Hochschule Aalen

Fakultät Elektronik und Informatik
Studienbereich Informatik

Advanced Programming with MOSTflexiPL

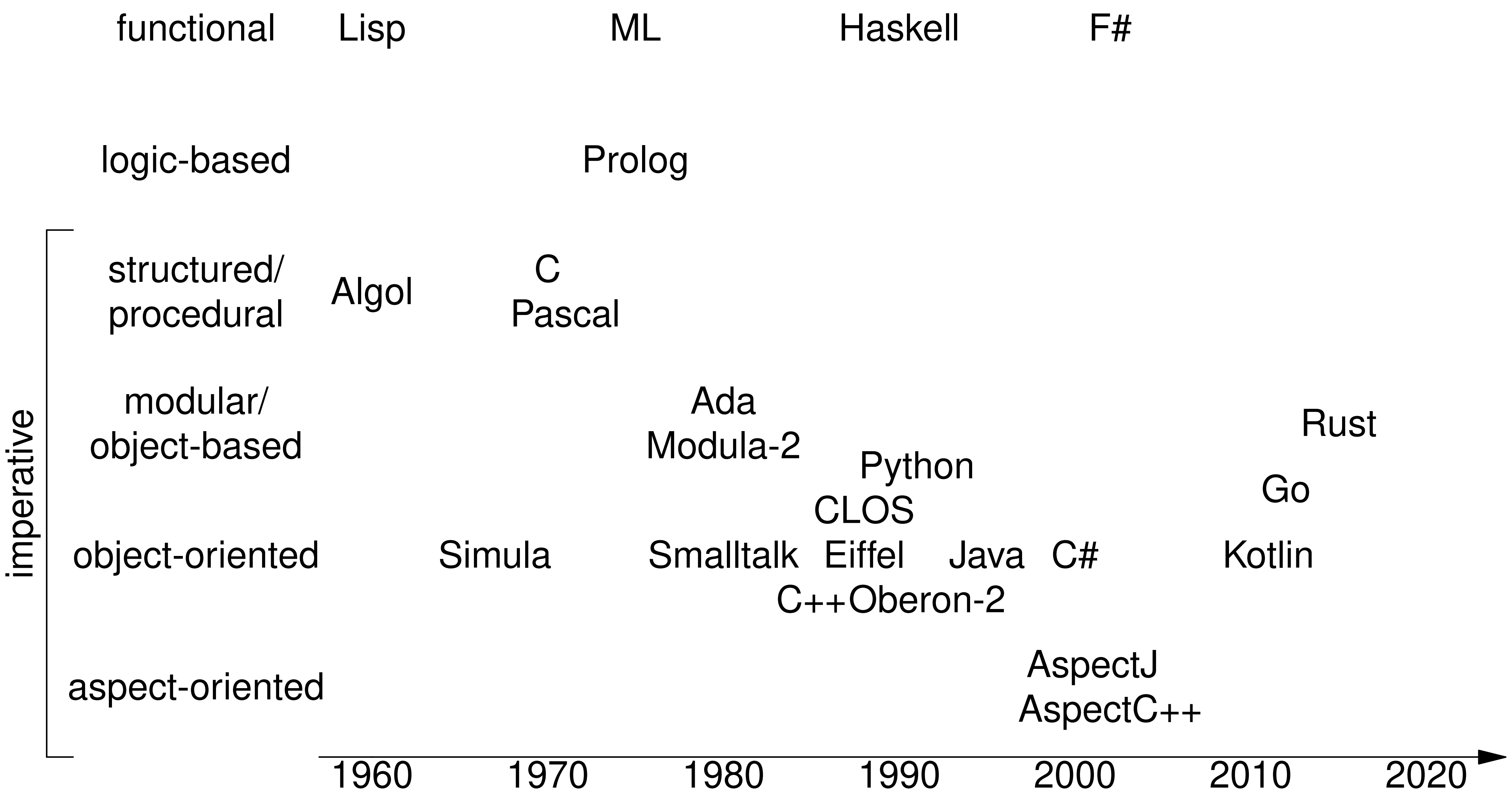
Lecture in Wintersemester 2025/2026

Prof. Dr. habil. Christian Heinlein

christian.heinleins.net

1 Introduction

1.1 Well-known Programming Languages and Paradigms



1.2 Imperative and Functional Programming

1.2.1 Difference

❐ *Imperative* programs are *state-based*, i. e., they can *modify* the state of their objects by assignments to (global, local, or instance) variables, which are in turn used in conditions of branches and loops and therefore affect a program's behaviour.

❐ If an object is used at different locations of a program, a modification at one location can easily have unexpected and undesired side effects at other locations.

