Teoria da Computação e Compiladores

Análise Léxica

Prof. Dr. Fernando Kakugawa

fernando.kakugawa@animaeducacao.com.br

Universidade Anhembi Morumbi

Análise Léxica

- Papel do analisador léxico:
  - ler o arquivo fonte em busca de unidades significativas (tokens) instanciadas por lexemas ou átomos
- Também denominado de scanner, porque varre o arquivo de entrada eliminando comentários e caracteres indesejáveis ao agrupar caracteres com um papel bem definido

Análise Léxica

- A separação da análise léxica da análise sintática facilita o projeto e torna o compilador mais eficiente e portável
- Um analisador léxico executa tarefas como:
  1. contar as linhas de um programa
  2. eliminar comentários
  3. contar a quantidade de caracteres de um arquivo
  4. tratar espaços

Análise Léxica

- Tokens – são padrões de caracteres com um significado específico em um código fonte
  - Definida por um alfabeto e um conjunto de definições regulares
- Lexemas – são ocorrências de um *token* em um código fonte, também são chamados de átomos por alguns autores

Análise Léxica

- Um mesmo *token* pode ser produzido por várias cadeias de entradas
- Tal conjunto de cadeias é descrito por uma regra denominada padrão, associada a tais *tokens*
- O padrão reconhece as cadeias de tal conjunto, ou seja, reconhece os *lexemas* que são padrão de um *token*

Análise Léxica

- Usualmente os padrões são convenções determinadas pela linguagem para formação de classes de *tokens*
  - identificadores: letra seguida por letras ou dígitos
  - literal: cadeias de caracteres delimitadas por aspas
  - num: qualquer constante numérica

Análise Léxica

- Os *tokens* usualmente são conhecidos pelo seu *lexema* (sequência de caracteres que compõe um único *token*) e atributos adicionais
- Os *tokens* podem ser entregues ao *parser* como tuplas na forma <a, b, ..., n> assim a entrada:
  - a = b + 3
- poderia gerar as tuplas:
  - <**id**,a> <=, > <**id**,b> <**op_arit**,+ > <**num**,3>
  - note que alguns *tokens* não necessitam atributos adicionais

Análise Léxica

- A declaração C seguinte;
  - `int k = 123;`
- Possui várias subcadeias:
  - **`int`** é o *lexema* para um *token* tipo palavra-reservada.
  - **`k`** é o lexema para um *token* tipo identificador
  - **`=`** é o lexema para um *token* tipo operador de atribuição
  - **`123`** é o lexema para um *token* tipo número literal cujo atributo valor é 123
  - **`;`** é o lexema para um *token* tipo pontuação

Análise Léxica

- Quais os tokens que podem ser reconhecidos em uma linguagem de programação como C ?

palavras reservadas identificadores	`if else while do`
operadores relacionais	`< > <= >= == !=`
operadores aritméticos	`+ * / -`
operadores lógicos	`&& || & | !`
operador de atribuição	`=`
delimitadores	`; ,`
caracteres especiais	`( ) [ ] { }`

Análise Léxica

- Quais os *tokens* que podem ser reconhecidos em uma linguagem de marcação como HTML ?

Tags	<html> <body> <table>...
Comentários	<!-- ... -->
Conteúdos	Página de teste…
Especiais	é

Análise Léxica

- Lexemas podem ter atributos como número da linha em que se encontra no código fonte e o valor de uma constante numérica ou um literal
- Normalmente utiliza-se um único atributo que é um apontador para a Tabela de Símbolos que armazena essas informações em registros

Análise Léxica

- O analisador léxico simplesmente varre a entrada (arquivo fonte) em busca de *padrões pertencentes a uma linguagem*.
- Possibilidade de ocorrer erro:
  - aparecer um caractere que não pertence ao alfabeto da linguagem
  - ou que não tenha uma expressão regular que o reconheça
- Na ocorrência deste erro, existem duas possibilidades
  - ou o projetista realmente esqueceu de incluir o caractere no alfabeto
  - ou realmente o usuário utilizou algum caractere que não pertence ao alfabeto da linguagem

Especificação de Tokens

- *Tokens* são padrões que podem ser especificados através de expressões regulares
- Um alfabeto determina o conjunto de caracteres válidos para a formação de cadeias, sentenças ou palavras
- Cadeias são sequências finitas de caracteres. Algumas operações podem ser aplicadas a alfabetos para ajudar na definição de cadeias:
  - Concatenação
  - União
  - Fechamento

Especificação de Tokens

- Concatenação

  L.M = {*st* | *s* pertence a L e *t* pertence a M}

- União

  L ∪ M= {*s* | *s* pertence a L ou a M}

- Fechamento

  L^*= ∪ L_i, i=0 ...

