3 Generalized Operator Declarations

3.1 Syntax of a General Operator Declaration

```
operdecl : [+|\] sig (->|=>|->>|=>>) ( [ type ] [ = expr ] )

sig      : part { part }

part     : name
         | pardecl
         | [ < names > ] [ sig { | sig } ]
         | [ < names > ] { sig { | sig } }
         | [ < names > ] ( sig | sig { | sig } )
         | < names > ( sig )
         | expr

pardecl  : ( [ names : ] [ type ] [ = expr ] )

names    : name { name }
```

3.2 Brackets in the Signature of an Operator

- Square brackets in the signature of an operator declaration denote *optional* syntax parts which might be omitted in an application of the operator.
- Curly brackets denote *repeatable* syntax parts which might appear any number of times (zero or more) in an application of the operator.
- Round brackets with vertical bars denote *alternative* syntax parts where exactly one of the alternatives must appear in an application of the operator.
- In fact, square and curly brackets might also contain multiple alternatives where at most one (for square brackets) or any number of successive alternatives (for curly brackets) might appear in an application of the operator.
- Normally, round brackets must contain at least two alternatives.
  (Round brackets with just one alternative are described in § 3.10.)
- Each of these brackets can have an optional *denomination* consisting of one or more names (cf. § 2.4) in angle brackets before the opening bracket.

3.3 Corresponding Bracket Operators

- For every bracket described in § 3.2 in the signature of an operator, there is a corresponding *bracket operator* having operands corresponding to the alternatives of the bracket, which are also separated by vertical bars.
  For a square bracket, the bracket operator also has an additional optional operand.

- No matter whether the denomination of a bracket operator mentioned in § 3.2 is present or not, an application of such an operator might always optionally start with angle brackets which are (even if names are present) either empty (and then only have a certain kind of signaling effect) or contain the names of the bracket.

- If the i-th alternative of a round or square bracket has been traversed in an application of an operator, the i-th operand of the corresponding bracket operator is evaluated and its value is returned as the result of an application of the bracket operator.
  Therefore, all operands of the bracket operator must have the same type here.

- If a square bracket has not been traversed at all, the optional operand of the corresponding bracket operator is evaluated and its value is returned as the result, if it is present; otherwise, the bracket operator returns nil in that case.

- If the alternatives $i_1, \ldots, i_n$ of a curly bracket have been traversed successively, the operands $i_1, \ldots, i_n$ of the corresponding bracket operator are evaluated successively. The result value is the number of passes, i. e., n.
  Therefore, the operands of the bracket operator can also have different types here.

- Bracket operators can be used any number of times, in any order, and arbitrarily nested.

3.4 Visibility of Parameters and Bracket Operators

- The parameter declared by a parameter declaration `pardecl` is visible in the subsequent part of the (sub)signature `sig` to which the declaration belongs, that means:
  - in the subsequent parameter declarations of this (sub)signature and therefore in the type `type` and in the default expression `expr` (see below) of these parameters;
  - in the subsequent brackets of this (sub)signature and therefore indirectly also in the parameter declarations and nested brackets of the subsignatures of these brackets.
- Likewise, the bracket operator belonging to a particular bracket is visible in the subsequent part of the (sub)signature to which the bracket belongs.
- Parameters and bracket operators coming from a *simple option bracket*, i. e., a square bracket with just one alternative, are also visible in the subsequent part of the (sub)signature to which this square bracket belongs.

- In the result type and in the implementation of an operator, only those parameters and bracket operators are visible which are visible at the end of its (top level) signature.

- In the *i*-th operand of a bracket operator, the parameters and bracket operators which are visible at the end of the *i*-th subsignature of the corresponding bracket are also visible.

- By these rules, parameters are normally only visible and usable if the (sub)signature to which they belong has actually been traversed in an application of the operator.

- However, by means of the above additional rule for simple option brackets, parameters coming from such brackets are also visible if the bracket has not been traversed.
  In that case, such a parameter has a *default value* which is obtained by evaluating the *default expression* `expr` given in its declaration. If there is no such default expression, the default value is nil.

