Análise Léxica

ÉFREN L. SOUZA

Roteiro

1. Introdução
2. O papel do analisador léxico
3. Especificação de Tokens
4. Reconhecimento de *Tokens*
5. Bibliografia

O Papel do Analisador Léxico

Considerando o fragmento de código...

- A função da análise léxica é dividir esse código em unidades léxicas
- Para um programador, isso não é difícil, uma vez que ele conhece a linguagem e ainda possui a ajuda visual

```
int w;
if ( x == y )
    w = 1;
else
    w = 0;
```

Considerando o fragmento de código...

- Entretanto, para o analisador léxico o programa é um arquivo de *bytes*
- Nesse caso ele precisa reconhecer as divisões entre as unidades léxicas
- A leitura é feita *byte* por *byte*

```
int w;
if ( x == y )
    w = 1;
else
    w = 0;
```

```
int w;\nif ( x == y )\n\tw = 1;\nelse\n\tw = 0;
```

Analisador Léxico

- Também chamado de *lexer*
- É a primeira fase do processo de compilação
- A função do analisador léxico é:
  - Ler os caracteres do programa fonte
  - Agrupá-los em **lexemas** de acordo com um **padrão**
  - Produzir um ***token*** para cada lexema

Termos semelhantes

- O *lexer* possui três conceitos importantes
  - Lexema
  - Padrão
  - Token

Lexema

- Um **lexema** é uma sequência de caracteres do programa fonte que seguem um padrão específico
- Cada lexema casa com o **padrão** de um tipo de **tokens**
- Exemplos de lexema
  - if
  - <=
  - 3.1415

Padrão

- Um padrão é uma descrição da forma que o lexema pode assumir

- Exemplos de padrão
  - Caractere “i” seguido do caractere “f”
  - Sequência de um ou mais dígitos
  - Letra seguida de letras ou dígitos

Token

- É um par formando por um tipo e um atributo
  - $\langle tipo, atributo \rangle$
- O **tipo** do *token* é um é um símbolo abstrato que define uma classe para o **lexema**
- O **atributo** é geralmente o próprio **lexema**

Exemplos de Tokens

Tipo do Token (TAG)	Descrição (padrão)	Lexemas
IF	Caracteres i, f	if
ELSE	Caracteres e, l, s, e	else
REL	< | > | <= | >=	<, >, <=, >=
ID	Letra seguida de letras ou dígitos	score, D2, max, x
LIT_INT	Sequência de dígitos	123, 432, 5870, 0

Tipos de Tokens mais comuns

- Para cada palavra-chave
- Para os operadores
- Para os identificadores
- Para os literais
- Para cada símbolo de pontuação da linguagem