❐ In contrast, *functional* programs are *state-less*, i. e., there are only constants, but no variables and assignments and therefore no mutable state. By that means, unexpected side effects cannot happen at all.

❐ Branches (whose conditions are based on constant values) are still possible, but recursion has to be employed instead of loops.

❐ A data structure that is frequently used in functional programs are lists consisting of a first element (head) and a (possibly empty) rest list (tail).

❐ Functions can conveniently process such lists by processing the first element directly and the rest list by calling the function recursively.

❐ The correctness of such functions can usually be easily proven by induction on the length of the list (mostly with base case 0).

1.2.2 Examples

❐ Construction of a list with the natural numbers from `m` up to `n`

❐ Replacing some value `u` with some value `v` in a list of integral numbers

Implementation in the Imperative Programming Language Java

```
List<Integer> nat (int m, int n) {
  List<Integer> xs = new LinkedList<Integer>();
  for (int i = m; i <= n; i++) xs.add(i);
  return xs;
}

List<Integer> subst (List<Integer> xs, int u, int v) {
  List<Integer> ys = new LinkedList<Integer>();
  for (int x : xs) {
    if (x == u) x = v;
    ys.add(x);
  }
  return ys;
}
```

❐ The lists `xs` in the `nat` method and `ys` in the `subst` method are repeatedly modified during the loop by calling the `add` method.

Implementation in the Functional Programming Language Haskell

```
nat m n = if m > n then [] else m : (nat (m+1) n)

subst [] u v = []
subst (x:xs) u v = (if x == u then v else x) : (subst xs u v)
```

- ❐ Here, every call of the functions `nat` and `subst`, whether directly from outside or via recursion, returns an immutable list of integral numbers.

- ❐ A definition of the form `f x ... = impl` defines a function with name `f`, parameters `x ...`, and implementation `impl`, which can afterwards be called with expressions of the form `f a ...`

- ❐ An expression of the form `if x then y else z` yields, according to the value of the condition `x`, either the value of `y` or the value of `z`.

- ❐ `[]` denotes the empty list, an expression of the form `x:xs` constructs and returns a new list with first element `x` and rest list `xs`.
  (Technically, `x:xs` creates only a new list node containing the element `x` and the pointer `xs` and returns the pointer to this node, which logically represents the whole list.)

- The two definitions of the `subst` function constitute an implicit branch by so-called *pattern matching*:
  If the conveyed list is of the form `[]` (i. e., empty), the first definition is called; if it is of the form `x:xs` (i. e., it contains at least one element `x`), the second definition is called, whose parameters `x` and `xs` will contain the first element and the (possibly empty) rest list of the conveyed list, respectively.

- As mentioned in § 1.2.1, the correctness of the functions `nat` and `subst` can easily be proven by induction (on the length of the conveyed list in the case of `subst` and on the length $k = \max(n - m + 1, 0)$ of the returned list in the case of `nat`).

- Even though Haskell is statically typed, the parameter and result types of functions can be omitted, if they can be deduced by the compiler. (By that means, the `nat` function can actually be called for any numeric type, while the `subst` function can be called with lists of any type.)

1.2.3 Remarks

❒ Imperative and functional programming are basically just *programming paradigms*, i. e., they define the principal way of doing programming (and the way of thinking while programming).

❒ Imperative and functional programming languages provide the mechanisms which are required to do programming in the respective paradigm.

❒ Nevertheless, it is usually possible to write functional programs in an imperative programming language (if it provides recursion) by completely abstaining from assignments.

❒ However, it is impossible to write imperative programs in a purely functional language due to the absence of variables.

1.3 Meaning of the Name MOSTflexiPL

The acronym MOSTflexiPL (pronounced like *most flexible*, but with *p* instead of *b*) stands for:

- **MO**dular
  - A comprehensive program can be divided into manageable modules.
  - A module can be used as a library in different programs.
- **S**tatically **T**yped
  - A program is completely checked for being type-correct at compile time.
  - Type errors cannot occur at run time.
- **fl**exibly **ex**tensi**b**le
  - The syntax of the language can be easily extended and customized in a virtually unlimited way by every programmer.
- **P**rogramming **L**anguage
  - It is a general-purpose programming language.

1.4 Descriptive Analogies

1.4.1 Rearrangement of a Room

- In a room one can
  - freely move around furniture
  - mount lamps at the ceiling and attach shelves to the walls
  - freely paint or repaper walls
  - and so on
- But one cannot simply
  - displace doors and windows
  - move around or remove walls
  - move ceilings up or down
  - and so on

Analogy to Programming Languages

- With a conventional programming language one can
  - define new data types and functions
  - possibly overload predefined operators (e. g., in C++)
  - possibly define new prefix, infix, and postfix operators (e. g., in Haskell)
- But one cannot simply
  - extend the syntax of types and declarations
  - define new operators with arbitrary syntax
  - define new control flow statements
- With MOSTflexiPL, one can do all this and even much more, just as if one could rearrange a room in a completely unlimited way.