- Even though default expressions are useful only for parameters in simple option brackets, other parameters might also have default expressions which are never evaluated, however.

3.5 Examples

3.5.1 Optional Syntax Parts

- The operator •+!• increments the value of the variable `x` by `y`.
  If there is no operand belonging to the parameter `y` in an application of the operator, it has the default value `1`:

```
(x:int?) "+!" [ (y:int=1) ] -> (int = x =! ?x + y);
```

```
x : int?;
x =! 0;
x +! 5;
x +!;
print ?x    $$ 6
```

3.5.2 Repeatable Syntax Parts

- The operator `sum` computes the sum of any number of values:

```
sum of (x:int) { and (y:int) } end -> (int =
  s : int?;
  s =! x;
  { s +! y };
  ?s
);
```

```
sum of 1 end;
sum of 1 and 2 end;
sum of 1 and 2 and 3 end
```

Without the terminating `end`, expressions such as `sum of 1 * 2` would be ambiguous.

3.5.3 Alternative Syntax Parts

- Because the predefined operators for addition and subtraction have the same binding properties, they are actually defined as a single operator which computes either the sum or the difference of the `int` values `x` and `y`:

```
(x:int) ("+"|"-") (y:int) -> (int = ......);
                                1 + 2;
                                1 - 2
```

- Likewise for multiplication, division, and remainder:

```
(x:int) ("*"|":"|"-:-") (y:int) -> (int = ......);
                                5 * 3;
                                5 : 3;
                                5 -:- 3
```

- The user-defined operator `calc` also computes either the sum or the difference of the `int` values `x` and `y`:

```
calc (x:int) (plus|minus) (y:int) end -> (int =
  (x + y | x - y)
);
                                calc 1 plus 2 end;
                                calc 1 minus 2 end
```

3.5.4 Combination of Different Brackets

- This operator `calc` computes arbitrary sums and differences:

```
calc [minus] (x:int) { (plus|minus) (y:int) } end -> (int =
  r : int?;
  r =! [-x | x];
  { (r +! y | r -! y) };
  ?r
);
```

```
calc 1 plus 2 minus 3 end;
calc minus 1 plus 2 end
```

- The predefined operator `print` prints either an `int` value `x` or a `char` value `y` and – if `only` is not present – a terminating line separator:

```
print [only] ((x:int) | (y:char)) -> (bool = ......);
```

```
print 1 + 2 * 3;
print only 1 + 2 * 3;
print 'x';
print only 'x'
```

- The operator `cnf` determines the truth value of a logic formula written in conjunctive normal form (CNF, i. e., a conjunction of disjunctions) by using the logic operators `/•` (negation), `•/\•` (conjunction), and `•\/•` (disjunction), which are defined in a task sheet:

```
cnf [not] (a:bool) { or <nb>[not] (b:bool) }
{ and <nc>[not] (c:bool) { or <nd>[not] (d:bool) } } end
-> (bool =
  res : bool?;
  res =! [/ a | a];
  { res =! ?res \/ <nb>[/ b | b] };
  { tmp : bool?;
    tmp =! <nc>[/ c | c];
    { tmp =! ?tmp \/ <nd>[/ d | d] };
    res =! ?res /\ ?tmp
  };
  ?res
);
```

```
cnf u or v and not x end;
cnf not u and v or not w and x end
```

Even though the curly brackets do not have denominations, the use of the corresponding bracket operators is unambiguous because of the parameters used in their operands.

3.6 Exclusion of the Predefined Sequential Evaluation

- The rule mentioned in § 2.9.1 about the implicit exclusion of the predefined sequential evaluation •`;`• is generalized as follows:

- If the *operands* belonging to a particular parameter of an operator *can* appear, in any application of the operator, before the first or after the last name of this expression, then the exclude set of the parameter implicitly contains the predefined sequential evaluation, even if the parameter itself does not appear before the first or after the last name of the operator in its signature.

