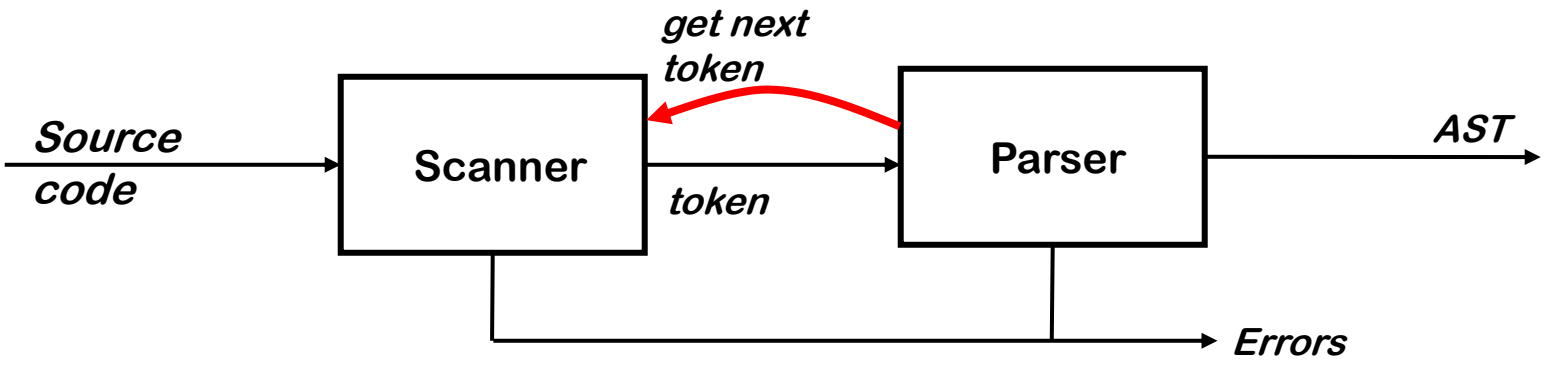
Assignment #1-(1): Scanner

Jeonggeun Kim
Kyungpook National University

Scanner/Parser Interaction



Scanner

- operates as a subroutine that is called by the parser.
- Parser calls "**get next** token"' when it needs a new token from the input stream.
- Unlike Section 2.7 of the Dragon book, we store token information (such as the lexeme) with the token object itself.
  - Instead of using a symbol table.

Tokens (recap from the lecture)

- The tokens of our MiniC language are classified into token types:
- identifiers: `i, j, initial, position, …`
- keywords: `if, for, while, int, float, bool, …`
- operators: `+ - * / <= && …`
- separators: `{ } ( ) [ ] ; ,`
- literals
  - integer literals: `0, 1, 22, …`
  - float literals: `1.25 1. .01 1.2e2 …`
  - bool literals: `true, false`
  - string literals: ``` ``string literals'', ``compiler'', ... ```

Lexemes (Spellings of Tokens)

- The **lexeme of a token**: the character sequence forming the token.

Source code	Token Type	Lexeme
main	ID	main
foobar	ID	foobar
+	plus operator	+
<=	less-equal operator	<=
100	INTLITERAL	100
1.2e2	FLOATLITERAL	1.2e2
true	BOOLLITERAL	true

- Regular expressions (the token specifications for our scanner) will be provided in the MiniC language specification.

Regular expressions for integers and reals in C

Integers:

digit: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

INTLITERAL: digit digit*

Reals:

FLOATLITERAL: digit* fraction exponent?
| digit^{+}.
| digit^{+}.?exponent

fraction: .digit^{+}

exponent: (E|e)(+|-)?digit^{+}

Please refer to the MiniC language specification for details

Source Organization of our MiniC Compiler

- You are provided with a skeleton compiler.
  - Through assignments, we will extend the skeleton to a full compiler.
- Structure of our MiniC Compiler:
  - **Scanner.java:** a skeleton scanner (to be completed by you)
  - **Token.java:** The class for representing all MiniC tokens
    - class Token already provided for you
    - able to distinguish between identifiers and keywords
  - **SourceFile.java:** source-file handling
  - **SourcePos.java:** the class for defining the position of a token in the source file.
  - **MiniC.java:** the driver program of our MiniC compiler. At the moment the driver contains a loop to repeatedly call the scanner for the next token:

```
Token t;
scanner  = new Scanner(source);
scanner.enableDebugging();
do {
    t = scanner.scan(); // scan 1 token
} while (t.kind != Token.EOF);
```