- Fechamento Positivo

  L^+= ∪ L_i, i=1 ...

Especificação de Tokens

- Expressões regulares podem receber um nome (definição regular), formando o *token* de um analisador léxico
- Algumas convenções podem facilitar a formação de definições regulares
  1. Uma ou mais ocorrência (+)
  2. Zero ou mais ocorrências (*)
  3. Zero ou uma ocorrência (?)
  4. Classe de caracteres [a-z] = a | b | ... | z

Especificação de Tokens

- São definições regulares

letra → [A-Z]|[a-z]

dígito → [0-9]

dígitos → dígito dígito*

identificador → letra[letra|dígito]*

fração_opc → .dígitos|ε

exp_opc → E[+|-|ε]dígitos| ε

num → dígitos fração_opc exp_opc

delim → ‘ ’ | ’\t’ | ’\n’

Reconhecimento de Tokens

- *Tokens* podem ser reconhecidos através de autômatos finitos onde o estado final dispara o reconhecimento de um *token* específico e/ou um procedimento específico (inserir na tabela de símbolo, por exemplo).

- Normalmente constrói-se um diagrama de transição para representar o reconhecimento de *tokens*.

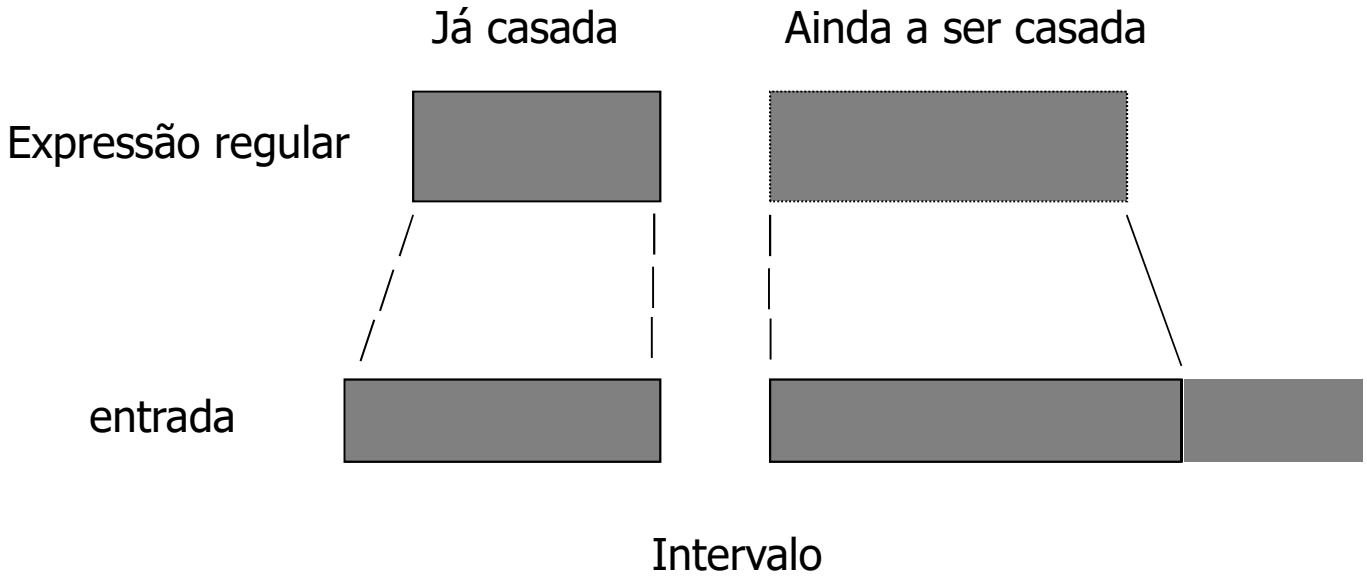
Reconhecimento de Tokens

- Como são reconhecidos os identificadores e as palavras reservadas?
- Como um compilador sabe o que é uma palavra reservada?
- O reconhecimento de identificadores e de palavras reservadas é idêntico
  - A tabela de símbolos é responsável pela identificação das palavras reservadas

