Algorithms and Functions

Algorithm Sections

Whereas equations are very well suited for physical modeling, there are situations where computations are more conveniently expressed as algorithms, i.e., sequences of instructions, also called statements

```
algorithm
  ...
  <statements>
  ...
<some keyword>
```

Algorithm sections can be embedded among equation sections

```
equation
  x = y*2;
  z = w;
algorithm
  x1 := z+x;
  x2 := y-5;
  x1 := x2+y;
equation
  u = x1+x2;
  ...
```


Iteration Using for-statements in Algorithm Sections

```
for <iteration-variable> in <iteration-set-expression> loop
    <statement1>
    <statement2>
    ...
end for
```

The general structure of a `for`-statement with a single iterator

```
class SumZ
  parameter Integer n = 5;
  Real[n]  z(start = {10,20,30,40,50});
  Real sum;
algorithm
  sum := 0;
  for i in 1:n loop      // 1:5 is {1,2,3,4,5}
    sum := sum + z[i];
  end for;
end SumZ;
```

A simple `for`-loop summing the five elements of the vector `z`, within the class `SumZ`

Examples of `for`-loop headers with different range expressions

```
for k in 1:10+2 loop           // k takes the values 1,2,3,...,12
for i in {1,3,6,7} loop        // i takes the values 1, 3, 6, 7
for r in 1.0 : 1.5 : 5.5 loop  // r takes the values 1.0, 2.5, 4.0, 5.5
```


Iterations Using while-statements in Algorithm Sections

```
while <conditions> loop
  <statements>
end while;
```

The general structure of a `while`-loop with a single iterator.

```
class SumSeries
  parameter Real eps = 1.E-6;
  Integer i;
  Real sum;
  Real delta;
algorithm
  i := 1;
  delta := exp(-0.01*i);
  while delta>=eps loop
    sum := sum + delta;
    i := i+1;
    delta := exp(-0.01*i);
  end while;
end SumSeries;
```

The example class `SumSeries` shows the `while`-loop construct used for summing a series of exponential terms until the loop condition is violated , i.e., the terms become smaller than eps.

if-statements

```
if <condition> then
  <statements>
elseif <condition> then
  <statements>
else
  <statementss>
end if
```

The general structure of `if`-statements.
The `elseif`-part is optional and can occur zero or more times whereas the optional `else`-part can occur at most once

The `if`-statements used in the class `SumVector` perform a combined summation and computation on a vector `v`.

```
class SumVector
  Real sum;
  parameter Real v[5] = {100,200,-300,400,500};
  parameter Integer n = size(v,1);
algorithm
  sum := 0;
  for i in 1:n loop
    if v[i]>0 then
      sum := sum + v[i];
    elseif v[i] > -1 then
      sum := sum + v[i] -1;
    else
      sum := sum - v[i];
    end if;
  end for;
end SumVector;
```


when-statements

```
when <conditions> then
  <statements>
elsewhen <conditions> then
  <statements>
end when;
```

when-statements are used to express actions (statements) that are only executed at events, e.g. at discontinuities or when certain conditions become true

There are situations where several assignment statements within the same when-statement is convenient

```
when x > 2 then
  y1 := sin(x);
  y3 := 2*x + y1 + y2;
end when;
```

```
algorithm
  when x > 2 then
    y1 := sin(x);
  end when;
equation
  y2 = sin(y1);
algorithm
  when x > 2 then
    y3 := 2*x + y1 + y2;
  end when;
```

```
when {x > 2, sample(0,2), x < 5} then
  y1 := sin(x);
  y3 := 2*x + y1 + y2;
end when;
```

Algorithm and equation sections can be interleaved.

Function Declaration

The structure of a typical function declaration is as follows:

```
function <functionname>
  input  TypeI1 in1;
  input  TypeI2 in2;
  input  TypeI3 in3;
  ...
  output TypeO1 out1;
  output TypeO2 out2;
  ...
protected
  <local variables>
  ...
algorithm
  ...
  <statements>
  ...
end <functionname>;
```

All internal parts of a function are optional, the following is also a legal function:

```
function <functionname>
end <functionname>;
```

Modelica functions are *declarative mathematical functions*:

- Always return the same result(s) given the same input argument values

Function Call

Two basic forms of arguments in Modelica function calls:

- *Positional* association of actual arguments to formal parameters
- *Named* association of actual arguments to formal parameters

Example function called on next page:

```
function PolynomialEvaluator
  input  Real A[:];    // array, size defined
                       // at function call time
  input  Real x := 1.0;// default value 1.0 for x
  output Real sum;
protected
  Real   xpower;       // local variable xpower
algorithm
  sum := 0;
  xpower := 1;
  for i in 1:size(A,1) loop
    sum := sum + A[i]*xpower;
    xpower := xpower*x;
  end for;
end PolynomialEvaluator;
```

The function `PolynomialEvaluator` computes the value of a polynomial given two arguments:
a coefficient vector `A` and a value of `x`.

Positional and Named Argument Association

Using *positional* association, in the call below the actual argument `{1,2,3,4}` becomes the value of the coefficient vector `A`, and `21` becomes the value of the formal parameter `x`.

```
...
algorithm
  ...
  p:= polynomialEvaluator({1,2,3,4},21)
```

The same call to the function `polynomialEvaluator` can instead be made using *named* association of actual parameters to formal parameters.

```
...
algorithm
  ...
  p:= polynomialEvaluator(A={1,2,3,4},x=21)
```


Functions with Multiple Results

```
function PointOnCircle"Computes cartesian coordinates of point"
  input  Real angle  "angle in radians";
  input  Real radius;
  output Real x;    // 1:st result formal parameter
  output Real y;    // 2:nd result formal parameter
algorithm
  x := radius * cos(phi);
  y := radius * sin(phi);
end PointOnCircle;
```

Example calls:

```
(out1,out2,out3,...) = function_name(in1, in2, in3, in4, ...);  // Equation
(out1,out2,out3,...) := function_name(in1, in2, in3, in4, ...); // Statement

(px,py)  = PointOnCircle(1.2, 2);   // Equation form
(px,py) := PointOnCircle(1.2, 2);   // Statement form
```

Any kind of variable of compatible type is allowed in the parenthesized list on the left hand side, e.g. even array elements:

```
(arr[1],arr[2]) := PointOnCircle(1.2, 2);
```


External Functions

It is possible to call functions defined outside the Modelica language, implemented in C or FORTRAN 77

```
function polynomialMultiply
  input  Real a[:], b[:];
  output Real c[:] := zeros(size(a,1)+size(b, 1) - 1);
external
end polynomialMultiply;
```

The body of an external function is marked with the keyword **`external`**

If no language is specified, the implementation language for the external function is assumed to be C. The external function `polynomialMultiply` can also be specified, e.g. via a mapping to a FORTRAN 77 function:

```
function polynomialMultiply
  input  Real a[:], b[:];
  output Real c[:] := zeros(size(a,1)+size(b, 1) - 1);
external "FORTRAN 77"
end polynomialMultiply;
```