Design Issues in Hand-Crafting your Scanner

- What are the tokens of the language? – see Token.java(.cc and .hh)
- Are keywords reserved? – yes, as in C and Java
- How to distinguish identifiers from keywords? – see Token.java
- How to handle the end of file? – return a special Token: Token.EOF
- How to represent tokens: see Token.java (class Token)
- How to handle whitespace and comments: **throw them away**
- How to structure your scanner? – see Scanner.java
- What to do in case of lexical errors? – see the description of Assignment 1
- How many characters of **look-ahead** are needed to represent a token?
  - Described in the following slides

How to represent a token?

Lexeme	Representation
Counter1	`new Token(Token.ID, "Counter1", src_pos)`
12	`new Token(Token.INTLITERAL, "12", src_pos)`
1.2	`new Token(Token.FLOATLITERAL, "1.2", src_pos)`
+	`new Token(Token.PLUS, "+", src_pos)`
;	`new Token(Token.SEMICOLON, ";", src_pos)`

- **`src_pos`** is an instance of the class **`SourcePos`**:
  - **`StartCol`**: the column in the input file where the token starts
  - **`EndCol`**: the column where the token ends
  - **`StartLine=EndLine`**: number of the line in the input file where the token occurs
    Note: with MiniC, tokens cannot span multiple lines!

The Structure of a Hand-Written Scanner

```
public final class scanner {

  int scanToken (void) {

    // skip whitespace characters and comments...
    // produce the next token:
    switch (currentChar) {
    case '+':
           takeIt() and return token representation for '+'
    case '<':
           takeIt();   // adds '<' to the current lexeme
           if (currentChar == '=') {
              takeIt(); // adds '=' to the current lexeme
              return token representation for '<='
           } else {
              return token representation for '<'
```

Look-ahead

The Structure of a Hand-Written Scanner (cont.)

```
  case '.':
              // attempt to recognize a float:
              ...
  default:
              takeIt();
              return Error token
   }
}
...
return new Token (kind, lexeme, src_pos);
```

- You need to figure out how to efficiently recognize all MiniC tokens:
  - identifiers, keywords, literals, also.

My takeIt() Method in the Scanner Class

```
private void takeIt() {

    currentLexeme.append(currentChar);

    currentChar = sourceFile.readChar();

    // increment line number counter, if necessary

    // increment column counter
}
```

Maintaining 2 Invariants Across Calls to the Scanner

Every time the scanner is called

to return the next token, two invariants hold:

1) **`currentChar`** is pointing either to the beginning
   - of whitespace, or
   - a comment, or
   - a token.
2) The scanner always returns the longest possible match in the remaining input (**maximum munch**):

Def. Invariant: a condition which does not change during program execution.

Note: our two scanner invariants hold only **across** calls.

Input	Tokens
==	"==" and not "=" and "="
//	end-of-line comments, not "/" and "/". Note: we throw away comments in the scanner.

Lexical Errors and Look-Ahead

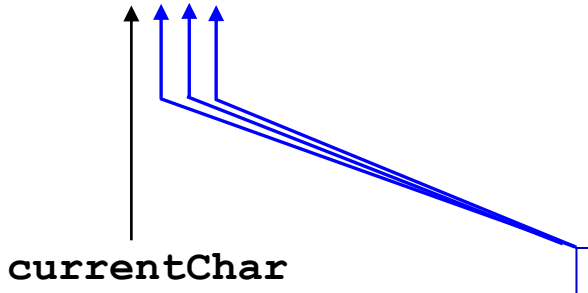
```
FLOATLITERAL        ::=   digit* fraction exponent?
                     |    digit+ .
                     |    digit+ .? exponent
fraction            ::=   . digit+
exponent            ::=   (e|E) (+|-)? digit+
```

- Example floating-point literals:

  2.3 4. .4 2e2 2E4 2.2E+2 2.4e-2 .1E3

- A tricky issue with floating-point literals:

