        ...
{ws}  {/* nenhuma ação e retorno */}
        if              {return(IF);}
        then            {return(THEN);}
        else            {return(ELSE);}
        {id}            {yylval = (int) installID(); return(ID);}
        {number}        {yylval = (int) installNum(); return(NUMBER);}
        "<"             {yylval = LT; return(RELOP);}
        "<="            {yylval = LE; return(RELOP);}
        "="             {yylval = EQ; return(RELOP);}
        "<>"            {yylval = NE; return(RELOP);}
        ">"             {yylval = GT; return(RELOP);}
        ">="            {yylval = GE; return(RELOP);}
%%
        ...
```

Análise léxica

Lexemas	*Token*	Atributo
Q. ws	-	-
if	if	-
then	then	-
else	else	-
Q. *id*	id	Tabela
Q. number	number	tabela

Lexemas	*Token*	Atributo
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

```
...
{ws} {/* nenhuma ação e retorno */}
    if              {return(IF);}
    then            {return(THEN);}
    else            {return(ELSE);}
    {id}            {yylval = (int) installID(); return(ID);}
    {number}        {yylval = (int) installNum(); return(NUMBER);}
    "<"             {yylval = LT; return(RELOP);}
    "<="            {yylval = LE; return(RELOP);}
    "="             {yylval = EQ; return(RELOP);}
    "<>"            {yylval = NE; return(RELOP);}
    ">"             {yylval = GT; return(RELOP);}
    ">="            {yylval = GE; return(RELOP);}
%%
    ...
```

Padrões e regras da seção

ws é um identificador com ação vazia associada.

Se for encontrado um espaço em branco, não retorna-se ao analisador sintático, mas procura-se outro lexema.

Análise léxica

Lexemas	*Token*	Atributo
Q. ws	-	-
if	if	-
then	then	-
else	else	-
Q. *id*	id	Tabela
Q. number	number	tabela

Lexemas	*Token*	Atributo
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

```
...
{ws} {/* nenhuma ação e retorno */}
    if          {return(IF);}
    then        {return(THEN);}
    else        {return(ELSE);}
    {id}        {yylval = (int) installID(); return(ID);}
    {number}    {yylval = (int) installNum(); return(NUMBER);}
    "<"         {yylval = LT; return(RELOP);}
    "<="        {yylval = LE; return(RELOP);}
    "="         {yylval = EQ; return(RELOP);}
    "<>"        {yylval = NE; return(RELOP);}
    ">"         {yylval = GT; return(RELOP);}
    ">="        {yylval = GE; return(RELOP);}
%%
...
```

O segundo *token* tem expressão regular simples *if*.

O analisador léxico consome essas duas letras da entrada e retorna o nome do *token* IF, ou seja, o inteiro que a constante manifesta IF representa.

Tratamento semelhante para as palavras-chave *then* e *else*.

Análise léxica

Lexemas	*Token*	Atributo
Q. ws	-	-
if	if	-
then	then	-
else	else	-
Q. *id*	id	Tabela
Q. number	number	tabela

Lexemas	*Token*	Atributo
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

```
...
{ws} {/* nenhuma ação e retorno */}
    if              {return(IF);}
    then            {return(THEN);}
    else            {return(ELSE);}
    {id}            {yylval = (int) installID(); return(ID);}
    {number}        {yylval = (int) installNum(); return(NUMBER);}
    "<"             {yylval = LT; return(RELOP);}
    "<="            {yylval = LE; return(RELOP);}
    "="             {yylval = EQ; return(RELOP);}
    "<>"            {yylval = NE; return(RELOP);}
    ">"             {yylval = GT; return(RELOP);}
    ">="            {yylval = GE; return(RELOP);}
%%
    ...
```

O token que tem o padrão definido por id é mais complexo.

Análise léxica

declarações
%%
regras de tradução
%%
funções auxiliares

```
{id}            {yylval = (int) installID(); return(ID);}
```

Embora as palavras-chave como *if* casem com esse padrão e com o anterior (*token* com padrão definido por *if*), o *Lex* escolhe qualquer padrão que esteja listado 1º em situações nas quais o prefixo mais longo casa com dois ou mais padrões.

A ação tomada é tripla:

1. *installID()* é chamada p/ colocar o lexema encontrado na tab. de símbolos.
2. Essa função retorna um apontador para a tabela de símbolos, que é colocado na variável global *yylval* para ser usado pelo analisador sintático ou outro componente do compilador. Duas variáveis estão disponíveis pelo AL: *yytext* (apontador para início do lexema) e *yyleng* (tamanho do lexema encontrado).
3. O nome do *token* ID é retornado ao analisador sintático.

Análise léxica

Lexemas	*Token*	Atributo
Q. ws	-	-
if	if	-
then	then	-
else	else	-
Q. *id*	id	Tabela
Q. number	number	tabela

Lexemas	*Token*	Atributo
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

