CS301 W'99 Lecture Notes
Lecture 4

Lexical Analysis

Convert source file characters into **token stream**.

Remove content-free characters (comments, whitespace, ...)

Detect lexical errors (badly-formed literals, illegal characters, ...)

Output of lexical analysis is input to syntax analysis.

Could just do lexical analysis as part of syntax analysis.

But choose to handle separately for better modularity, portability.

Idea: Look for **patterns** in input character sequence, convert to **tokens** with **attributes**, and pass them to parser in **stream**.

Lexical Analysis Example

Pattern	Token	Attribute
`if`	IF	
`else`	ELSE	
`print`	PRINT	
`then`	THEN	
`:=`	ASSIGN	
`=` or `<` or `>`	RELOP	enum
letter followed by letters or digits	ID	symbol
digits	NUM	int
chars between double quotes	STRING	string

Source code:

```
if x>17 then count:= 2
          else (* oops !*) print "bad!"
```

Lexeme	Token	Attribute
`if`	IF	
`x`	ID	`"x"`
`>`	RELOP	`GT`
`17`	NUM	`17`
`then`	THEN	
`count`	ID	`"count"`
`:=`	ASSIGN	
`2`	NUM	`2`
`else`	ELSE	
`print`	PRINT	
`"bad!"`	STRING	`"bad!"`

More Details

A **token** describes a **class** of character strings with some distinguished meaning in language.

• May describe **unique** string (e.g., IF, ASSIGN)

• or set of possible strings, in which case an **attribute** is needed to indicate which.

(Tokens are represented as elements of an **enumeration**.)

A **lexeme** is the string in the input that actually matched the pattern for some token.

Attributes represent lexemes converted to a more useful form, e.g.,:

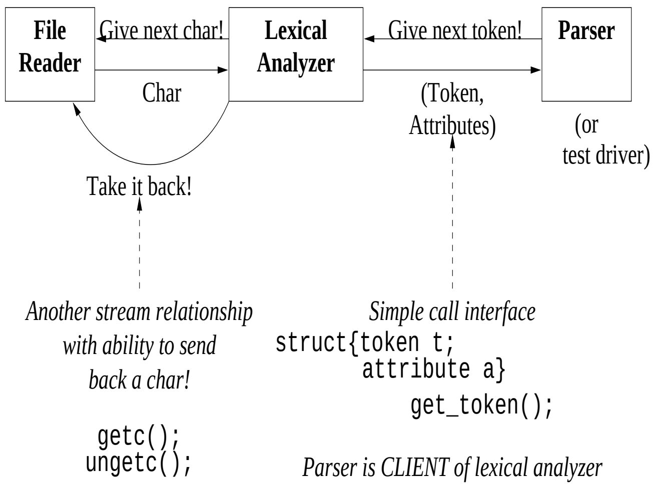
- strings
- symbols (like strings, but perhaps handled separately)
- numbers (integers, reals, ...)
- enumerations

Whitespace (spaces, tabs, new lines, ...) and comments just disappear!

Stream Interface

Could convert entire input file to list of tokens/attributes.

But parser needs only one token at a time, so use **stream** instead:



Hand-coded Scanner (in Psuedo-C)

```
get_token() {
  while (true) {
     c = getchar();
     if (c is whitespace)
       ignore it;
     else if (c is digit) {
        n = 0;
        do {n = n * 10 + (c- '0');
            c = getchar(); }
        until (c not a digit);
        ungetchar();
        return (NUM,n);
     } else if (c is alpha) {
        s = "";
        do { s = s ^ c;
             c = getchar();
        } until (c is not an alphanumeric);
        ungetchar();
        return (ID,S);
    } else ...
  }
}
```

Efficient! Easy to get wrong!

Note intermixed code for input, output, patterns, conversion.

Hard to specify! (esp. **patterns**).

Example

How can we formalize this pattern description?

"An **identifier** is a letter followed by any number of letters or digits."

- Exactly what is a letter?

```
LETTER →  a | b | c | d | e | f | g | h | i | j
        | k | l | m | n | o | p | q | r | s | t
        | u | v | w | x | y | z | A | B | C | D
        | E | F | G | H | I | J | K | L | M | N | O
        | P | Q | R | S | T | U | V | W | X | Y | Z
```

- Exactly what is a digit?

```
DIGIT →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- How can we express "letters or digits" ?

```
LORD → LETTER | DIGIT
```

- How can we express "any number of" ?