Além dos *tokens*

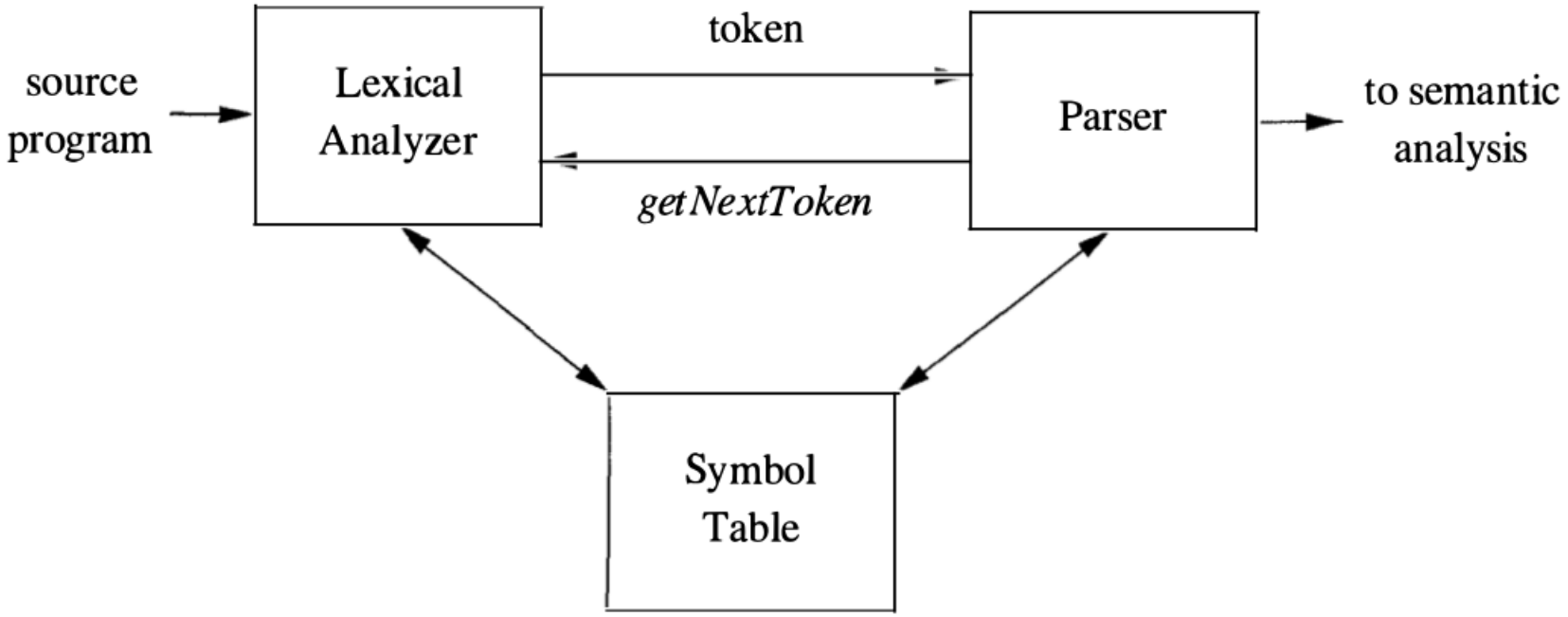
- Expansão de macros
- Remover comentários e espaços em branco
  - espaço, tabulação e quebra de linha
- Correlacionar mensagens de erro com a linha do código fonte
  - Registra o número de quebras de linha

Lexer em duas etapas

- O *lexer* pode ser dividido em duas partes
- *Scanning*
  - Não se preocupa com *tokens*
  - Remove comentários e espaços em branco
  - Expande o macros
- Análise léxica propriamente dita
  - Etapa mais complexa
  - Gera a sequência de *tokens* como saída

Interações do Lexer

- Interação com o analisador sintático (*parser*)
  - Os *tokens* produzidos pelo *lexer* são enviados para o *parser*
- Interação com a tabela de símbolos
  - Para diferenciar identificadores de palavras reservadas



Separação das análises Léxica e Sintática

- Simplicidade do projeto
- Melhora a eficiência
  - Permite a aplicação de técnicas específicas de cada análise
- Melhora a portabilidade

Erros Léxicos

- É difícil para o analisador léxico encontrar um erro
- `fi ( a == f(x) )`
  - `fi` é a palavra-chave `if` escrita errada?
  - `fi` é um identificador?

- Erros léxicos são caracteres não previstos ou padrões que não existem na linguagem

Especificação de *Tokens*

Especificação de *Tokens*

- As **expressões regulares** são usadas para especificar padrões de lexemas

- Embora não possam expressar qualquer padrão, são úteis para expressar padrões usados na prática

Expressões Regulares

- As expressões regulares expressam **linguagens regulares** usando as operações sobre linguagens

Operação	Precedência
União	Baixa
Concatenação	Intermediaria
Fecho de Kleene	Alta
Fecho positivo de Kleene	

Definições Regulares

- São nomes dados a certas expressões regulares

- Esses nomes podem ser usados em expressões posteriores como se fossem símbolos

Definições Regulares para Identificadores

$$LETRA \longrightarrow A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z$$

$$DIGITO \longrightarrow 0 \mid 1 \mid \ldots \mid 9$$

$$ID \longrightarrow LETRA(LETRA \mid DIGITO)^*$$

Qual é a definição regular de uma palavra reservada?

$$SENAO \longrightarrow else$$

Qual é a definição regular dos operadores relacionais do C?

$$REL \longrightarrow < \mid > \mid <= \mid >=$$

Qual é a definição regular de um literal inteiro do C?

$$DIGITO \longrightarrow 0 \mid 1 \mid \ldots \mid 9$$

$$LIT\_INT \longrightarrow DIGITO(DIGITO)^*$$

$$DIGITO\_NAO\_NULO \longrightarrow 1 \mid 2 \mid \ldots \mid 9$$

$$DIGITO \longrightarrow 0 \mid 1 \mid \ldots \mid 9$$

$$LIT\_INT \longrightarrow 0 \mid DIGITO\_NAO\_NULO(DIGITO)^*$$

Reconhecimento de Tokens

Reconhecimento

- Estuda os padrões de lexemas para construir o código que irá gerar os *tokens*

- Como passo intermediário (antes do código), podemos representar os padrões como **diagramas de transição**