- For example: If the operator `sum` from § 3.5.2 would have been defined without the terminating `end`, then the sequential evaluation would be excluded for both parameters, `x` and `y`, because the operands belonging to `x` can also appear after the last name of the respective expression (e. g., `sum of 1`), even though `x` appears before the name `and` in the operator's signature:

```
sum of (x:int) { and (y:int) } -> (int = ......);
                                  sum of 1
                                  sum of 1 and 2
                                  sum of 1 and 2 and 3
```

3.7 Deducible Parameters and Generic Operators

- If a parameter is defined in a simple option bracket, it can be used, according to § 3.4, in particular in the type of successive parameter declarations.
- In such a case, the assignment of such an optional parameter (i. e., the corresponding operand if no explicit operand has been given) can normally be *deduced* from the type of the operand belonging to the successive parameter.
- Therefore, such *deducible parameters* are comparable, for example, with template parameters in C++ and can therefore be used to define *generic* operators.
- If a deducible parameter appears in multiple successive parameters, the same assignment must result from all corresponding operands.
- Differing from the rules in § 3.4, a default value is never assigned to a deducible parameter; if its assignment can not or not unambiguously be deduced, the operator application is erroneous.
- Even though deducible parameters frequently have type `type`, i. e., they denote types, they can also have other types in principle.

3.7.1 Examples of Predefined Generic Operators

Loop

```
"?*" [(X:type)] (x:X) -> (int = ......)
```

Or:

```
[(X:type)] "?*" (x:X) -> (int = ......)
```

Or:

```
"?*" [ "[" (X:type) "]" ] (x:X) -> (int = ......)
```

- A deducible parameter can be declared anywhere in the signature before its first use in the type of another parameter.
- For all predefined generic operators the rule applies, that each deducible parameter appears as late as possible in the signature; furthermore, it is surrounded by “literal” square brackets which allows for more efficient processing of expressions by the compiler.
- Therefore, the predefined loop operator is actually defined in the third way shown above.

Branch

```
[ "[" (X:type) "]" ] (x:X) "?"
[ "[" (YZ:type) "]" ] (y:YZ) "!" (z:YZ) -> (YZ = ......)
```

- Because the condition `x` might have a type different from the operands `y` and `z`, while these operands must have the same type, there must be a deducible parameter `X` for `x` and another deducible parameter `YZ` for `y` and `z`, which is also used in the result type of the operator.

Remark

- The branch and loop operators evaluate their operands only if required. This can only be expressed with lambda parameters (cf. § 3.8) and is therefore omitted in the above examples.

3.7.2 Examples of User-Defined Generic Operators

Swapping Variable Contents

```
[(T:type)] (x:T?) "<->" (y:T?) -> (T? =
  z := ?x;
  x =! ?y;
  y =! z;
  y
);

b1 : bool?; b1 =! true;          i1 : int?; i1 =! 1;
b2 : bool?; b2 =! false;         i2 : int?; i2 =! 2;
b1 <-> b2;                       i1 <-> i2;


b1 <-> i2        $$ Error, because of different assignments for T.
```

Conversion of Arbitrary Types to `bool`

```
bool [(T:type)] (x:T) -> (bool = x ? true ! false);

bool 1;
bool 'x';


bool nil        $$ Eror, because neither the type of nil nor the
                $$ assignment of T can be unambiguously deduced
```

Chained Equality/Inequality Test

```
[(T:type)] (x:T) ("="|"/=") (y:T) { ("="|"/=") (z:T) }
-> (bool = ......)
```

- Because all operands must have the same type here, there is only one deducible parameter `T` for all other parameters.

