6.1 Expressions

Description

An expression is a combination of one or more constants, variables, operators and functions in the form of method invocations. The *evaluation* of an expressions yields a *datum*, which may be a value or a reference. We use the term evaluation for execution of an expression.

Value expressions

One kind of expression is a sequence of digits representing a number like 10, -12, 3.14, and 1.3E4. The numbers 10 and -12 represent integer numbers whereas 3.14 and 1.3E4 represent float numbers where a dot ('.') may be used to specify the decimal point and the 'E' specifies a possible exponential art. A representation of a number in this way is often called a *literal* .

Evaluation of a number like 10 of course yields the value 10, and evaluation of 3.14 of course yields the corresponding float value.

You may combine numbers into *compound expressions* using binary operators as in: 111 + 23, 3 * 12, and 0.1 * 111.12. Evaluation of such a compound expression consists of applying the operator to its operands: 111 + 23 thus yields the value 134.

A data-item representing a number may also be used as an expression. If `anInt` is a constant or variable integer, then `anInt` may be used in an expression that evaluates to the value that `anInt` is currently holding.

A method invocation returning a value may also be used as an expression. An example is `anAccount.withdraw(140)`. Here `withdraw` returns the value of the `balance` of `anAccount` after withdrawal of 140.

Expressions may in general be combined into arbitrary expressions like `anInt * 10 + max(a,b) * 4`, where max is a method returning the larger of its two arguments.

For compound expressions the order of evaluation of is subexpressions is important. Consider `3 + 5 * 2`. The standard way to evaluate this expression is to evaluate `5 * 2` and add the result to 3 yielding `13`. We thus have to be aware of *priorities* between operators.

The operator `^`(exponentiation) has highest priority, then comes multiplication (`*`) and division (`/`), and addition (`+`) and subtraction (`-`) have the lowest priority.

For operators of the same priority one must be aware of whether an expression is evaluated from left to right or right to left. For one operators like + and * it does not make a difference, but an operator like – it makes a difference. For an expression `10-6-3`; if you evaluate from left you get `(10-6)-3 = 1` whereas from right you get `10-(6-3) = 7.`

Parentheses may be used to control the evaluation of subexpressions as in: `(3 + 5) * 2`. Here the parentheses means that `3 + 5` is evaluated yielding the value 8 which then is multiplied to 2 yielding `16`. For a newcomer, it may be a good idea to use extra parentheses if in doubt about how a givne expression is evaluated.

The rules for priority, associativity and parentheses we use in this book are the ones generally adapted mathematics and most programming languages.

An expression has a *type* being the type of the value yielded by evaluation of the expression. In the above examples, we have seen expressions of type integer and float.

The above kind of expressions using integer numbers and float numbers are called arithmetic expression.

We also have *relational expressions* like `a < b`. The *relational operator* `'<'` tests whether or not a it is less than `b`. It

evaluates to the Boolean value `true` or `false`.

A third kind of expressions are boolean expressions where the operands are boolean values and the operators are logical operators like `&&` (and), `||` (or), etc.

The condition of a conditional statement like if-then must be a boolean expression as shown here:

```
if ((a < 10) && (b = 100)) :then
   ...
```

The expression (`a < 10`) evaluates to one of the Boolean values `true` or `false` and so do (`b = 100`). The Boolean results of these two expressions are then evaluated using the Boolean and-operator `&&`, which also yields either `true` or `false`.

Operators

The table below show the possible operators that may be used in expressions:

Arithmetic operators	+, -, *, /, ^ (exponentiation)
Relational operators	=, <>, <=, <, >, >=
Boolean operators	&&, ||, not
Operators	

The above expression consists of numbers and operators with numbers as operands. Characters and Strings may also be used as expressions and the resulting types is then of type char or String.

Character literals like ‘a’, ‘9’, ‘?’, etc. may also be used as expressions yielding the characters ‘a’, ‘9’, ‘?’, etc. – as of no surprise.

A *String literal* like `"Hello"` may be used as an expression and `"Hello " + "world!"` is an expression yielding the string `"Hello world!"`.

Reference expressions

An expression may also yield a reference as explained in section 5.1, but we repeat it here for completeness. One form of a reference expression is the name of a data-item declared as a reference using **`obj`** or **`ref`**:

```
c1: ref Customer
c2: obj Customer
c1 := c2
```

Here the `c2` on the right-side of the assignment is an example of such a reference expression. And it of course evaluates to a reference to the `Customer` object currently referred to by `c2`. After the assignment, `c1` refers to the same `Customer` object as `c2`.

Another form of a reference expression is instantiation of an object:

```
c1 := Customer("John Smith")
```

Here `Customer("John Smith")` is an example of a reference expression that when evaluated generates a new `Customer` object and the resulting value is a reference to this new object.

The `this` reference

Suppose that we want to notify the owner of an `Account` whenever there has been a transaction on his/her account. To represent this, we may add a notify method to class `Customer`:

```
class Customer(name: var String):
   -"-
```

```
   notify(acc: ref Account, amount: var float):
      ...
```

Notify has two parameters:

- `acc` is a reference to the `Account` where there has been a transaction.
- `amount` is the value that has been deposited or withdrawn. A positive value for `amount` represents a `deposit` and a negative value represents a `withdraw`.

We may then invoke this method from `withdraw` and `deposit` t using an invocation like:

```
owner.notify(theAccount,amount)
```

where `theAccount` is supposed to be a reference to the `Account` object where `deposit` or `withdraw` has been `invoked`. But so far, we have no such reference.

We may, however, use the expression `this(Account)` within `deposit` and withdraw. It evaluates to a reference to the enclosing `Account` object. We may thus use `this(Account)` in class `Account`:

```
class Account(owner: ref Customer):
   -"-
   deposit(amount: var float):
      -"-
      owner.notify(this(Account),amount)
   withdraw(amount: var float):
      -"-
      owner.notify(this(Account),- amount)
```

Object instantiation and method invocations

In general object instantiation/generation and method invocations may be elements of an expression. The previous section has an example of an object generation using `Customer("John Smith")`.

A method invocation as an expression is shown here:

```
newBalance: var float
newBalance := anAccount.withdraw(200)
```

Here `anAccount.withdraw(200)` is an example of a method invocation used as an expression returning the new balance of the `anAccount`.

For further descriptions of object instantiation and method invocation, see chapter and chapter .