Lexical Analysis
(a.k.a. Scanning)

1

Class Position

- Class `Position` encapsulates the concept of a position in a source file.
  - used primarily for error reporting
- The position is characterized by an ordered pair of integers
  - line number relative to the source file
  - character number relative to that line
- Note: `Position` objects are immutable – once created they can't be modified.
- Key methods

```
public int getLineNumber()
public int getCharNumber()
```


2

Class Source

- Class `Source` is essentially an iterator that steps through the characters in a source file one character at a time. At any point during the iteration you can examine the current character and its position within the source file before advancing to the next character.
- Class `Source`
  - Encapsulates the source file reader
  - Maintains the position of each character in the source file
  - Input: a `Reader` (usually a `FileReader`)
  - Output: individual characters and their position within the file

3

Class Source: Key Methods

```
/**
 * Returns the current character (as an int) in the source
 * file.  Returns EOF if the end of file has been reached.
 */
public int getChar()

/**
 * Returns the position (line number, char number) of the
 * current character in the source file.
 */
public Position getCharPosition()

/**
 * Advance to the next character in the source file.
 */
public void advance() throws IOException
```


4

Testing Class Source

```
String fileName = args[0];
FileReader fileReader = new FileReader(fileName);
Source source = new Source(fileReader);

while (source.getChar() != Source.EOF)
  {
    int c = source.getChar();

    if (c == '\n')
        System.out.print("\\n");
    else if (c != '\r')
        System.out.print((char) c);

    System.out.println("\t" + source.getCharPosition());

    source.advance();
  }
```


5

Results of Testing Class Source
(Input File is `Source.java`)

```
p   line 1, character 1
a   line 1, character 2
c   line 1, character 3
k   line 1, character 4
a   line 1, character 5
g   line 1, character 6
e   line 1, character 7
    line 1, character 8
e   line 1, character 9
d   line 1, character 10
u   line 1, character 11
.   line 1, character 12
c   line 1, character 13
i   line 1, character 14
t   line 1, character 15
a   line 1, character 16
...
```


6

Symbol
(a.k.a. Token Type)

- The term **symbol** will be used to refer to the basic lexical units returned by the scanner. From the perspective of the parser, these are the terminal symbols.
- Symbols include
  - reserved words ("`while`", "`if`", …)
  - operators and punctuation ("`:=`", "`+`", "`;`", …),
  - identifier
  - intLiteral
  - special symbols (EOF, unknown)

7

Enum `Symbol`

```
public enum Symbol
  {
    // reserved words
    BooleanRW("Boolean"),
    IntegerRW("Integer"),
    ...
    whileRW("while"),
    writeRW("write"),
    writelnRW("writeln"),

    // arithmetic operator symbols
    plus("+"),
    minus("-"),
    times("*"),
    divide("/"),
```

(continued on next slide)

8

Enum `Symbol`
(continued)

```
    ...
    // literal values and identifier symbols
    intLiteral("Integer Literal"),
    charLiteral("Character Literal"),
    stringLiteral("String Literal"),
    identifier("Identifier"),

    // special scanning symbols
    EOF("End-of-File"),
    unknown("Unknown");
    ...
}
```

See source file for details.

9

Token

- The term **token** will be used to refer to a symbol together with additional information including
  - the position (line number and character number) of the symbol in the source file
  - the text associated with the symbol
- The additional information provided by a token is used for error reporting, constraint analysis, and code generation, but not to determine if the program is syntactically correct.

10

Examples: Text Associated with Symbols

- "`average`" for an identifier
- "`100`" for an integer literal
- "`Hello, world.`" for a string literal
- "`while`" for the reserved word "`while`"
- "`<=`" for the operator "`<=`"

The text associated with user-defined symbols such as identifiers or literals is more significant than the text associated with language-defined symbols such as reserved words or operators.

11

Class `Token`: Key Methods

```
/**
 * Returns the token's symbol.
 */
public Symbol getSymbol()

/**
 * Returns the token's position within the source file.
 */
public Position getPosition()

/**
 * Returns the string representation for the token.
 */
public String getText()
```


12

Implementing Class Token

Class Token is implemented in two separate classes:

- An abstract, generic class that can be instantiated with any Symbol enum class

```
public abstract class AbstractToken
    <Symbol extends Enum<Symbol>>
```

- A concrete class that instantiates the generic class using the Symbol enum class for CPRL

```
public class Token extends AbstractToken<Symbol>
```

Class AbstractToken is reusable on compiler projects other than a compiler for CPRL.

13

Scanner
(Lexical Analyzer)

- Class Scanner is essentially an iterator that steps through the tokens in a source file one token at a time. At any point during the iteration you can examine the current token, its text, and its position within the source file before advancing to the next token.
- Class Scanner
  - Consumes characters from the source code file as it constructs the tokens
  - Removes extraneous white space and comments
  - Reports any errors
  - Input: Individual characters (from class Source)
  - Output: Tokens (to be consumed by the parser)

14

Class Scanner: Key Methods

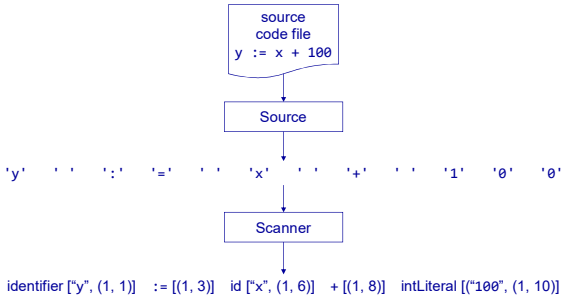
```
/**
 * Returns a copy of the current token in the source file.
 */
public Token getToken()

/**
 * Returns a reference to the current symbol in the source file.
 */
public Symbol getSymbol()

/**
 * Advance to the next token in the source file.
 */
public void advance() throws IOException
```