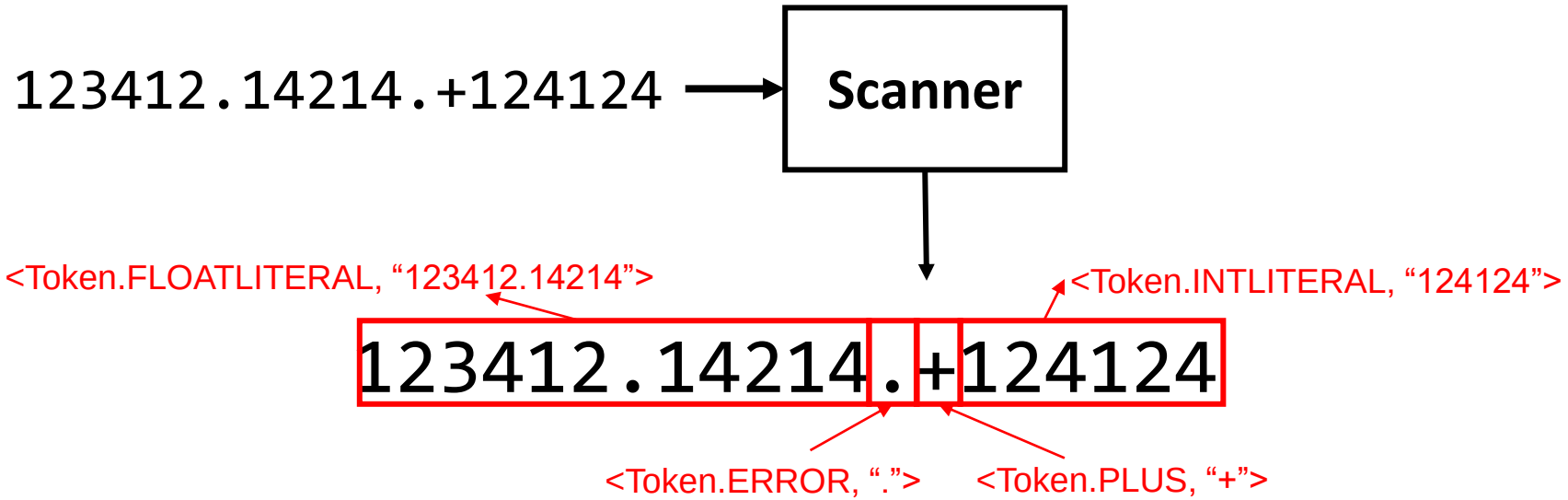
  2.4e+ 2 → "2.4" "e" "+" "2" (four tokens!)



Blue arrows represent look-ahead in the input stream.

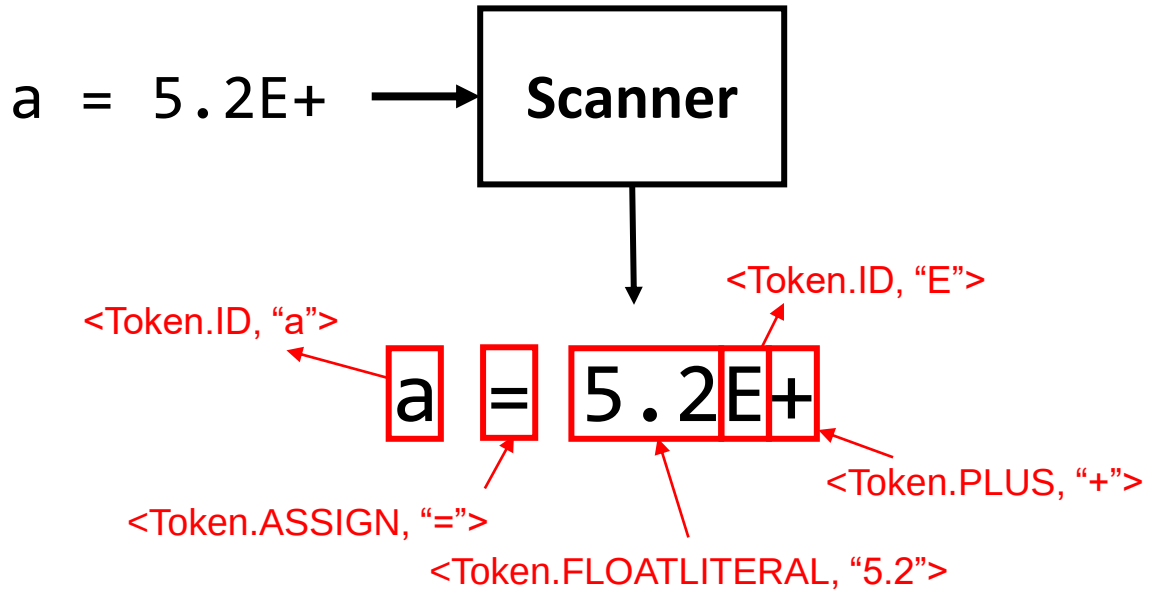
3 characters of look-ahead required to decide that this is "2.4" "e" "+" instead of a floating-point literal that has an exponent (``2.4e+...'').