Diagramas de transição – Estados

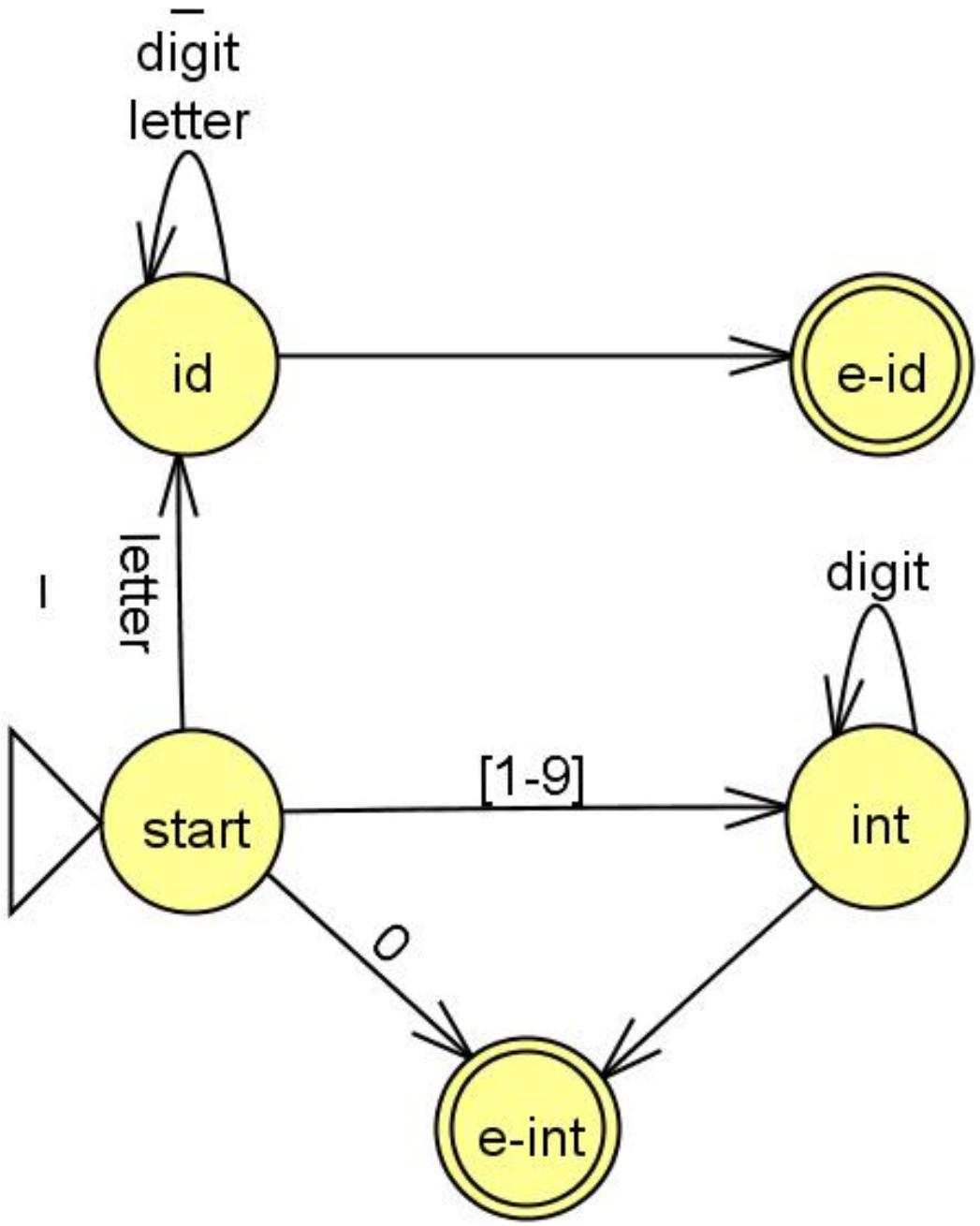
- Possuem um conjunto de nós chamados estados
  - Cada estado representa uma situação que pode ocorrer enquanto se lê o código fonte para reconhecer os *tokens*
  - Inicia em um estado inicial antes que qualquer leitura
  - Um estado final indica que um lexema foi encontrado

