CSE420 Compiler Design

Lecture: 3
Lexical Analysis (Part 1)

Lexical Analysis

- Basic Concepts & Regular Expressions
  - What does a Lexical Analyzer do?
  - How does it Work?
  - Formalizing Token Definition & Recognition
- Reviewing Finite Automata Concepts
  - Non-Deterministic and Deterministic FA
  - Conversion Process
    - Regular Expressions to NFA
    - Regular Expressions to DFA
- Relating NFAs/DFAs /Conversion to Lexical Analysis

Lexical Analysis

The **lexical analyzer** breaks the syntaxes into a series of *tokens*, by removing any whitespace or comments in the source code.

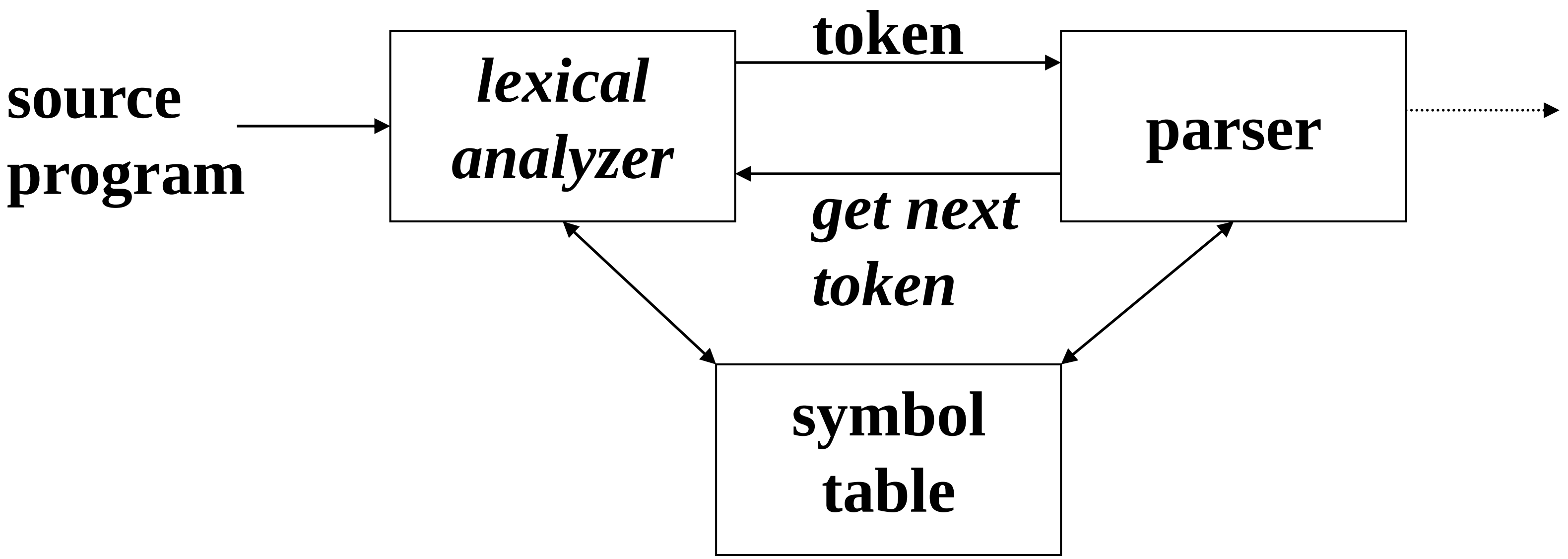
Code segment:

```
if(x/2==0)
	even=1;
else
	odd=1;
```

The text script of this code segment:

```
if(x/2==0)\n\teven=1;\nelse\n\todd=1;
```

Lexical Analyzer in Perspective



Important Issue:

What are Responsibilities of each Box ?

Focus on Lexical Analyzer and Parser.

Lexical Analyzer in Perspective

- **LEXICAL ANALYZER**
  - Scan Input
  - Remove WS, NL, …
  - Identify Tokens
  - Create Symbol Table
  - Insert Tokens into ST
  - Generate Errors
  - Send Tokens to Parser

- **PARSER**
  - Perform Syntax Analysis
  - Actions Dictated by Token Order
  - Update Symbol Table Entries
  - Create Abstract Rep. of Source
  - Generate Errors
  - And More…. (We'll see later)

Introducing Basic Terminology

- What are Major Terms for Lexical Analysis?
  - **TOKEN**
    - A classification for a common set of strings
    - Examples Include <Identifier>, <number>, etc.
  - **PATTERN**
    - The rules which characterize the set of strings for a token
    - Recall File and OS Wildcards ([A-Z]*.*)
  - **LEXEME**
    - Actual sequence of characters that matches pattern and is classified by a token
    - Identifiers: x, count, name, etc…

Introducing Basic Terminology

Token	Sample Lexemes	Informal Description of Pattern
const	const	const
if	if	if
relation	<, <=, =, < >, >, >=	< or <= or = or < > or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	“core dumped”	any characters between “ and “ except “

Classifies Pattern

Actual values are critical. Info is :

1. Stored in symbol table
2. Returned to parser

Language and Regular Expressions

- A Regular Expression is a Set of Rules / Techniques for Constructing Sequences of Symbols (Strings) From an Alphabet.
- Let Σ Be an Alphabet, r a Regular Expression Then $L(r)$ is the Language that is Characterized by the Rules of r.

Rules for specifying Regular Expressions

- Regular expressions over alphabet Σ

1. ε is a regular expression that denotes $\{\varepsilon\}$.
2. If **a** is a symbol (i.e., if **a** $\in \Sigma$), then **a** is a regular expression that denotes $\{a\}$.
3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then
   a) (r) | (s) is a regular expression denoting L(r) U L(s).
   b) (r)(s) is a regular expression denoting L(r)L(s).
   c) $(r)^*$ is a regular expression denoting $(L(r))^*$.
   d) (r) is a regular expression denoting L(r).