Reconhecimento de Tokens

- Em geral, a tabela de símbolos é iniciada com o registro das palavras reservadas da linguagem
- O compilador sempre insere identificadores na tabela de símbolo? Isto é necessário?
  - Não, os identificadores são armazenados apenas uma vez, mas seus atributos podem ser alterados ao longo da análise de um programa
    - `int a;`
    - `a = 10;`

Criação de um Analisador Léxico



- $T \rightarrow \alpha \bullet \beta$

Criação de um Analisador Léxico

- Movendo Caracteres
- Se o ponto estiver à esquerda de um caractere *c* e se a entrada tiver *c* na próxima posição, o item será transportado para o outro lado e o ponto será reposicionado de acordo:
  - $T \rightarrow \alpha \bullet c\ \beta \quad \Rightarrow \quad T \rightarrow \alpha\ c \bullet \beta$

Criação de um Analisador Léxico

- Item de Redução:
  - $T \rightarrow \alpha\beta\bullet$
- O objetivo é atingir um item de redução
  - Durante o processo de reconhecimento, cada símbolo é reconhecido na entrada
- Isto não significa que o *token* correto foi encontrado, pois um *token* mais longo ainda pode surgir

Criação de um Analisador Léxico

- O processo de reconhecimento continua até que não haja mais nenhuma hipótese a ser verificada.
- Assim, o *token* reconhecido mais recentemente é o *token* mais longo.
- Ponto de Partida:

$$R \rightarrow \bullet\alpha$$

Criação de um Analisador Léxico

- Um item no qual o ponto está a esquerda de um padrão controlado por operador tem de ser substituído por um ou mais outros itens que expressem o significado do operador.

$$T \rightarrow \alpha \bullet (R)^* \beta \qquad (1)$$

- Movimentações para o ponto:

$$T \rightarrow \alpha (R)^* \bullet \beta \qquad (2)$$

$$T \rightarrow \alpha (\bullet R)^* \beta \qquad (3)$$

Criação de um Analisador Léxico

- Quando o ponto no item (3) é movido para o fim de R, novamente surge duas novas possibilidades de movimentação do ponto: ou esta foi a última ocorrência de R ou há outra vindo. Portanto, o item

$$T \rightarrow \alpha(R\bullet)^*\beta \qquad (4)$$

- Deve ser substituído pelos itens (2) e (3)

Criação de um Analisador Léxico

- Tipos de Movimentações para os Operadores:
- Operador *

$T \rightarrow \alpha\bullet(R)^*\beta$

$\Rightarrow \quad T \rightarrow \alpha(R)^*\bullet\beta$

$\Rightarrow \quad T \rightarrow \alpha(\bullet R)^*\beta$

$T \rightarrow \alpha(R\bullet)^*\beta$

$\Rightarrow \quad T \rightarrow \alpha(R)^*\bullet\beta$

$\Rightarrow \quad T \rightarrow \alpha(\bullet R)^*\beta$

Criação de um Analisador Léxico

- Operador +

$T \rightarrow \alpha\bullet(R)^{+}\beta$

$\Rightarrow \quad T \rightarrow \alpha(\bullet R)^{+}\beta$

$T \rightarrow \alpha(R\bullet)^{+}\beta$

$\Rightarrow \quad T \rightarrow \alpha(R)^{+}\bullet\beta$

$\Rightarrow \quad T \rightarrow \alpha(\bullet R)^{+}\beta$

Criação de um Analisador Léxico

- Operador ?

$T \rightarrow \alpha \bullet (R)^{?} \beta$

$\Rightarrow \quad T \rightarrow \alpha (R)^{?} \bullet \beta$

$\Rightarrow \quad T \rightarrow \alpha (\bullet R)^{?} \beta$

$T \rightarrow \alpha (R \bullet)^{?} \beta$

$\Rightarrow \quad T \rightarrow \alpha (R)^{?} \bullet \beta$

Criação de um Analisador Léxico

- Operador |

$T \rightarrow \alpha\bullet(R1|R2|...)\beta$

$\Rightarrow T \rightarrow \alpha(\bullet R1|R2|...)\beta$

$\Rightarrow T \rightarrow \alpha(R1|\bullet R2|...)\beta$

...

