CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

- **Specification of Tokens**
  - ~~**String and Languages**~~
  - ~~**Operations on Languages**~~
  - **Regular Expressions (We start from here today)**
  - **Regular Definitions**
  - **Extensions of Regular Expressions**
- **Recognition of Tokens (Ch 3.4)**
  - **Transition Diagrams (Ch 3.4.1) (This one will be covered partially today. Good for resources as hand-written homework ☺)**

Regular Expressions

- In Example 3.3 (in previous slide), we were able to **describe identifiers** by giving names to sets of letters and digits and using the language **operators** union, concatenation, and closure.
- Regular Expression is a useful tool that is used to describe all the languages that can be built from those operators applied to the symbols of some alphabet
  - In this notation, if letter is established to stand for any **letter or the underscore**, and
  - Digit is established to stand for any digit, then we can describe the C language's identifiers by:

$$letter\_ \ (\ letter\_ \mid digit \ )^*$$

Regular Expressions

- The vertical bar above means union (or) $letter\_ \ (\ letter\_ \mid digit \ )^*$
- The parentheses are used to group sub-expressions
- The star means **zero or more** occurrences of…something
- The regular expressions are built recursively out of **smaller** regular expressions, using the rules described below:
  - Each regular expression r denotes **a language L(r),** which is also denoted **recursively** from languages denoted by r's sub-regular expressions

Regular Expressions

- Based on the knowledge of larger regular expressions are built from smaller ones, we have the following properties: (suppose r and s are regular expressions denoting languages L(r) and L(s), respectively.
  - (r)|(s) is a regular expression denoting the language L(r) | L(s)
  - (r)(s) is a regular expression denoting the language L(r)L(s)
  - (r)* is a regular expression denoting, (L(r))*
  - (r) is a regular expression denoting L(r)
- A couple of conventions
  - operator * has highest precedence and is left associative.
  - Concatenation has second highest precedence and is left associative
  - | has lowest precedence and is left associative.

Regular Expressions

- For example, a | b*c means:
  - A set of strings that are either a “single a” or are “zero or more b(s)” followed by one c.
- Some other examples, the regular expressions over some alphabet Σ
  - Let Σ = {a, b}
    - regular expression a|b denotes language {a, b}
    - (a|b)(a|b) denotes {aa, ab, ba, bb}; (a|b)(a|b) can be rewritten as aa|ab|ba|bb
    - a* denots all strings of zero or more a(s), that is {epsilon, a, aa, aaa,…}
    - (a|b)* means {epsilon, a, b, aa, ab, ba ,bb, aaa,…}. It can be rewritten as (a*b*)*
    - a|a*b means the language {a, b, ab, aab, aaab,…}

Regular Expressions

- A language that can be defined by a regular expression is called a regular set.
- If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s. For instance, (a|b) = (b|a).
- Figure 3.7 shows some of the algebraic laws that hold for arbitrary regular expressions r, s, and t.
- (See the next page for algebraic laws)

Regular Expressions

LAW	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	$*$ is idempotent

Figure 3.7: Algebraic laws for regular expressions

Regular Definitions

- Regular definitions are just a convenience; they add no power to regular expressions.
- See the following example, a *regular definition* is a sequence of definitions
- An important difference between regular definitions and productions (the later one is more powerful) is that, **each d_i cannot depend on following d's**
- **r_i are regular expressions**

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

Regular Definitions

Example: C identifiers can be described by the following regular definition

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

- letter_ is not depending on Cid, because if CId is crossed out, letter_ is still good.