Maximal Munch



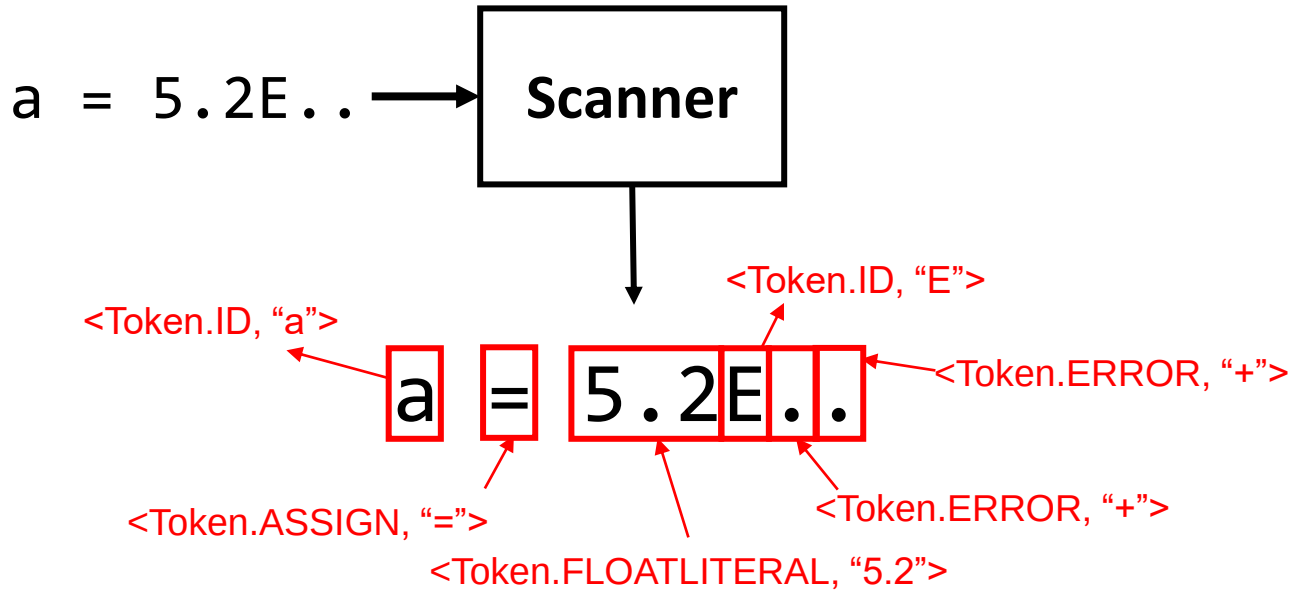
- Our MiniC scanner is going to always **consume the longest possible match (maximal munch)**!
  - **Lexical errors** only occurs in **four cases** (see CCu_Assignment1.pdf)

Maximal Munch



- Our MiniC scanner is going to always **consume the longest possible match (maximal munch)**!
  - **Lexical errors** only occurs in **four cases** (see CCu_Assignment1.pdf)

Maximal Munch



- Our MiniC scanner is going to always **consume the longest possible match (maximal munch)**!
  - **Lexical errors** only occurs in **four cases** (see CCu_Assignment1.pdf)

Longest Match (Maximal Munch)

- Tokens should be built from the maximum possible number of characters from the input stream
  - Eliminates ambiguities in regular expressions