$T \rightarrow \alpha(R1\bullet|R2|...)\beta$

$\Rightarrow T \rightarrow \alpha(R1|R2|...)\bullet\beta$

$T \rightarrow \alpha(R1|R2\bullet|...)\beta$

$\Rightarrow T \rightarrow \alpha(R1|R2|...)\bullet\beta$

...

Exemplo

- Analisador Léxico para os símbolos de número inteiro e de ponto fixo

  numero_inteiro $\rightarrow$ [0-9]$^{+}$

  numero_ponto_fixo $\rightarrow$ [0-9]*'.'[0-9]$^{+}$

- Tentativa de reconhecer a entrada **3.1;** usando a expressão regular:

  numero_ponto_fixo $\rightarrow$ ([0-9])*'.'([0-9])$^{+}$

Exemplo

numero_ponto_fixo → ●([0-9])*'.'([0-9])$^{+}$

- O movimento produz dois itens:
  - numero_ponto_fixo → (●[0-9])*'.'([0-9])$^{+}$ **(1)**
  - numero_ponto_fixo → ([0-9])*●'.'([0-9])$^{+}$ **(2)**
- O **(1)** item pode ser movido sobre o **3**, resultando em:
  - numero_ponto_fixo → ([0-9]●)*'.'([0-9])$^{+}$
- Mas o **(2)** item é descartado

Exemplo

numero_ponto_fixo → ([0-9]●)*'.'([0-9])$^{+}$

- O novo item se desenvolve em:
  - numero_ponto_fixo → (●[0-9])*'.'([0-9])$^{+}$
  - numero_ponto_fixo → ([0-9])*●'.'([0-9])$^{+}$
- O movimento deste conjunto sobre o caractere '.' deixa apenas um item:
  - numero_ponto_fixo → ([0-9])*'.'●([0-9])$^{+}$
- Que se desenvolve em:
  - numero_ponto_fixo → ([0-9])*'.' (●[0-9])$^{+}$

Exemplo

numero_ponto_fixo → ([0-9])*'.' (●[0-9])$^{+}$

- Este item pode ser movido sobre o **1**, o que resulta em:
  - numero_ponto_fixo → ([0-9])*'.' ([0-9]●)$^{+}$
- Por sua vez se desenvolve em:
  - numero_ponto_fixo → ([0-9])*'.' (●[0-9])$^{+}$
  - numero_ponto_fixo → ([0-9])*'.' ([0-9])$^{+}$● **<<< reconhecido**
- Observa-se que o último item é um item de redução, assim reconhecemos um símbolo e a sua classe

Exemplo

- A classe do símbolo e o ponto final são registrados e o algoritmo continua na captura por uma sequência mais longa
- Contudo, nenhum item pode ser movido sobre o **;** que segue 3.1 na entrada
  - E o processo é interrompido.
- Deste modo, retorna-se o *token* com a classe e a representação
  - E a posição de entrada é movida para a localização do **;**

Qual análise realizar?

- Analisador Léxico para os símbolos de número inteiro e de ponto fixo

numero_inteiro $\rightarrow$ [0-9]$^{+}$

numero_ponto_fixo $\rightarrow$ [0-9]*'.'[0-9]$^{+}$

3.1;

Pesquisa Concorrente

- O algoritmo anterior procurava apenas por uma classe de *tokens*
  - Para modificá-lo para procurar por todas as classes de *tokens* basta inserir todos os itens iniciais no conjunto de itens iniciais.
- Processe a entrada **3.1;** com todos os símbolos

Autômatos na Análise Léxica

- O conjunto de itens considerado pelo analisador léxico em um dado momento é chamado de seu estado
- O ponto de partida é o seu estado inicial
- As transições de estado são feitas através do próximo símbolo a ser lido
  - Esta transição é chamada de função de transição
- Com isso é possível elaborar um autômato para realizar a análise léxica

Autômatos na Análise Léxica

- Para cada caractere *Ch* no conjunto de caracteres, calcula-se a sua transição para os estados do autômato
- O processo é repetido até que todos os estados acessíveis sejam gerados

Exemplo: Autômatos

- Para as descrições de número_inteiro e número_ponto_fixo anteriores podemos ter como estado inicial:
  - numero_inteiro $\rightarrow$ (●[0-9])$^{+}$
  - numero_ponto_fixo $\rightarrow$ (●[0-9])*'.'([0-9])$^{+}$
  - numero_ponto_fixo $\rightarrow$ ([0-9])*●'.'([0-9])$^{+}$
- Chamaremos este estado de S_0.