Extensions of Regular Expressions

- The references to this chapter contain a discussion of some regular-expression (RE) variants (extensions) in use today
  - One or more instances
    - This means the positive closure of RE and its language
    - Here are the tricks of the rules:

$$r^* = r^+|\epsilon \text{ and } r^+ = rr^* = r^*r$$

  - Zero or one instance
    - The unary postfix operator ? means “zero or one occurrence."
    - r? is equivalent to r|epsilon, or put another way, L(r?) = L(r) U {epsilon}.
  - Character classes
    - [abc] is shorthand for a|b|c, and [a-z] is shorthand for a|b|…|z|

Extensions of Regular Expressions

Example 3.5 : C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$\begin{array}{rcl} letter\_ & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\ digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ id & \rightarrow & letter\_ \; (\; letter\_ \mid digit \;)^* \end{array}$$

Example 3.6 : Unsigned numbers (integer or floating point) are strings such as `5280`, `0.01234`, `6.336E4`, or `1.89E-4`. The regular definition

$$\begin{array}{rcl} digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ digits & \rightarrow & digit \; digit^* \\ optionalFraction & \rightarrow & \texttt{.} \; digits \mid \epsilon \\ optionalExponent & \rightarrow & (\; \texttt{E} \; (\; \texttt{+} \mid \texttt{-} \mid \epsilon \;) \; digits \;) \mid \epsilon \\ number & \rightarrow & digits \; optionalFraction \; optionalExponent \end{array}$$

Example 3.7 : Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

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

The regular definition of Example 3.6 can also be simplified:

$$\begin{array}{rcl} digit & \rightarrow & \texttt{[0-9]} \\ digits & \rightarrow & digit^+ \\ number & \rightarrow & digits \; (\texttt{.} \; digits)? \; (\; \texttt{E} \; \texttt{[+-]}? \; digits \;)? \end{array}$$

Recognition of Tokens

- In the example from the book, our current goal is to perform the lexical analysis needed for the following grammar