```
...
{ws} {/* nenhuma ação e retorno */}
    if              {return(IF);}
    then            {return(THEN);}
    else            {return(ELSE);}
    {id}            {yylval = (int) installID(); return(ID);}
    {number}        {yylval = (int) installNum(); return(NUMBER);}
    "<"             {yylval = LT; return(RELOP);}
    "<="            {yylval = LE; return(RELOP);}
    "="             {yylval = EQ; return(RELOP);}
    "<>"            {yylval = NE; return(RELOP);}
    ">"             {yylval = GT; return(RELOP);}
    ">="            {yylval = GE; return(RELOP);}
%%
    ...
```

O *token* que tem o padrão os operadores de comparação (*relop*), são semelhantes as palavras-chave, entretanto, também retornam o atributo para a variável global *yylval*.

Análise léxica

Lexemas	*Token*	Atributo
Q. ws	-	-
if	if	-
then	then	-
else	else	-
Q. *id*	id	Tabela
Q. number	number	tabela

Lexemas	*Token*	Atributo
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

...

```
int installID() {/* função para instalar o lexema, cujo primeiro caractere é apontado por yytext, e cujo tamanho é yyleng, na tab. símbolos, e retorna um apontador para lá */}

int installNum() {/* semelhante a installID(), mas coloca constantes numéricas em uma tabela separada */}
```

Seção de função auxiliar: pode-se ver duas dessas funções, installID() e installNum().

Assim como parte da seção de declaração que aparece entre %{...%}, tudo na seção auxiliar é copiado diretamente para o arquivo *lex.yy.c*, mas pode ser usado nas ações.

Análise léxica

O gerador de analisador léxico – Solução de conflito no *Lex*

Problema: vários prefixos da entrada casam com um ou mais padrões.

Solução: Duas regras costumam ser usadas para decidir sobre a seleção do lexema mais apropriado:

1. Sempre prefira um prefixo mais longo a um prefixo mais curto.

... continuar lendo letras e dígitos a fim de encontrar o prefixo mais longo desses caracteres e agrupá-lo como um identificador. Também deve tratar <= como um único lexema, ao invés de selecionar < como um lexema e = como lexema seguinte.

Análise léxica

O gerador de analisador léxico – Solução de conflito no *Lex*

2. Se for possível casar o prefixo mais longo com dois ou mais padrões, prefira o padrão listado primeiro no *programa Lex*;

... torna as palavras-chave reservadas, se forem listadas antes de *id* no programa.

Por exemplo: se *then* for determinado para ser o prefixo mais longo da entrada que casa com qualquer padrão, e o padrão *then* preceder {*id*}, então o *token* THEN é retornado, em vez de ID.

Análise léxica

O gerador de analisador léxico – O operador *lookahead*

O *Lex* lê de forma automática um caractere após o último caractere que forma o lexema selecionado, e depois recua a entrada de modo que só o próprio lexema seja consumido.

Porém, as vezes, queremos que certo padrão seja casado com a entrada apenas se for seguido por certos outros caracteres.

Neste caso, podemos usar a barra em um padrão para indicar o fim da parte do padrão que casa com o lexema. O que aparece após a "/" é um padrão adicional, que precisa ser casado antes de podermos decidir que o *token* em questão foi visto, mas o que casa com esse segundo padrão não faz parte do lexema.

Análise léxica

O gerador de analisador léxico – O operador *lookahead*

Exemplo: em Fortran e em algumas outras linguagens, as palavras-chave não são reservadas. Essa situação cria um problema como no comando:

IF(I,J) = 3, onde IF é o nome de um arranjo, e não uma palavra-chave.

Esse comando difere de comando na forma:

IF (*condition*) THEN ..., onde IF é uma palavra-chave.

Felizmente, podemos estar certos de que a palavra-chave IF sempre é seguida por um parêntese esquerdo, algum texto – *condition* – que pode conter parênteses, um parêntese direito e uma letra.

Análise léxica

O gerador de analisador léxico – O operador *lookahead*

Assim, poderíamos escrever uma regra do *Lex* para a palavra-chave IF da seguinte forma: IF / \(.* \) {*letter*}

Essa regra diz que o padrão que o lexema casa são apenas duas letras IF. Já a barra diz que o padrão adicional vem em seguida, mas não casa com o lexema.

Neste padrão, o primeiro caractere é o parêntese esquerdo "(". Como esse caractere é um metassímbolo do *Lex*, ele precisa ser precedido por uma barra invertida "\" para indicar que possui seu significado literal.