Example

- Let $\Sigma = \{a, b\}$
  - The regular expression a | b denotes the set {a, b}
  - The regular expression (a|b)(a|b) denotes {aa, ab, ba, bb}
  - The regular expression a^* denotes the set of all strings of zero or more a's. i.e., {ε, a, aa, aaa, ….. }
  - The regular expression $(a|b)^*$ denotes the set containing zero or more instances of an a or b.
  - The regular expression a|a*b denotes the set containing the string a and all strings consisting of zero or more a's followed by one b.

How to “Parse” Regular Expressions

- **Precedence:**
  - * has highest precedence.
  - Concatenation as middle precedence.
  - | has lowest precedence.
  - Use parentheses to override these rules.

- *Examples:*
  - **a b* = a (b*)**
    - If you want **(a b)*** you must use parentheses.
  - **a | b c = a | (b c)**
    - If you want **(a | b) c** you must use parentheses.

- Concatenation and | are associative.
  - **(a b) c = a (b c) = a b c**
  - **(a | b) | c = a | (b | c) = a | b | c**
- *Example:*
  - **b d | e f * | g a = (b d) | (e (f *)) | (g a)**

Equality vs Equivalence

- Are these regular expressions equal?

  R = **a a* (b | c)**

  S = **a* a (c | b)**

  ... No!

- Yet, they describe the same language.

  L(R) = L(S)
- "Equivalence" of regular expressions

  If L(R) = L(S) then we say R ≅ S

  "R is equivalent to S"
- From now on, we'll just say R = S to mean R ≅ S

Algebraic law of regular expressions

Let R, S, T be regular expressions...

| is commutative

$$R \mid S = S \mid R$$

| is associative

$$R \mid (S \mid T) = (R \mid S) \mid T = \overbrace{R \mid S \mid T}^{\textit{Preferred}}$$

Concatenation is associative

$$R\,(S\,T) = (R\,S)\,T = \underbrace{R\,S\,T}_{\textit{Preferred}}$$

Concatenation distributes over |

$$R\,(S \mid T) = RS \mid RT$$

$$(R \mid S)\,T = RT \mid ST$$

ε is the identity for concatenation

$$\varepsilon R = R \varepsilon = R$$

* is idempotent

$$(R^*)^* = R^*$$

Relation between * and ε

$$R^* = (R \mid \varepsilon)^*$$

Regular Definition

```
Letter  =  a | b | c | ... | z
Digit   =  0 | 1 | 2 | ... | 9
ID      =  Letter ( Letter | Digit )*
```

Addition Notation / Shorthand

One-or-more: $+$

$X^{+} = X(X^{*})$

Digit^{+} = Digit Digit* = Digits

Optional (zero-or-one): ?

$X? = (X \mid \varepsilon)$

Num = Digit^{+} (. Digit^{+}) ?

Character Classes: [*FirstChar*−*LastChar*]

Assumption: The underlying alphabet is known ...and is ordered.

Digit = [0-9]

Letter = [a-zA-Z] = [A-Za-z]

Variations:

Zero-or-more: ab*c = a{b}c = a{b}*c

One-or-more: ab+c = a{b}+c

Optional: ab?c = a[b]c

Token Recognition

How can we use concepts developed so far to assist in recognizing tokens of a source language ?

<u>Given Tokens, What are Patterns ?</u>

digit → 0 | 1 | 2 | … | 9

id → letter (letter | digit)*

num → digit $^+$ (. digit $^+$) ? (E(+ | -) ? digit $^+$) ?

Unsigned Number

digit → 0 | 1 | 2 | … | 9

digits → digit digit*

optional_fraction → . **digits** | ∈

optional_exponent → (E (+ | -| ∈) **digits**) | ∈

num → **digits** **optional_fraction** **optional_exponent**

Shorthand

digit → 0 | 1 | 2 | … | 9

digits → digit^{+}

optional_fraction → (. **digits**) ?

optional_exponent → (E (+ | -) ? **digits**) ?

num → **digits** **optional_fraction** **optional_exponent**

1240, 39.45, 6.33E15, or 1.578E-41

What Else Does Lexical Analyzer Do?

Scan away *blanks*, new lines, tabs

Can we Define Tokens For These?

$$\textbf{blank} \rightarrow \textbf{blank}$$

$$\textbf{tab} \rightarrow \textbf{tab}$$

$$\textbf{newline} \rightarrow \textbf{newline}$$

$$\textbf{delim} \rightarrow \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}$$

$$\textbf{ws} \rightarrow \textbf{delim}^{+}$$

Ans: No token is returned to parser

Token Recognition

How can we use concepts developed so far to assist in recognizing tokens of a source language ?

Assume Following Tokens:

if, then, else, relop, id, num

Given Tokens, What are Patterns ?

Grammar:

stmt → |if *expr* then *stmt*
|if *expr* then *stmt* else *stmt*
|$\in$
expr → *term* relop *term* | *term*
term → id | num

if → **if**

then → **then**

else → **else**

relop → < | <= | > | >= | = | <>

id → **letter (letter | digit)***

num → **digit $^{+}$ (. digit $^{+}$) ? (E(+ | -) ? digit $^{+}$) ?**

What Else Does Lexical Analyzer Do?

All Keywords / Reserved words are matched as ids

- **After the match, the symbol table or a special keyword table is consulted**
- **Keyword table contains string versions of all keywords and associated token values**

if	**15**
then	**16**
begin	**17**
...	**...**

- **When a match is found, the token is returned, along with its symbolic value, i.e., "then", 16**
- **If a match is not found, then it is assumed that an id has been discovered**

End of slide