Diagramas de transição – Transições

- Possuem um conjunto de arestas direcionadas chamadas transições
  - Representa o movimento de um estado para outro dependendo do símbolo lido do programa fonte

Identificadores e literais inteiros

- Uma vez no estado `id`, a leitura de qualquer outro símbolo que não seja letra ou dígito leva a geração de um identificador
- Uma vez no estado `int`, a leitura de qualquer outro símbolo que não seja um dígito leva a um inteiro
- Se o primeiro dígito for um zero, o lexema é reconhecido como o inteiro 0



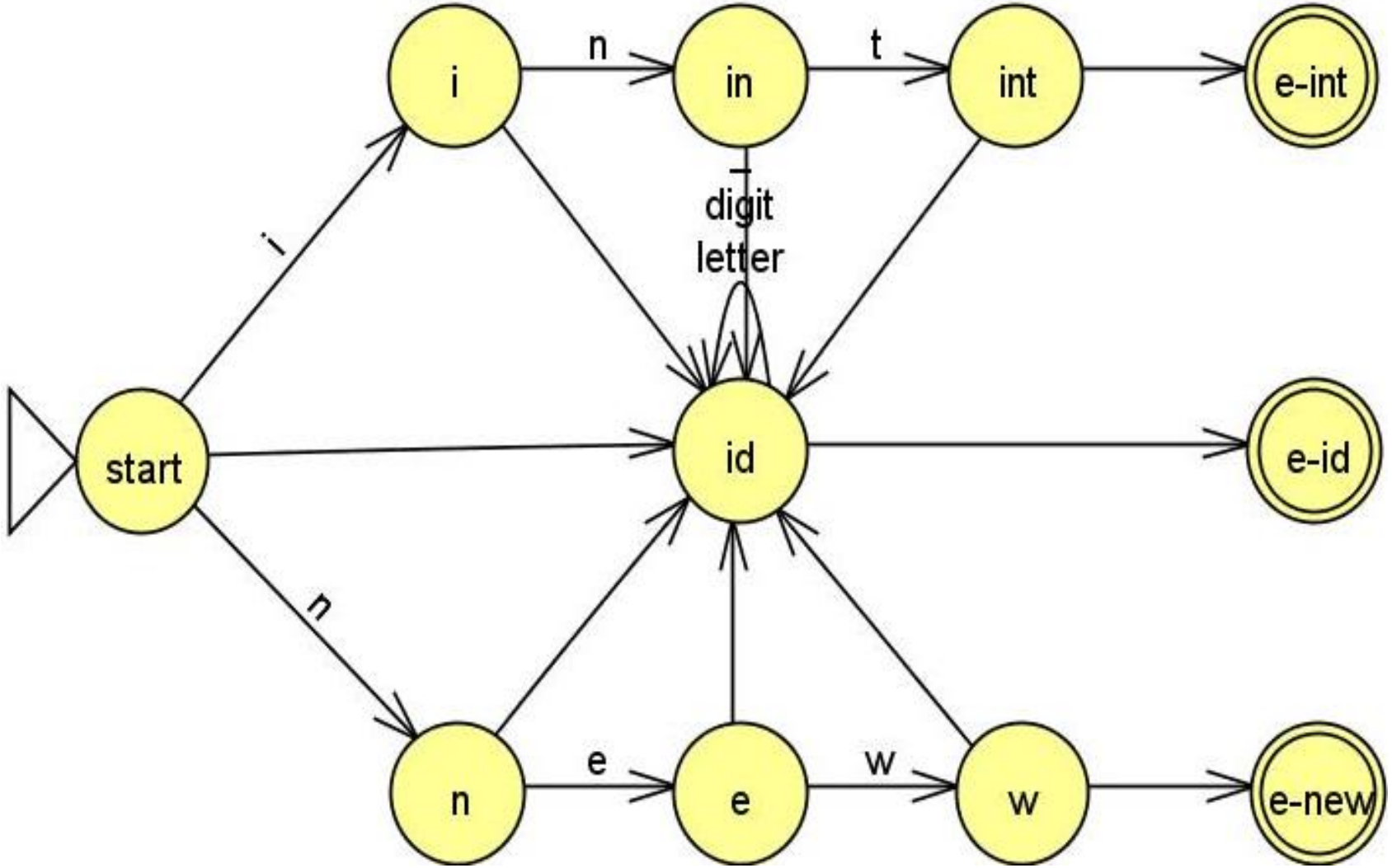
Identificadores e literais inteiros

- A variável identificada por `ch` guarda o valor do último caractere lido pela função `nextCh`
- Os ciclos se tornam comandos `while`
- As outras transições se tornam comandos `if`

```
if (isLetter(ch) || ch == '_' ) {
    buffer = new StringBuffer();
    while (isLetter(ch) || isDigit(ch) || ch == '_'){