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | ε
expr → term relop term   // relop is relational operator =, >, etc
     | term
term →  id
     | number
```

- Recall that the terminals are the tokens, the non-terminals can produce terminals. (“term”)
- relop? (see next page)

A regular definition for the terminals is

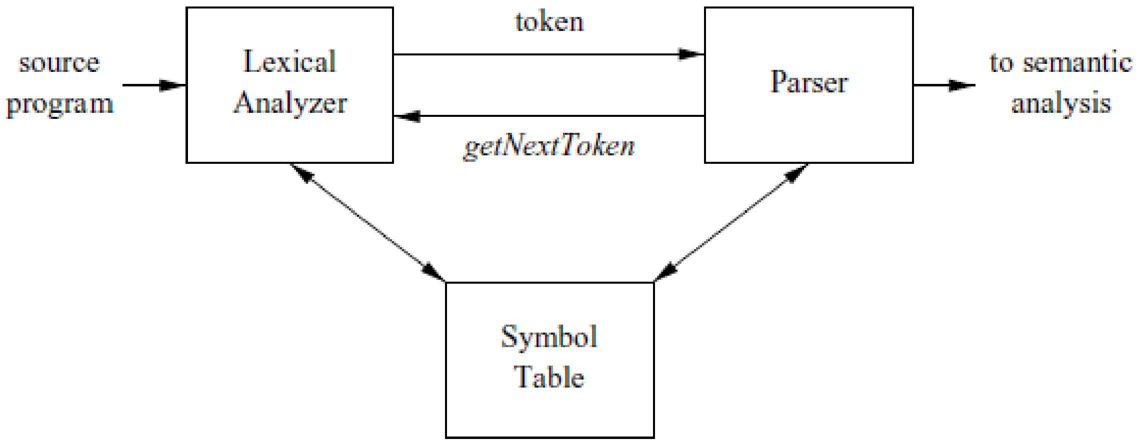
```
digit → [0-9]
digits → digits+
number → digits (. digits)? (E[+-]? digits)?
letter → [A-Za-z]
id → letter ( letter | digit )*
if → if
then → then
else → else
relop → < | > | <= | >= | = | <>
```

Recognition of Tokens

- For the parser, all the **relational ops** are to be treated the same so they are all the same token, relop
- For example, the very special ops for some languages, SQL

Recognition of Tokens

- We also want the lexer to remove white space so we define a new token
- ws → (blank | tab | newline) +
- Recall that the lexer will be called by the parser when the latter needs a new token.
- **If the lexer then recognizes the token ws, it does *not* return it to the parser** but instead, goes on to recognize the next token, which is then returned



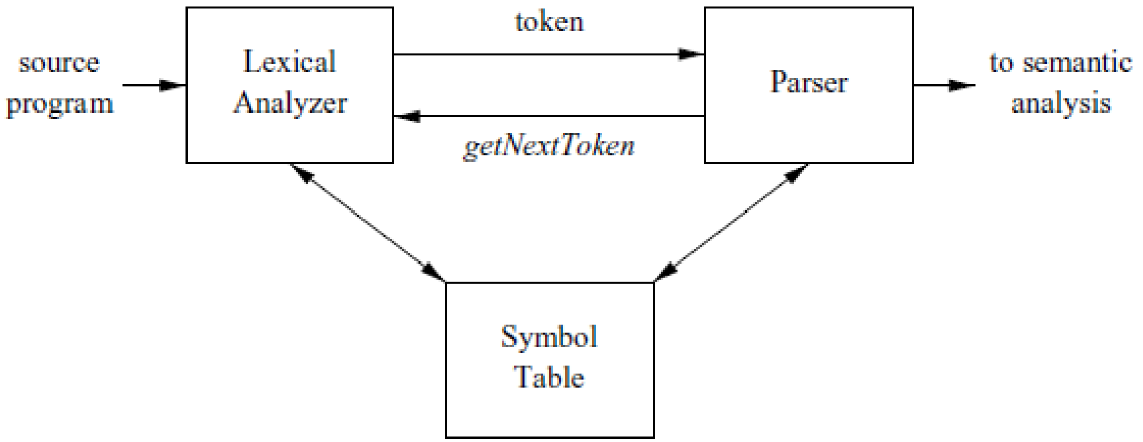
Recognition of Tokens

- For a given token, the lexer will match the **longest** lexeme starting at the current position that yields this token.
- The table on the right summarizes the situation
- These entries are saying “no Attribute”

Lexeme	Token	Attribute
Whitespace	ws	—
if	if	—
then	then	—
else	else	—
An identifier	id	Pointer to table entry
A number	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Transition Diagrams

