5.5 Language Elements Seen So Far

Chapters 4 and 5 have covered Scala’s language elements to express expressions and types comprising of primitive data and functions. The context-free syntax of these language elements is given below in extended Backus-Naur form, where ‘|’ denotes alternatives, [ ... ] denotes option (0 or 1 occurrence), and { ... } denotes repetition (0 or more occurrences).

Characters

Scala programs are sequences of (Unicode) characters. We distinguish the following character sets:

- whitespace, such as ‘ ’, tabulator, or newline characters,
- letters ‘a’ to ‘z’, ‘A’ to ‘Z’,
- digits ‘0’ to ‘9’,
- the delimiter characters . , ; ( ) { } [ ] \ " '

- operator characters, such as ‘#’ ‘+’, ‘:’. Essentially, these are printable characters which are in none of the character sets above.

Lexemes:

```
ident   =  letter {letter | digit}  
       |  operator { operator }  
       |  ident ' _ ' ident
literal =  "as in Java"
```

Literals are as in Java. They define numbers, characters, strings, or boolean values. Examples of literals as 0, 1.0e10, 'x', "he said "hi!"", or true.

Identifiers can be of two forms. They either start with a letter, which is followed by a (possibly empty) sequence of letters or symbols, or they start with an operator character, which is followed by a (possibly empty) sequence of operator characters. Both forms of identifiers may contain underscore characters ‘_’. Furthermore, an underscore character may be followed by either sort of identifier. Hence, the following are all legal identifiers:

```
x    Room10a    +    --    foldl_:    _vector
```

It follows from this rule that subsequent operator-identifiers need to be separated by whitespace. For instance, the input x+y is parsed as the three token sequence x,
+-, y. If we want to express the sum of x with the negated value of y, we need to add at least one space, e.g. x + -y.

The $ character is reserved for compiler-generated identifiers; it should not be used in source programs.

The following are reserved words, they may not be used as identifiers:

<table>
<thead>
<tr>
<th>abstract</th>
<th>case</th>
<th>catch</th>
<th>class</th>
<th>def</th>
</tr>
</thead>
<tbody>
<tr>
<td>do</td>
<td>else</td>
<td>extends</td>
<td>false</td>
<td>final</td>
</tr>
<tr>
<td>finally</td>
<td>for</td>
<td>if</td>
<td>implicit</td>
<td>import</td>
</tr>
<tr>
<td>match</td>
<td>new</td>
<td>null</td>
<td>object</td>
<td>override</td>
</tr>
<tr>
<td>package</td>
<td>private</td>
<td>protected</td>
<td>requires</td>
<td>return</td>
</tr>
<tr>
<td>sealed</td>
<td>super</td>
<td>this</td>
<td>throw</td>
<td>trait</td>
</tr>
<tr>
<td>try</td>
<td>true</td>
<td>type</td>
<td>val</td>
<td>var</td>
</tr>
<tr>
<td>while</td>
<td>with</td>
<td>yield</td>
<td>-</td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>=&gt;</td>
<td>&lt;-</td>
<td>&lt;%</td>
</tr>
</tbody>
</table>

Types:

```
Type = SimpleType | FunctionType
FunctionType = SimpleType => Type | '(' [Types] ')'=> Type
SimpleType = Byte | Short | Char | Int | Long | Float | Double | Boolean | Unit | String
Types = Type {',' Type}
```

Types can be:

- number types Byte, Short, Char, Int, Long, Float and Double (these are as in Java),
- the type Boolean with values true and false,
- the type Unit with the only value ()
- the type String,
- function types such as (Int, Int) => Int or String => Int => String.

Expressions:

```
Expr = InfixExpr | FunctionExpr | if '(' Expr ')' Expr else Expr
InfixExpr = PrefixExpr | InfixExpr Operator InfixExpr
Operator = ['+' | '-' | '!' | '~'] SimpleExpr
PrefixExpr = ['+' | '-' | '!' | '~'] SimpleExpr
SimpleExpr = ident | literal | SimpleExpr '.' ident | Block
FunctionExpr = (Bindings | Id) '=>' Expr
Bindings = '(' Binding {',' Binding} ')' 
Binding = ident [': Type]
Block = '{' {Def ';'} Expr '}'
```
Expressions can be:

- identifiers such as `x`, `isGoodEnough`, `*`, or `+-`,
- literals, such as `0`, `1.0`, or "abc",
- field and method selections, such as `System.out.println`,
- function applications, such as `sqrt(x)`,
- operator applications, such as `-x` or `y + x`,
- conditionals, such as `if (x < 0) -x else x`,
- blocks, such as `{ val x = abs(y) ; x * 2 }`,
- anonymous functions, such as `x => x + 1` or `(x: Int, y: Int) => x + y`.

Definitions:

```
Def      = FunDef  |  ValDef
FunDef   = 'def' ident {(' [Parameters] ')'} [':'] Type '=' Expr
ValDef   = 'val' ident [':'] Type '=' Expr
Parameters = Parameter {',' Parameter}
Parameter  = ident ':' ['=>'] Type
```

Definitions can be:

- function definitions such as `def square(x: Int): Int = x * x`,
- value definitions such as `val y = square(2)`. 