Conjunctive Normal Form with Arbitrary Operand Types (cf. § 3.5.4)

```
cnf <na>[not] [(A:type)] (a:A) { or <nb>[not] [(B:type)] (b:B) }
{ and <nc>[not] [(C:type)] (c:C)
                    { or <nd>[not] [(D:type)] (d:D) } } end
-> (bool = .......)
```

- Because every operand can have a different type here, there is a distinct deducible parameter for each of the parameters `a` to `d`.
- Because the parameters `B` to `D` are each defined inside curly brackets, they can also be assigned a different type in every pass through the bracket.
- Differing from § 3.5.4, the first square bracket `<na>[NOT]` must be named here, because there is another square bracket `[(A:type)]` in the top level signature and, therefore, the use of the corresponding bracket operator in the implementation would be ambiguous without the denomination.

3.7.3 Deduction from the Result Type

- If a parameter appears in the result type of an operator, its assignment can (in contrast to, e. g., C++) normally be deduced from the context of the respective operator application.

- Example: Predefined operator `nil`

```
nil [ "[" (T:type) "]" ] -> (T = var : T?; ?var);

                          $$ T is deduced as:

1 + nil;                  $$ int

x : char;
x = nil ? ... ! ...;      $$ char

... ? nil ! 1;            $$ int

print nil;                $$ int or char => ambiguity
nil = nil                 $$ arbitrary => error
```

- Example: Operator `uniq`, that can be used analogously to `nil`:

```
uniq [(T:type)] -> (T = const : T)
```

3.8 Unevaluated Operands and Lambda Parameters

3.8.1 Basic Principle

- Normally, when a user-defined operator is applied, all operands are evaluated from left to right, and the corresponding parameters are initialized with the resulting values (cf. § 2.8).

- In order to define control flow operators such as branches and loops, it must be possible, however, to pass at least some operands *unevaluated* to allow the implementation of the operator to decide whether and possibly when and how often an operand should be evaluated.

- For that purpose, the corresponding parameters can be defined as *lambda parameters*, which can then be used like operators whose implementation expression is the respective operand.

- For example:

```
if [(X:type)] (x:X) then
[(YZ:type)] (\ y -> (YZ)) else (\ z -> (YZ)) end -> (YZ =
    x ? y ! z
)
```

Explanations

- The result value of an operator declaration is not the declared operator (which is simply available after the declaration), but rather the type of this operator, i. e., the result type of an operator declaration is `type`.
- According to § 3.1, the names of the parameters as well as the implementation of the operator can be omitted in an operator declaration. (The parameters are anonymous in that case.) Furthermore, an operator declaration can optionally start with a plus sign or a backslash.
- Therefore, `\ y -> (YZ)` (and likewise `\ z -> (YZ))` is basically a correct operator declaration (if there is a type `YZ` at the respective location which is the case in the example above) and, therefore, also a correct (operator) type.
- This implies in turn, that (`\ y -> (YZ))` is a correct declaration of an anonymous parameter of the `if` operator. The operator defined by this declaration is visible everywhere where this anonymous parameter would be visible.
- In contrast to normal operator declarations, where a leading plus sign or backslash is meaningless, a backslash (having a certain similarity to the Greek letter λ) at the begin of a parameter declaration marks the parameter as a lambda parameter.

- Because, as already mentioned, the operands belonging to a lambda parameter are used as the implementation of the operator defined by the lambda parameter, the type of such an operand must not be equal to the (operator) type of the parameter itself, but to its result type.
- Therefore, the operands belonging to the lambda parameters `y` and `z` can be expressions with any type `YZ`, which are not yet evaluated in an application of the `if` operator.
- Only when an application of one of the operators `y` or `z` is evaluated during the evaluation of the implementation of the `if` operator, the respective operand is evaluated as the implementation of the respective operator.

Example

```
if ... then
    print 1
else
    print 0
end
```

- Because `x` is an ordinary parameter of the `if` operator, the corresponding operand `...` is evaluated immediately in the application of this operator.
- But because `y` and `z` are lambda parameters, the corresponding operands `print 1` and `print 0` are not evaluated, but used as the implementation of the parameter or rather operator `y` or `z`, respectively.
- During the evaluation of the implementation `x ? y ! z` of the `if` operator, depending on the value of the parameter `x`, either only the application of the operator `y` or only the application of the operator `z` is evaluated according to § 2.6.3, i. e., either the operand `print 1` or the operand `print 0`.