- As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called transition diagrams. This means, some mechanism in this box
- Transition diagrams have a collection of **nodes** or **circles**, called **states**.



Transition Diagrams

- Each state represents a condition that could occur during the process of “scanning the input looking for a lexeme” that matches one of several patterns
- We can say a “**state**” is summarizing all we need to know about what characters we have seen --- between the **lexemeBegin** pointer and the **forward** pointer, as in the Fig. 3.3

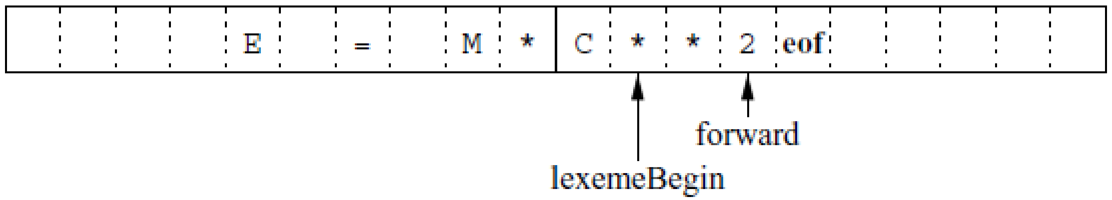
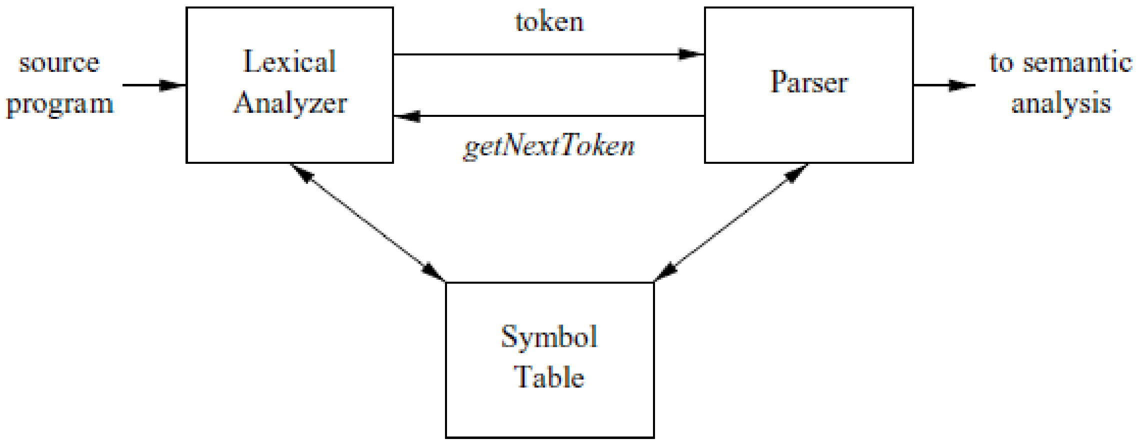


Figure 3.3: Using a pair of input buffers



Transition Diagrams

- Edges are directed from one state of the transition diagram to another.
- Each edge is **labeled** by a **symbol** or **set of symbols**
- If we are in some state "s", and the next input symbol is "a", **we look for an edge out of state s labeled by a** (and perhaps by other symbols, as well).e is labeled by a symbol or set of symbols
- If we find such an edge, we can advance the **forward pointer** and enter the state of the transition diagram to which that edge leads.
- We shall assume that all our transition diagrams are **deterministic**, meaning that there is never more than one edge out of a given state with a given symbol (i.e. "a", in our previous example) among its labels.

Transition Diagrams

- Some important conventions about transition diagrams
  - (TBD. In Part 4) Will be covered in the next lecture
  - Kind of complicated, let's peek the diagram quickly!?
  - I will explain that next time!

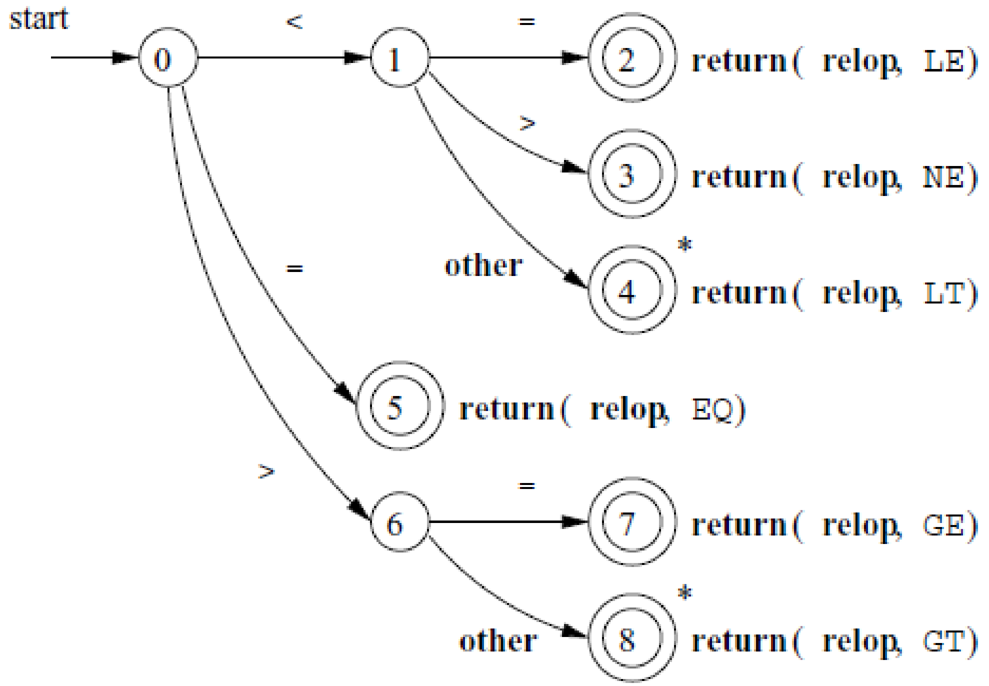


Figure 3.13: Transition diagram for **relop**