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.
- Primary constructor

```
class Position(val lineNumber : Int = 0,
               val charNumber : Int = 0)
```


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 Properties and Methods

```
/**
 * The current character (as an Int) in the source file.
 * Property has the value EOF (-1) if the end of file has
 * been reached.
 */
var currentChar = 0

/**
 * The position (line number, char number) of the current
 * character in the source file.
 */
val charPosition : Position

/**
 * Advance to the next character in the source file.
 */
fun advance()
```


4

Testing Class Source

```
val fileName   = args[0]
val fileReader = FileReader(fileName)
val source     = Source(fileReader)

while (source.currentChar != Source.EOF)
  {
    val c = source.currentChar

    if (c == '\n'.toInt())
        print("\\n")
    else if (c != '\r'.toInt())
        print(c.toChar())

    println("\t ${source.charPosition}")
    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 Class `Symbol`

```
enum class Symbol(val label : String)
  {
    // 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 Class `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");

    ...   // helper methods
}
```

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 Properties

```
val symbol   : Symbol

val position : Position

var text     : String
```


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

```
abstract class AbstractToken<Symbol : Enum<Symbol>>
    (val symbol : Symbol,
     val position : Position,
     text : String)
  {
    ...
  }
```

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

13

Implementing Class Token (continued)

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

```
class Token(symbol   : Symbol,
            position : Position,
            text     : String)
    : AbstractToken<Symbol>(symbol, position, text)
```


14

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)

15

Class Scanner: Key Properties and Methods

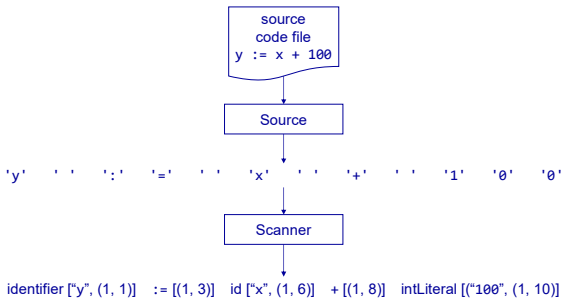
```
/**
 * The current symbol in the source file.
 */
var symbol = Symbol.unknown   // initialized to unknown

/**
 * The current token in the source file.
 */
val token : Token
    get() = Token(symbol, position, text)

/**
 * Advance to the next token in the source file.
 */
fun advance()
```


16

Classes Source and Scanner



17

Method advance()

```
...
try
  {
    skipWhiteSpace()

    // currently at starting character of next token
    position = source.charPosition()
    text = ""

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

(continued on next page)

18

Method advance()
(continued)

```
else if (Character.isLetter(source.currentChar.toChar()))
  {
    val idString = scanIdentifier()
    symbol = getIdentifierSymbol(idString)

    if (symbol == Symbol.identifier)
        text = idString
  }
else if (Character.isDigit(source.currentChar.toChar()))
  {
    text = scanIntegerLiteral()
    symbol = Symbol.intLiteral
  }
```

(continued on next page)

19

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

```
else
  {
    when (source.currentChar.toChar())
      {
        '+'  ->
          {
            symbol = Symbol.plus
            source.advance()
          }
        '-'  ->
          {
            symbol = Symbol.minus
            source.advance()
          }

        ...
```

(continued on next page)

20

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

```
        '>'  ->
          {
            source.advance()
            if (source.currentChar.toChar() == '=')
              {
                symbol = Symbol.greaterOrEqual
                source.advance()
              }
            else
                symbol = Symbol.greaterThan
          }

        ...
```


21

Example: Scanning an Integer Literal

```
private fun scanIntegerLiteral() : String
  {
    // assumes that source.currentChar is the first digit
    // of the integer literal

    clearScanBuffer()

    do
      {
        scanBuffer.append(source.currentChar.toChar())
        source.advance()
      }
    while (Character.isDigit(source.currentChar.toChar()))

    return scanBuffer.toString()
  }
```


22

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.
 */
private fun scanIdentifier() : String
```

- 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".

23

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 fun error(message : String)
                    = ScannerException(position, message)
```


24

Handling Lexical Errors in Method advance()

```
catch (e : ScannerException)
  {
    ErrorHandler.reportError(e)

    // set token to either EOF or UNKNOWN
    symbol = if (source.currentChar == Source.EOF) Symbol.EOF
                                          else Symbol.unknown
  }
```


25

Testing Class Scanner

```
val fileName   = args[0]
val fileReader = FileReader(fileName)

val source  = Source(fileReader)
val scanner = Scanner(source)
var token : Token

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


26

Testing Class Scanner (continued)

```
fun printToken(token : Token)
  {
    System.out.printf("line: %2d   char: %2d   token: ",
        token.position.lineNumber,
        token.position.charNumber)

    val symbol = token.symbol
    if (symbol.isReservedWord())
        print("Reserved Word -> ")
    else if (symbol == Symbol.identifier
          || symbol == Symbol.intLiteral
          || symbol == Symbol.stringLiteral
          || symbol == Symbol.charLiteral)
        print(token.symbol.toString() + " -> ")

    println(token.text)
  }
```


27

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

```
line:  2   char:  1   token: Reserved Word -> and
line:  2   char: 11   token: Reserved Word -> array
line:  2   char: 21   token: Reserved Word -> begin
line:  2   char: 31   token: Reserved Word -> Boolean
...
line:  9   char: 31   token: Reserved Word -> while
line:  9   char: 41   token: Reserved Word -> write
line: 10   char:  1   token: Reserved Word -> 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: <=
...
```


28