Exemplo: Autômatos

- Consideraremos apenas três classes de caracteres:
  - Dígitos
  - Pontos decimais
  - Outros:
    - sinais de ponto-e-vírgula, parênteses, etc
- Calcularemos as transições de S_0 com estas três classes

Exemplo: Autômatos

- Transição (S_0 , dígito)
  - numero_inteiro $\rightarrow$ (●[0-9])$^+$
  - numero_inteiro $\rightarrow$ ([0-9])$^+$● **<<< reconhecido**
  - numero_ponto_fixo $\rightarrow$ (●[0-9])*'.'([0-9])$^+$
  - numero_ponto_fixo $\rightarrow$ ([0-9])*●'.'([0-9])$^+$
- Chamaremos este estado de S_1.
- (S_0 , dígito , S_1)

Exemplo: Autômatos

- Transição (S_0 , .)
  - numero_ponto_fixo → ([0-9])*'.'(●[0-9])$^+$
- Produzindo o estado S_2. (S_0 , . , S_2)

- Transição (S_0 , outros)
- Produzindo o estado S_L. (estado de lixo) (S_0 , outros , S_L)

Exemplo: Autômatos

- Transição (S_1 , dígito)
  - numero_inteiro → (●[0-9])$^+$
  - numero_inteiro → ([0-9])$^+$● **<<< reconhecido**
  - numero_ponto_fixo → (●[0-9])*'.'([0-9])$^+$
  - numero_ponto_fixo → ([0-9])*●'.'([0-9])$^+$
- Que é o próprio estado S_1

Exemplo: Autômatos

- Transição (S_1 , .)
  - numero_ponto_fixo $\rightarrow$ ([0-9])*'.'(●[0-9])$^+$
- Produzindo o estado S_2. (S_1 , . , S_2)
- Transição (S_1 , outros)
- Produzindo o estado S_L. (estado de lixo) (S_1 , outros , S_L)

Exemplo: Autômatos

- Transição (S_2 , dígito)
  - numero_ponto_fixo → ([0-9])*'.'(●[0-9])$^+$
  - numero_ponto_fixo → ([0-9])*'.'([0-9])$^+$●

<< reconhecido

- Produzindo o estado S_3. (S_2 , dígito , S_3)
- Transição (S_2 , .) e (S_2 , outros)
- Produzindo o estado S_L. (estado de lixo) (S_2 , . , S_L) (S_2 , outros , S_L)

Exemplo: Autômatos

- Transição (S_3 , dígito)
  - numero_ponto_fixo → ([0-9])*'.'(●[0-9])$^+$
  - numero_ponto_fixo → ([0-9])*'.'([0-9])$^+$●

