CSE 331
Compiler Design

The Role of the Lexical Analyzer

- Roles
  - Primary role: Scan a source program (a string) and break it up into small, meaningful units, called tokens.
  - Example: position := initial + rate * 60;
  - Transform into meaningful units: identifiers, constants, operators, and punctuation.

- Other roles:
  - Removal of comments
  - Case conversion
  - Removal of white spaces

Why separate LA from parser?

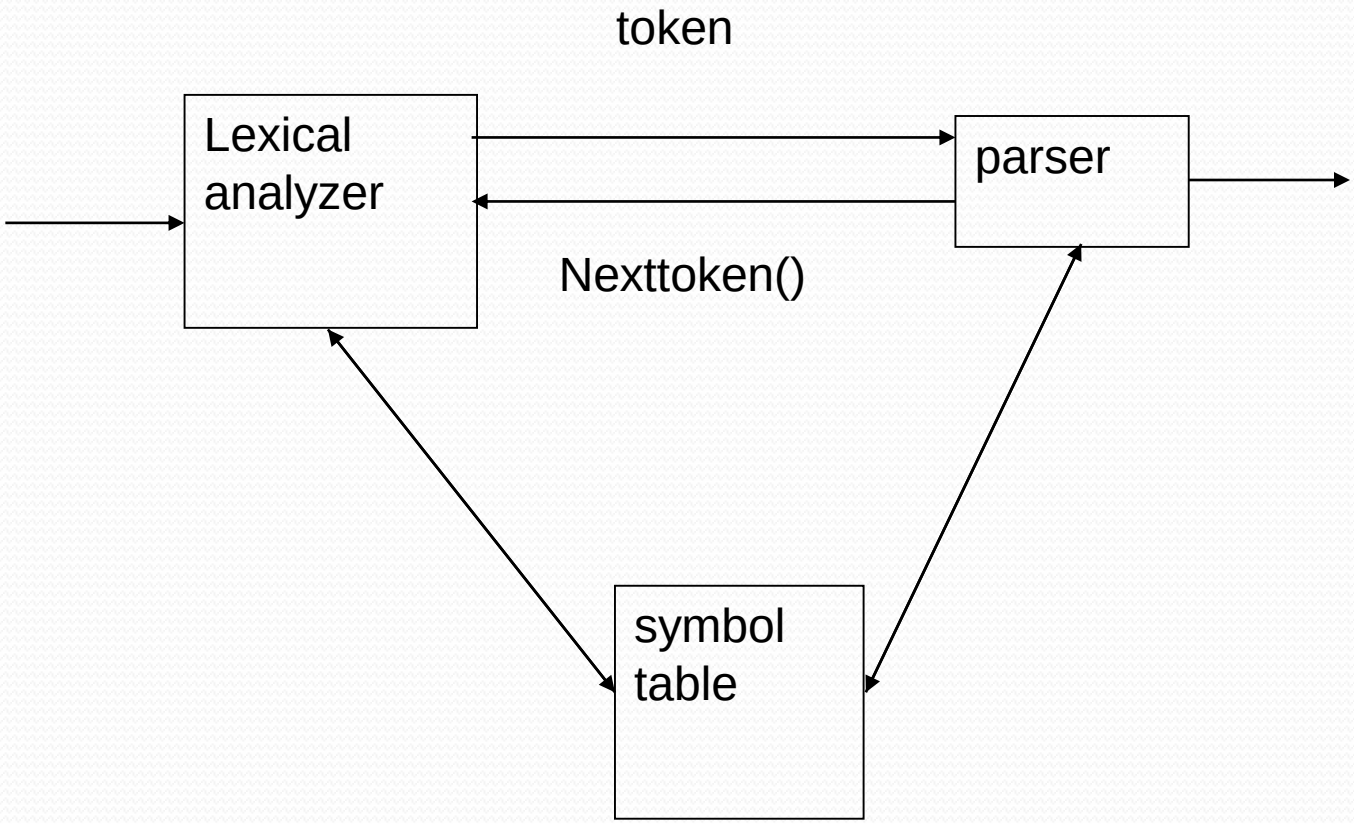
- Simpler design of both LA and parser
- More efficient compiler
- More portable compiler

Tokens

- Examples of Tokens
  - Operators = + – > ({ := == <>
  - Keywords if while for int double
  - Numeric literals 43 6.035 -3.6e10 0x13F3A
  - Character literals ‘a’ ‘~’ ‘\’’
  - String literals “3.142” “aBcDe” “\”
- Examples of non-tokens
  - White space space(‘ ’) tab(‘\t’) eoln(‘\n’)
  - Comments /*this is not a token*/

Interaction of Lexical analyzer and parser

- Example



How it works

- The Lexical analyzer perform certain other tasks besides identification of tokens. One such task is stripping out comments and *whitespace (blank, newline, tab, and* perhaps other characters that are used to separate tokens in the input).

  Sometimes, lexical analyzers are divided into two processes:

- a) *Scanning consists of the simple processes that do not require tokenization* of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

- b) *Lexical analysis proper is the more complex portion, where the scanner* produces the sequence of tokens as output.

- Type of tokens in C++:
  - Constants:
    - char constants: ‘a’
    - string constants: “I=%d”
    - int constants: 50
    - float point constants
  - Identifiers: i, j, counter, ……
  - Reserved words: main, int, for, …
  - Operators: +, =, ++, /, …
  - Misc. symbols: (,), {, }, …

```
main() {
    int i, j;
    for (I=0; I<50; I++) {
        printf(“I = %d”, I);
    }
}
```

Tokens, Patterns, and Lexemes

- Token: a certain classification of entities of a program.
  - four kinds of tokens in previous example: identifiers, operators, constraints, and punctuation.

- Lexeme: A specific instance of a token. Used to differentiate tokens. For instance, both position and initial belong to the identifier class, however each a different lexeme.

- Patterns: Rule describing how tokens are specified in a program.

Example…cntd

printf (“Total = %d\n”, *score*) ;

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Lexical Errors

fi (a==f(x)) - **fi** is misspelled or keyword? Or undeclared function identifier?

- If **fi** is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler - handle the error

How?

1. Delete one character from the remaining input.

2. Insert a missing character into the remaining input.

3. Replace a character by another character.

4. Transpose two adjacent characters.

Type of Errors

- **Lexical** : name of some identifier typed incorrectly
- **Syntactical** : missing semicolon or unbalanced parenthesis
- **Semantical** : incompatible value assignment
- **Logical** : code not reachable, infinite loop

Errors Recovery Strategies

- Panic mode
- Statement mode
- Error productions
- Global correction

Errors Recovery Strategies(Cont.)

Panic Mode:

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement Mode:

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Errors Recovery Strategies(Cont.)

Error productions:

Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction:

The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.