```
LORDS → LORD*
```

- How can we express "followed by" ?

```
IDENT → LETTER LORDS
```


Regular Expressions

A **regular expression (R.E.)** is a concise formal characterization of a **regular language** (or **regular set**).

Example: The regular language containing all IDENTs is described by the regular expression

```
letter (letter | digit)*
```

where "|" means "or" and "e^*" means "zero or more copies of e."

Regular languages are one particular kind of **formal** languages.

Languages: Some preliminary definitions

• An **alphabet** is a set of symbols (e.g., the ASCII character set).

• A **language** over an alphabet is a set of strings of symbols from that alphabet.

• We write ϵ for the **empty string** (containing zero characters); some authors use δ instead.

• If x and y are strings, then the **concatenation** xy is the string consisting of the the characters of x followed by the characters of y.

• If L and M are languages, then their **concatenation** $LM = \{xy \mid x \in L, y \in M\}$.

• The **exponentiation** of a language L is defined thus: $L^0 = \{\epsilon\}$, the language containing just the empty string, and $L^i = L^{i-1}L$ for $i > 0$.

Regular Expressions and Languages

Each **R.E.** over an alphabet Σ denotes a **regular language** over Σ, according to the following **inductive definition**:

Base rules:

• The R.E. ϵ denotes $\{\epsilon\}$.

• For each $a \in \Sigma$, the R.E. a denotes $\{a\}$, the language containing the single string containing just a.

Inductive rules:

If the R.E. R denotes L_R and the R.E. S denotes L_S, then

• $R|S$ denotes $L_R \cup L_S$.

• $R \cdot S$ (or just RS) denotes $L_R L_S$.

• R^* denotes $L_R^* = \bigcup_{i=0}^{\infty} L^i$, the **"Kleene closure"** (the concatenation of zero or more strings from L_R).

Also:

• (R) denotes L_R.

Precedence rules: () before * before $\cdot$ before $|$.

Regular Expressions

Examples (over alphabet $\{a, b\}$)

a^*	zero or more a's
$(a\vert b)^*$	all strings of a's and b's of length ≥ 0
$(a^*b^*)^*$	ditto
$(aa\vert ab\vert ba\vert bb)^*$	all strings of a's and b's of even length

Counterexamples (Not every language is regular!)

• $\{a^n b^n \mid n \geq 0\}$

• Set of strings over $\{(,)\}$ such that parentheses are properly matched.

Implication: regular languages can't be used to describe arithmetic expressions.

R.E.'s are everywhere in Unix

`grep`, `awk`, shell commands, etc.

Regular Definitions

Give names to R.E.'s and then use these as a shorthand.

• Must avoid recursive definitions!

• Example of syntactic sugar.

Examples:

ID	$\rightarrow$	letter (letter$\vert$digit)*
NUM	$\rightarrow$	digit (digit)*
		or this shorthand: digit$^+$
IF	$\rightarrow$	`if` *Not too useful!*
THEN	$\rightarrow$	`then`
RELOP	$\rightarrow$	$<\vert>\vert<=\vert>=\vert=$
		or: $<(\epsilon\vert=)\vert>(\epsilon\vert=)\vert=$
ASSGN	$\rightarrow$	`:=`
STRING	$\rightarrow$	"(nonquote)*"
letter	$\rightarrow$	a$\vert$b$\vert\ldots\vert$z$\vert$A$\vert\ldots\vert$Z
		or this shorthand: `[a-zA-Z]`
digit	$\rightarrow$	0$\vert$1$\vert$2$\vert$3$\vert$4$\vert$5$\vert$6$\vert$7$\vert$8$\vert$9
		or this shorthand: `[0-9]`
nonquote	is	letter$\vert$digit$\vert$!$\vert$\$$\vert$%$\vert\ldots$

Note that ID and keywords have overlapping patterns.

Specifying Lexical Analyzers

Can define lexical analyzer via list of pairs:

(**regular expression**, **action**)

where **regular expression** describes token pattern (maybe using auxiliary regular defns.)

and **action** is a function, parameterized by the matching lexeme, that returns a (token,attribute) pair

Example

(digit^+, {`return (NUM,strtoint(lexeme));`})
($\text{alpha}(\text{alpha}|\text{digit})^*$, {`return (ID,lexeme);`})
(space|tab|newline, {})
(., .)
(., .)
(., .)

So R.E'.s can help us **specify** scanners.

But can they help us **generate** running code that performs pattern matching?