        buffer.append(ch);
        nextCh();
    }
    return new TokenInfo(IDENTIFIER, buffer.toString(), line);
}
else if (ch == '0') {
    nextCh();
    return new TokenInfo(INT_LITERAL, "0", line);
}
else if (isDigit(ch)){
    buffer = new StringBuffer();
    while (isDigit(ch)) {
        buffer.append(ch);
        nextCh();
    }
    return new TokenInfo(INT_LITERAL, buffer.toString(), line);
}
```

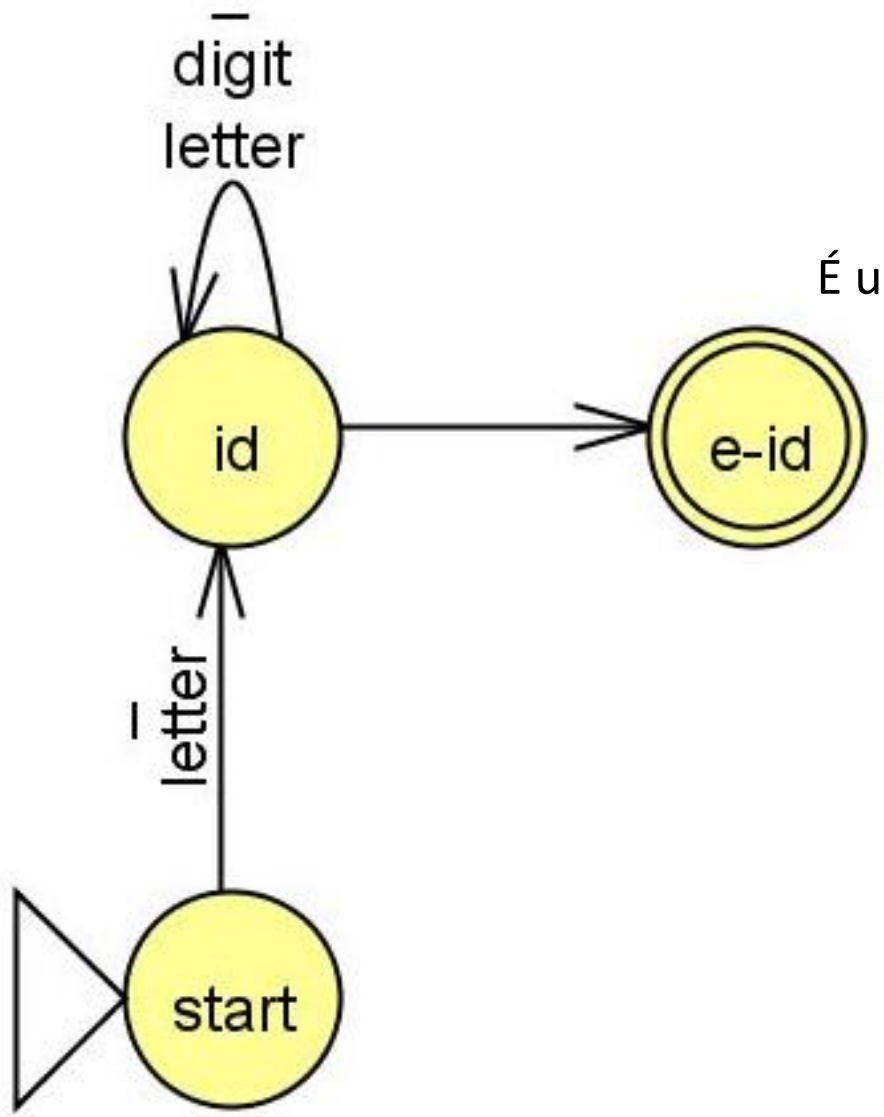
Palavras Reservadas

- Há duas formas de reconhecer palavras reservadas
- Na primeira todas as palavras reservadas são previstas em um complexo diagrama de transições
- A sequencia só é um identificador se não combinar com o padrão de alguma palavra reservada



Palavras Reservadas

- A segunda forma simplesmente reconhece identificadores que podem ser palavras reservadas ou não
- Nesse caso há uma tabela de palavras reservadas que é checada



É uma palavra reservada?