3.8.2 Further Examples

Head-controlled loop

- Definition:

```
while [(X:type)] (\ x -> (X))
do [(Y:type)] (\ y -> (Y)) end -> (int =
  (?* if x then y; true else false end) - 1
)
```

- Because the operands belonging to the parameters `x` and `y` are passed unevaluated, they can be evaluated any number of times (even not at all) in the implementation of the `while` operator.
- The result value of the operator is the number of *complete* iterations, i. e., the number of evaluations of `y`, which is 1 less than the number of evaluations of the expression `if ... end` that is returned by the predefined loop operator `?*•`.
- Exemplary use (the operator •<=• is defined on a task sheet):

```
i : int?; i =! 1;
while ?i <= 10 do
  print ?i; i =! ?i + 1
end
```

Counting Loop

- Definition:

```
for (var:int?) "=" (lower:int) ".." (upper:int)
do [(T:type)] (\ body -> (T)) end -> (int =
  var =! lower;
  while ?var <= upper do
    body;
    var =! ?var + 1
  end
)
```

- Exemplary use:

```
i : int?;
for i = 1 .. 10 do
  print ?i
end
```

3.8.3 Lambda Parameters with Parameters

- Because a lambda parameter can be an operator with an arbitrary signature, it might have parameters itself.

- And because the operands belonging to a lambda parameter are used as the implementation of this operator, the parameters of the lambda parameter are visible in these operands, just as the parameters of an operator are always visible in its implementation.

- If a lambda parameter is applied in the implementation of the operator, its parameters are initialized, just as in every operator application, with the values of the corresponding operands.

Example

- Definition:

```
for (lower:int) ".." (upper:int)
do [(T:type)] (\ body (current:int) -> (T)) end -> (int =
  var : int?;
  var =! lower;
  while ?var <= upper do
    body ?var;
    var =! ?var + 1
  end
)
```

- Exemplary use:

```
for 1 .. 10 do
  print current
end
```

- Because the lambda parameter `body` has itself a parameter named `current`, this parameter is visible in the corresponding operand `print current`.
- Because the current value of the variable `var` is passed as an operand to every application of `body`, `current` will have the respective value in every iteration.

3.8.4 Passing Names to Operators

- If a parameter has the predefined type `sym` or `syms`, no normal expressions can be passed as operands, but either a single name (for the type `sym`) or a sequence of one or more names (for the type `syms`) according to § 2.4.

- If such a parameter is in turn used instead of a name in a parameter or operator declaration, it will be replaced there with the passed name or names, respectively, in every application of the operator to which it belongs.

- To make such uses unambiguous, the names of such parameters must not be “simple” names according to § 2.4 (i. e., sequences of letters and digits starting with a letter), but must contain at least one other character (or start with a digit).

Example

- Definition:

```
for ("#name":syms) "=" (lower:int) ".." (upper:int)
do [(T:type)] (\ body (#name:int) -> (T)) end -> (int =
  var : int?;
  var =! lower;
  while ?var <= upper do
    body ?var;
    var =! ?var + 1
  end
)
```

- Exemplary use:

```
for i = 1 .. 10 do
  print i
end
```

- In the application of the `for` operator, the name `i` is passed to the parameter `#name`, which is used in turn as the name of the parameter of `body`.
  Therefore, this parameter has the name `i` in this application of `for` and, therefore, can be used with this name in the operand `print i` belonging to `body`.

- In another application of `for`, some other name could be used instead of `i`, for example:

```
for j = 1 .. 10 do
  print j
end
```

- Because the parameter `#name` has type `syms`, it is also possible to use multi-part names, for example:

```
for current value = 1 .. 10 do
  print current value
end
```

- If the parameter `#name` would have the simple name `name`, its use in the parameter declaration `(name:int)` would be ambiguous, because the character sequence `name` could then either be directly interpreted as a name or as an application of the parameter with this name.