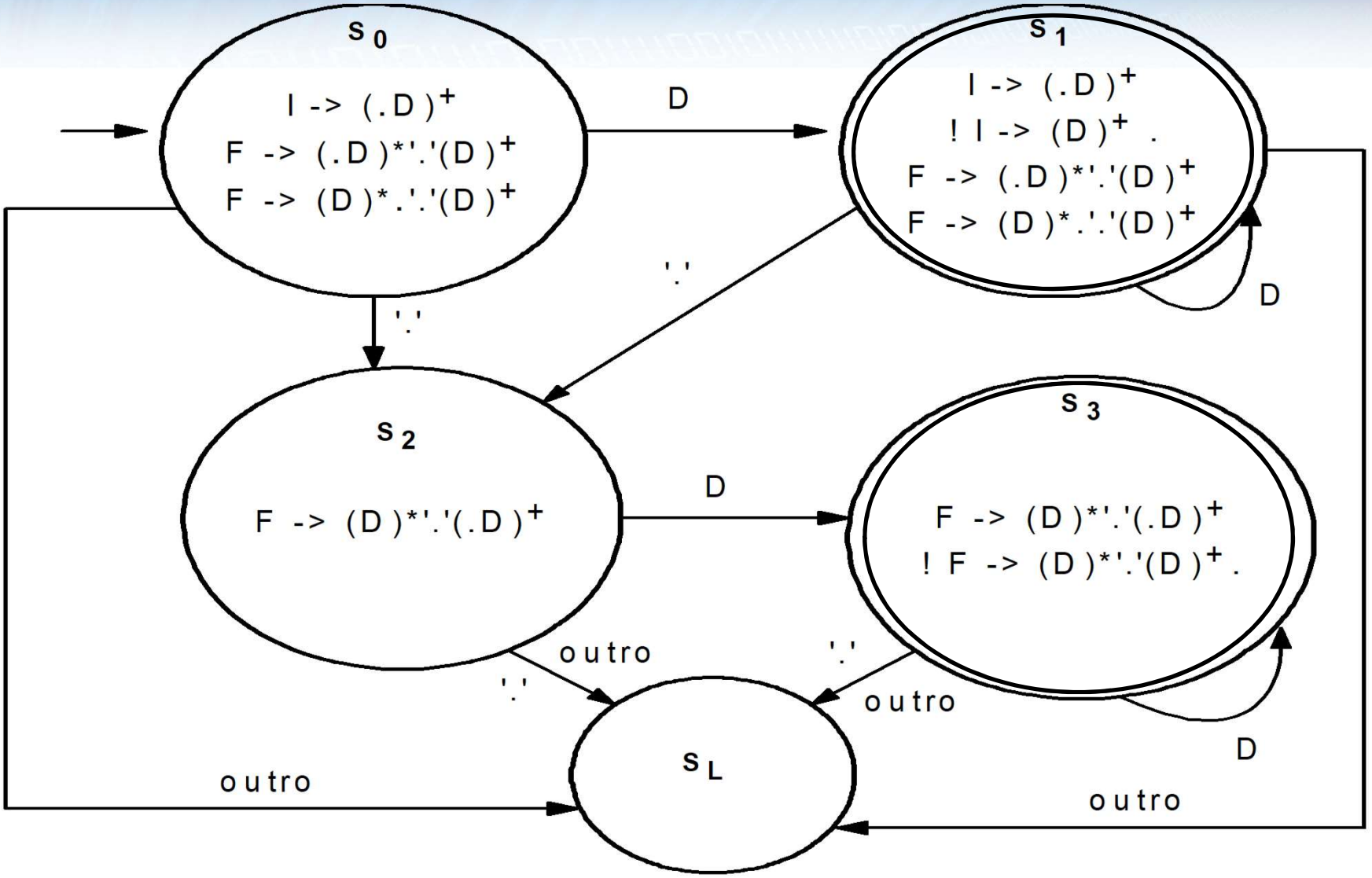
<< reconhecido

- Produzindo o estado S_3. (S_3 , dígito , S_3)
- Transição (S_3 , .) e (S_3 , outros)
- Produzindo o estado S_L. (estado de lixo) (S_3 , . , S_L) (S_3 , outros , S_L)

Exemplo: Autômatos

	Novo estado			Token Reconhecido
Estado	Ch			
	dígito	ponto	outros	
S_0	S_1	S_2	S_L	
* S_1	S_1	S_2	S_L	**inteiro**
S_2	S_3	S_L	S_L	
* S_3	S_3	S_L	S_L	**ponto_fixo**
S_L	S_L	S_L	S_L	

Exemplo: Autômatos



Projeto de Analisador Léxico

1. Definir o alfabeto
2. Listar os *tokens* necessários
3. Especificar os *tokens* por meio de definições regulares
4. Montar os autômatos para reconhecer os *tokens*
5. Implementar o analisador léxico

Projeto de Analisador Léxico

- EXERCÍCIO
  - Projetar um analisador léxico para uma calculadora simples com números naturais e reais, operações básicas de soma e subtração, e parênteses