Sim – Retorne palavra reservada

Não – Retorne identificador

Palavras Reservadas

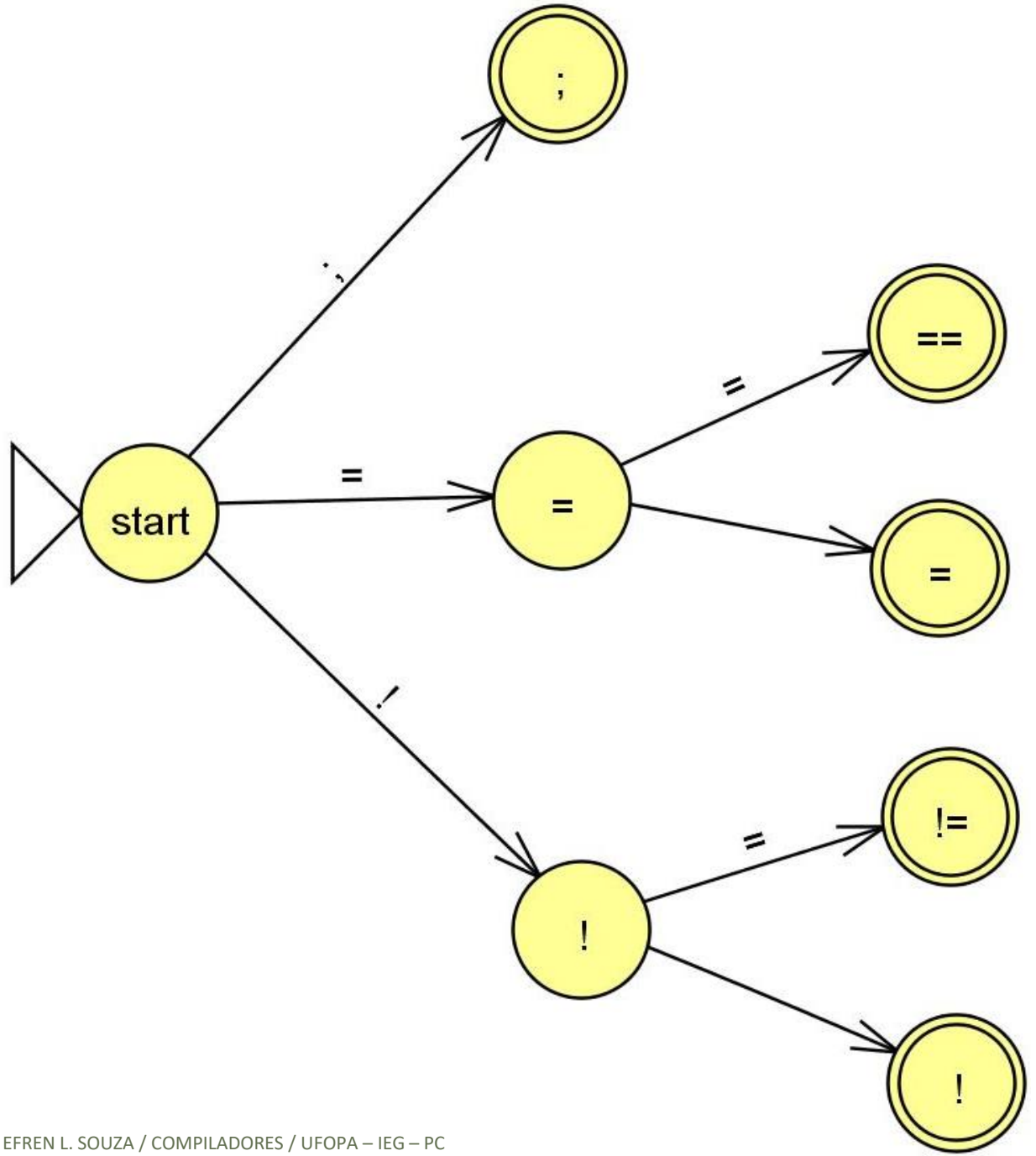
- Antes do reconhecimento, a tabela de palavras reservadas é preenchida
- A análise sintática reconhece o padrão de um identificador e checa se ela existe nessa tabela
- Se ela existir, se trata de uma palavra reservada, caso contrário se trata de um identificador

```
reserved = new Hashtable<String, Integer>();
reserved.put("abstract", ABSTRACT);
reserved.put("boolean", BOOLEAN);
reserved.put("char", CHAR);
...
reserved.put("while", WHILE);
```

```
if (isLetter(ch) || ch == '_' ) {
    buffer = new StringBuffer();
    while (isLetter(ch) || isDigit(ch) ||
           ch == '_'){
        buffer.append(ch);
        nextCh();
    }
    String identifier = buffer.toString();
    if (reserved.containsKey(identifier)) {
        return new TokenInfo(reserved.get(identifier),
                             line);
    }
    else {
        return new TokenInfo(IDENTIFIER, identifier,
                             line);
    }
}
```