15

Classes Source and Scanner



16

Method advance()

```
...
try
  {
    skipWhiteSpace();

    // currently at starting character of next token
    position = source.getCharPosition();
    text = null;

    if (source.getChar() == Source.EOF)
      {
        // set symbol but don't advance source
        currentToken.setSymbol(Symbol.EOF);
      }
```

(continued on next page)

17

Method advance()
(continued)

```
else if (Character.isLetter((char) source.getChar()))
  {
    String idString = scanIdentifier();
    symbol = getIdentifierSymbol(idString);

    if (symbol == Symbol.identifier)
        text = idString;
  }
else if (Character.isDigit((char) source.getChar()))
  {
    text   = scanIntegerLiteral();
    symbol = Symbol.intLiteral;
  }
```

(continued on next page)

18

Method advance()
(continued – scanning "+" and "-" symbols)

```
else
  {
    switch((char) source.getChar())
      {
        case '+':
            currentToken.setSymbol(Symbol.plus);
            source.advance();
            break;
        case '-':
            currentToken.setSymbol(Symbol.minus);
            source.advance();
            break;
        ...
```

(continued on next page)

19

Method advance()
(continued – scanning ">" and ">= " symbols)

```
case '>':
    source.advance();
    if ((char) source.getChar() == '=')
      {
        currentToken.setSymbol(Symbol.greaterOrEqual);
        source.advance();
      }
    else
        currentToken.setSymbol(Symbol.greaterThan);
    break;
...
```


20

Example: Scanning an Integer Literal

```
protected String scanIntegerLiteral() throws IOException
  {
    // assumes that source.getChar() is the first digit
    // of the integer literal
    assert Character.isDigit((char) source.getChar()) : " ... ";

    clearScanBuffer();

    do
      {
        scanBuffer.append((char) source.getChar());
        source.advance();
      }
    while (Character.isDigit((char) source.getChar()));

    return scanBuffer.toString();
  }
```

assertion failure error message

21

Tips on Scanning an Identifier

- Use a single method to scan all identifiers, including reserved words.

```
/**
 * Scans characters in the source file for a valid identifier.
 */
protected String scanIdentifier() throws IOException
```

- Use an "efficient" search routine to determine if the identifier is a user-defined identifier or a reserved word.

```
/**
 * Returns the symbol associated with an identifier
 * (Symbol.arrayRW, Symbol.ifRW, Symbol.identifier, etc.)
 */
protected Symbol getIdentifierSymbol(String idString)
```

See handout "Searching for Reserved Words".

22

Lexical Errors

- There are several kinds of errors that can be detected by the scanner when processing a source file. Examples include
  - failure to properly close a character or string literal (e.g., encountering an end-of-line before a closing quote)
  - encountering a character that does not start a valid symbol (e.g., '#' or '@'), etc.
- Scanner method `error()`

```
private ScannerException error(String errorMsg)
  {
    return new ScannerException(getPosition(), errorMsg);
  }
```


23

Handling Lexical Errors in Method advance()

```
catch (ScannerException e)
  {
    ErrorHandler.getInstance().reportError(e);

    // set token to either EOF or unknown
    if (source.getChar() == Source.EOF)
      {
        if (getSymbol() != Symbol.EOF)
            currentToken.setSymbol(Symbol.EOF);
      }
    else
        currentToken.setSymbol(Symbol.unknown);
  }
```


24

Testing Class Scanner

```
String fileName = args[0];
FileReader fileReader = new FileReader(fileName);

Source  source  = new Source(fileReader);
Scanner scanner = new Scanner(source);
Token   token;

do
  {
    token = scanner.getToken();
    printToken(token);
    scanner.advance();
  }
while (token.getSymbol() != Symbol.EOF);
```

 Slide 25

25

Testing Class Scanner
(continued)

```
public static void printToken(Token token)
  {
    System.out.printf("line: %2d   char: %2d   token: ",
        token.getPosition().getLineNumber(),
        token.getPosition().getCharNumber());
    if (   token.getSymbol() == Symbol.identifier
        || token.getSymbol() == Symbol.intLiteral
        || token.getSymbol() == Symbol.stringLiteral
        || token.getSymbol() == Symbol.charLiteral)
        System.out.print(token.getSymbol().toString() + " -> ");
    System.out.println(token.getText());
```

 Slide 26

26

Results of Testing Class Scanner
(Input File is Correct_01.cprl in ScannerTests)

```
line:  2   char:  1   token: and
line:  2   char: 11   token: array
line:  2   char: 21   token: begin
line:  2   char: 31   token: Boolean
...
line:  9   char: 31   token: while
line:  9   char: 41   token: write
line: 10   char:  1   token: writeln
line: 13   char:  1   token: +
line: 13   char:  6   token: -
line: 13   char: 11   token: *
line: 13   char: 16   token: /
line: 16   char:  1   token: =
line: 16   char:  5   token: !=
line: 16   char: 10   token: <
line: 16   char: 14   token: <=
...
```

 Slide 27

27