Enfoques de Implementação

- Existem 3 enfoques básicos para construção de um analisador léxico:
  1. Utilizar um gerador automático de analisadores léxicos (tal como o compilador LEX, que gera um analisador a partir de uma especificação)
  2. Escrever um analisador léxico usando uma linguagem de programação convencional que disponha de certas facilidades de E/S
  3. Escrever um analisador léxico usando linguagem de montagem

Enfoques de Implementação

- As alternativas de enfoque estão listadas em ordem crescente de complexidade e (infelizmente) de eficiência
- A construção via geradores é praticamente adequada quando o problema não esbarra em questões de eficiência e flexibilidade
- A construção manual é uma alternativa atraente quando a linguagem a ser tratada não for por demais complexa

Projeto de Analisador Léxico

- Exemplo - seja a cadeia 3.2 + (2 - 12.01), o analisador léxico teria como saída:

3.2 => número real

\+ => operador de soma

(=> abre parênteses

2 => número natural

\- => operador de subtração

12.01 => número real

) => fecha parênteses

Projeto de Analisador Léxico

- Qual símbolo usar como separador de casa decimais?
- A calculadora usa representação monetária?
- A calculadora aceita espaços entre os operandos e operadores?
- O projetista é quem decide sobre as características desejáveis do compilador ou interpretador
  - Para a maioria das linguagens de programação existem algumas convenções que devem ser respeitadas

1. Definição do Alfabeto

- ∑= {0,1,2,3,4,5,6,7,8,9,.,(,),+,-}

- OBS.: projetista deve considerar TODOS os símbolos que são necessários para formar os padrões

2. Listagem dos tokens

`OPSOMA:` operador de soma

`OPSUB`: operador de subtração

`AP`: abre parênteses

`FP`: fecha parênteses

`NUM`: número natural/real

- OBS.: projetista deve considerar *tokens* especiais e cuidar para que cada *token* seja uma unidade significativa para o problema

3. Especificação dos *tokens* com definições regulares

OPSOMA $\rightarrow$ +

OPSUB $\rightarrow$ -

AP $\rightarrow$ (

FP $\rightarrow$)

NUM $\rightarrow$ [0-9]$^{+}$.$^{?}$ [0-9]*

- OBS.: cuidar para que as definições regulares reconheçam padrões claros, bem formados e definidos

4. Montar os autômatos para reconhecer cada *token*

- OBS.: os autômatos reconhecem *tokens* individuais, mas é o conjunto dos autômatos em um único autômato não-determinístico que determina o analisador léxico de um compilador, por isto, deve ser utilizada uma numeração crescente para os estados

5. Implementar o analisador léxico

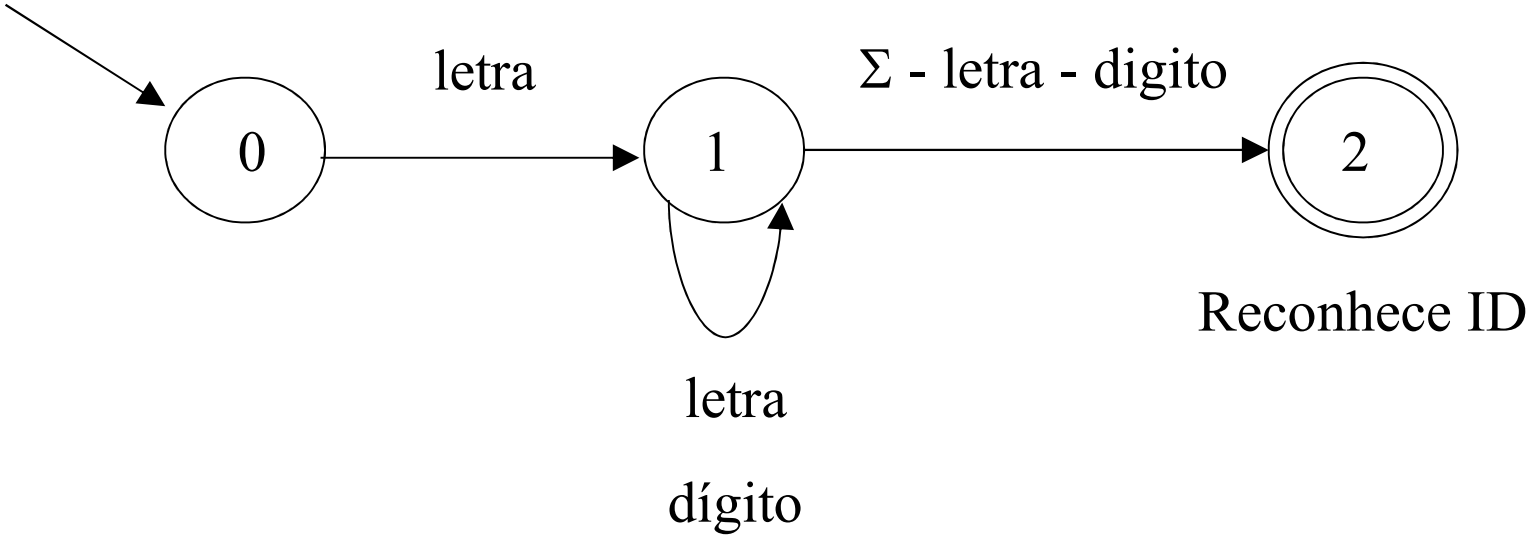
- A forma comum para implementar o autômato é através de uma tabela de transição