Separadores e Operadores

- Os diagramas de transição funcionam de forma bem natural para esses padrões
- Só se deve tomar cuidado com alguns operadores que possuem mais do que um caractere

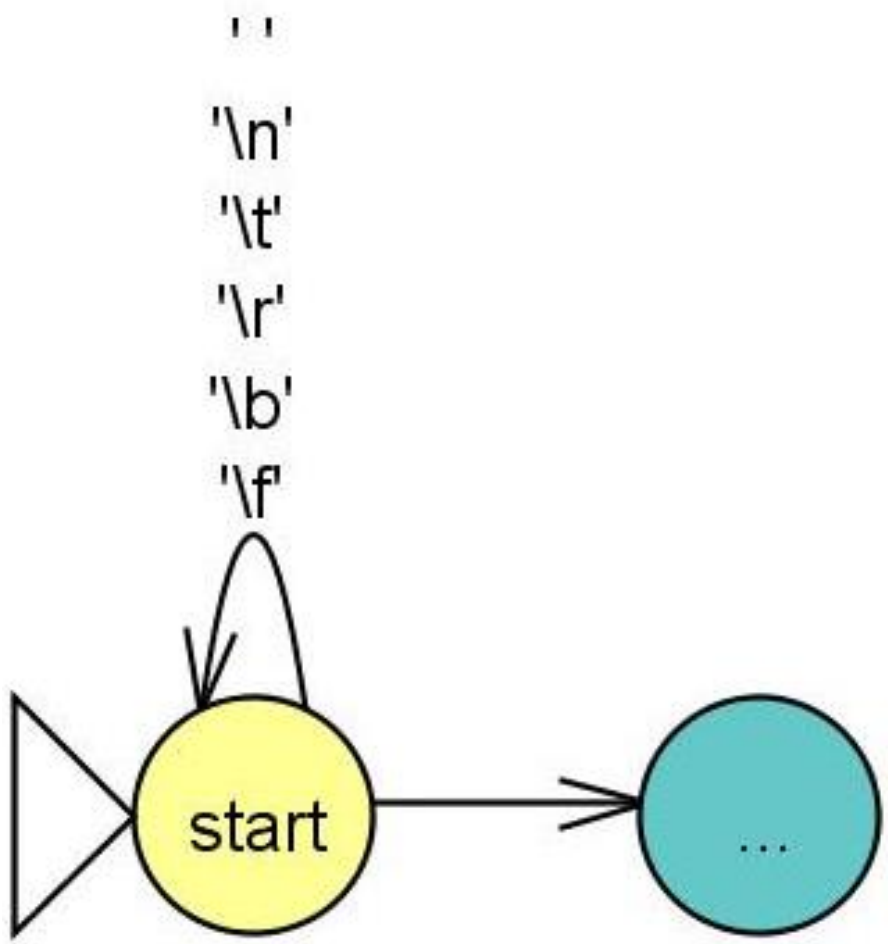


Separadores e Operadores

```
switch (ch) {
...
case ';':
    nextCh();
    return new TokenInfo(SEMI, line);
case '=':
    nextCh();
    if (ch == '=') {
        nextCh();
        return new TokenInfo(EQUAL, line);
    }
    else {
        return new TokenInfo(ASSIGN, line);
    }
case '!':
    nextCh();
    return new TokenInfo(LNOT, line);
```