Análise léxica

O gerador de analisador léxico – O operador *lookahead*

Continuação... IF / \(.* \) {*letter*}

O ponto e asterisco “.*” casam com “qualquer cadeia de caracteres sem uma quebra de linha”. O ponto “.” é um metassímbolo do *Lex,* significando “qualquer caractere exceto quebra de linha”.

Ele é seguido por um parêntese direito “\)” com uma barra invertida para dar seu significado literal.

O padrão adicional é seguido pelo símbolo *letter*, que é uma definição regular representando a classe de caracteres de todas as letras.

Análise léxica

O gerador de analisador léxico – O operador *lookahead*

Continuação... IF / \(.* \) {*letter*}

Para que esse padrão seja infalível, deve-se pré-processar a entrada para remover o espaço em branco.

No padrão não temos provisão p/ espaços em branco nem podemos lidar com a possibilidade de que a condição se estenda por várias linhas, pois o ponto não casa com qualquer quebra de linha.

Ex.: Pode ocorrer desse padrão ser solicitado para casar com um prefixo de entrada: IF (A<(B+C) *D) THEN ...

Análise léxica

O gerador de analisador léxico – O operador *lookahead*

Continuação... IF (A<(B+C) *D) THEN ... <<>> IF / \(.* \) {*letter*}

Os dois primeiros caracteres casam com "IF", o próximo casa com "\(", os nove caracteres seguintes casam com ".*", e os dois seguintes casam com "\)" e "*letter*".

Pode-se observar que o fato do primeiro parêntese do lado direito (depois de C) não ser seguido por uma letra é irrelevante. Apenas deve-se achar uma maneira de casar a entrada com o padrão.

Pode-se concluir que as letras IF constituem o lexema, e elas são uma instância do *token* IF.

Análise léxica

Gerador de analisadores léxicos

Flex – versão GNU do lex para C.
http://www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html

Jlex – versão Java com pequena diferença na sintaxe de entrada.
http://www.cs.princeton.edu/~appel/modern/java/JLex/

CSLex – versão C#, derivada do Jlex.
http://www.cybercom.net/~zbrad/DotNet/Lex

Análise léxica

Autômatos finitos

O *Lex* usa os autômatos finitos para transformar seu programa de entrada em um analisador léxico.

Os autômatos finitos são apenas reconhecedores, ou seja, dizem “sim” ou “não” para uma cadeia de caracteres de entrada.

Podem ser de dois tipos: Autômatos Finitos Não Determinísticos (AFND) ou Autômatos Finitos Determinísticos (AFD).

Os AFND e AFD são capazes de reconhecer as mesmas linguagens, inclusive, um AFND pode ser transformado em um AFD. Além disso, também pode ser representados em Tabelas de transição.

Análise léxica

Autômatos finitos

Autômatos finitos não determinísticos: não têm restrições sobre rótulos de suas arestas. Um símbolo pode rotular várias arestas saindo do mesmo estado, e ϵ, a cadeia vazia é um rótulo possível.

Autômatos finitos determinísticos: Possuem para cada estado e para cada símbolo de seu alfabeto de entrada, exatamente uma aresta com esse símbolo saindo desse estado.

Aceitação das cadeias de entrada pelo autômatos: um AFD aceita a cadeia de entrada x se e somente se houver algum caminho no grafo de transição do estado inicial para um dos estados finais, de modo que os símbolos ao longo do caminho componha x.

Análise léxica

Avaliação:

Disponível no EAD, a partir das 20:30h.

8 questões para serem respondidas.

3 tentativas.

Nota é igual a média tirada nas três tentativas (ou menos)

Intervalo entre as tentativas: 24h.

Boa sorte!!

Análise léxica

Prova sobre análise léxica

09/09/2013: JAC - 19:00h, chamada em sala de aula

Segunda-feira	09 de setembro
19:00h	Recepção
19:15h	Abertura da VII JAC
19:45h	**e-Saúde: áreas de atuação e perspectivas** Alessandra Dahmer Fundação Universidade Federal de Ciências da Saúde de Porto Alegre
	Segurança de Redes Virtuais: Um Passo rumo à Internet do Futuro Luciano Gaspary Instituto de Informática - UFRGS
	Objetos Digitais de Aprendizagem Juliano Tonezer da Silva Universidade de Passo Fundo – UPF
	Redes SDN e seu gerenciamento Lisandro Zambenedetti Granville Instituto de Informática – UFRGS
	Identificação eletrônica: princípios e aplicações Rafael Ramos dos Santos CEITEC

Análise léxica

Prova sobre análise léxica

16/09/2013

Laboratório 16.

Início da prova: 19:10h e término: 22:10h.

Prova individual e sem consulta.

COMPILADORES

Obrigado!!

Prof. Geovane Griesang
geovanegriesang@unisc.br