Estilo de Implementação do Léxico

- Cada *token* listado é codificado em um número natural
- Deve haver uma variável para controlar o estado corrente do autômato e outro para indicar o estado de partida do autômato em uso
- Uma função `falhar` é usada para desviar o estado corrente para um outro autômato no caso de um estado não reconhecer uma letra

Estilo de Implementação do Léxico

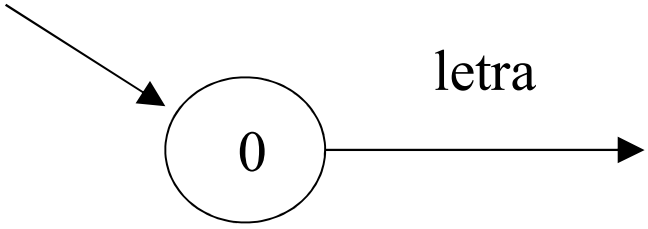
- Cada estado é analisado individualmente em uma estrutura do tipo switch…case



Estilo de Implementação do Léxico

- Cada estado é analisado individualmente em uma estrutura do tipo switch…case

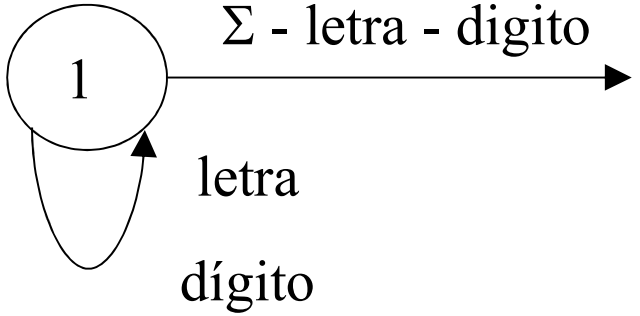
```
int lexico()
{
  while (1)
  {
    switch (estado)
    {
      case 0: c= proximo_caracter();
        if (isalpha(c))
        {
          estado= 1;
          adiante++;
        }
        else
        {
          falhar();
        }
        break;
         …
    }
}
```



Estilo de Implementação do Léxico

- Cada estado é analisado individualmente em uma estrutura do tipo switch…case

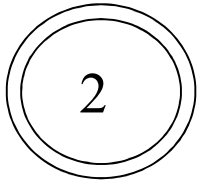
```
…
case 1: c= proximo_caracter();
  if (isalpha(c) || isdigit(c))
  {
    estado= 1;
    adiante++;
  }
  else
  {
    if ((c == '\n') || (c == '\t') || (c == '\b'))
      estado= 2;
    else
      falhar();
  }
  break;

 …
  }
}
```



Estilo de Implementação do Léxico

- Cada estado é analisado individualmente em uma estrutura do tipo switch…case

```
    …
    case 2: estado= 0;
      partida= 0;
      return ID;
      break;
  }
}
```



Reconhece ID

Material elaborado por:

Prof. Dr. Augusto Mendes Gomes Jr.

augusto.gomes@animaeducacao.com.br

Prof. Dr. Fernando Kakugawa

fernando.kakugawa@animaeducacao.com.br