Espaços em branco

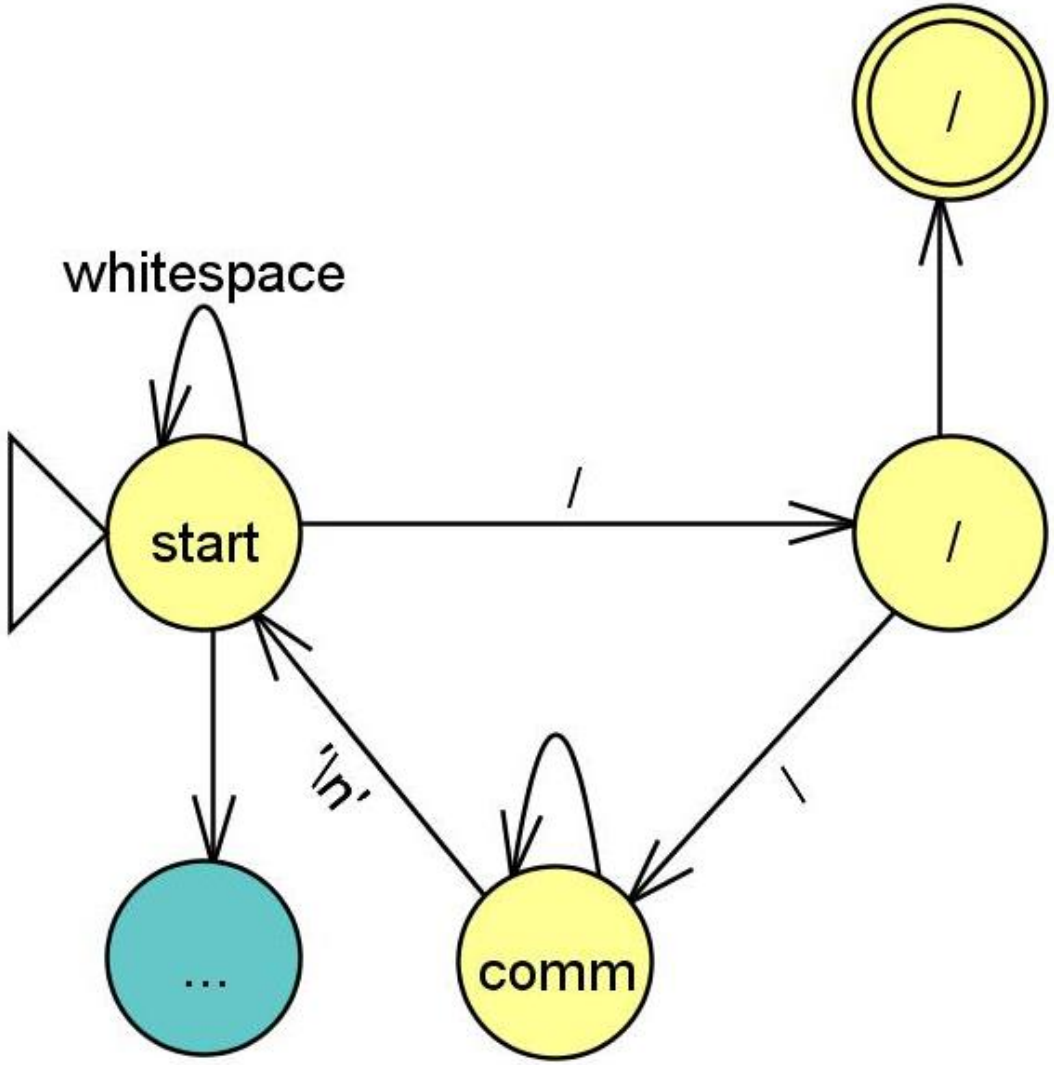
- Uma das funções do analisador léxico é pular todos os espaços em branco
- Os espaços em branco geralmente são:
  - Espaços
  - Tabulações
  - Quebras de linha




```
while (isWhitespace(ch)) {
    nextCh();
}
```

Comentários

- No caso do comentário de única linha, tudo entre / / e quebra de linha deve ser ignorado



Bibliografia

- AHO, A. V.; SETHI, R.; ULLMAN, J. D. Compiladores: princípios, técnicas e ferramentas. Rio de Janeiro: LTC, 1995.
- CAMPBELL, B.; LYER, S.; AKBAL-DELIBAS, B. Introduction to Compiler: Construction in a Java World. CRC Press, 2013.
- APPEL, A. W. Modern compiler implementation in C. Cambridge. Cambridge University Press, 1998.