1.4.2 Natural Language and Mathematical Notation

- In a natural language one may
  - define arbitrary new terms and use them afterwards
  - define arbitrary new abbreviations and use them afterwards

- But one may not simply
  - define new punctuation marks and use them afterwards
  - change the grammar of the language
  - change the rules for capitalization of words
  - write text in the opposite direction

- In a mathematical article one may
  - define arbitrary new symbols with arbitrary notation and use them afterwards, for example $\sum\limits_{k=1}^{n} a_k$ or $\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$

- Therefore, conventional programming languages are limited similarly to natural languages, while MOSTflexiPL offers possibilities comparable to mathematical notation.

1.4.3 Model Railways

- A model railway system provides
  - straight railtracks of different lengths
  - bent railtracks with different radiuses and angles
  - different kinds of switches and crossings
- But with these things one cannot get
  - straight railtracks of any length
  - bent railtracks with any radius or angle
  - completely different railtrack figures (e. g., ellipses)
  - arbitrary switches and crossings
    (e. g., switches with more than three ways or crossings with arbitrary angles)

Analogy to Programming Languages

❐ Conventional programming languages are limited similarly to normal model railway systems.

❐ MOSTflexiPL corresponds to a fictitious model railway system which allows to

- ❍ stretch or squeeze straight railtracks to any desired length
- ❍ shape bent railtracks in an arbitrary way
  (this is in fact provided by some manufacturers)
- ❍ build new forms of switches and crossings simply by stacking multiple bent and straight railtracks

❐ This flexibility is also expressed by the logo of the language:

MOST flexiPL

1.5 Basic Principles

- Everything is an *expression*, i. e., the application of an *operator* to subexpressions (*operands*).
- An operator may have any number of names and operands (depicted by •) in any order, for example (predefined and user-defined operators):
  - Infix operators: addition `•+•`, assignment `•=!•`
  - Prefix operators: change sign `-•`, logical negation `not •`
  - Postfix operators: faculty `•!`, square $\bullet^2$
  - Circumfix operators: parentheses `(•)`, absolute value $|\bullet|$
  - Control flow statements: branch `if•then•else•end`, loop `while•do•end`
  - Declarations: constant declaration `•:•=•`, operator declaration `•->(•=•)`
  - Type expressions: variable type `•?`, array type `•[]`
- At run time, every expression yields a value.
- Based on a small set of predefined operators (for arithmetic, logic, control flow), it is possible to define arbitrary user-defined operators.

1.6 Compiler

1.6.1 Availability

❐ The MOSTflexiPL compiler `flxc` is installed in the directory `/usr/local/bin` of a virtual bwCloud server running Ubuntu 24.04 LTS. The server's IP address will be announced during the lectures.

❐ After a user has been activated by depositing his public SSH key (which can be generated, if necessary, with `ssh-keygen`), he can use SSH with user name `studNNN` (where `NNN` is the user's matriculation number) to log in into this server without a password.

❐ During the semester, the compiler will be updated regularly.

1.6.2 Invocation

❒ `flxc filename.flx` checks the MOSTflexiPL program in the given source file for syntactic and semantic correctness und creates the corresponding syntax tree.

❒ Afterwards, if the program is correct and unambiguous, the syntax tree is directly executed and also saved in binary form in the file `filename.flx.bin`.

❒ If the source file (and the compiler) has not been modified since the last successful invocation of the compiler (i. e., if the binary file is newer than the source file and has been produced by the same version of the compiler), the time-consuming correctness check of the program is omitted; in that case, the syntax tree is directly read from the binary file and executed immediately, which is significantly faster for large programs.

1.6.3 Error Messages

- If the program is not correct, one or more diagnostic messages for possible error reasons are displayed.
- For that purpose, three main error categories are distinguished:
  - *Insertion errors*:
    A subexpression cannot be inserted as an operand into a still incomplete expression, either because its type does not match or because it would violate an exclude specification (cf. § 2.9).

    For example: `1 + true`
    The subexpression `true` with type `bool` cannot be inserted as an operand into the still incomplete addition `1 +`.
  - *Continuation errors*:
    A still incomplete expression cannot be continued, because it is not followed by a matching word in the input.

    For example: `if x > 2 else y end`
    The still incomplete expression `if x > 2` cannot be continued, because the word `then` which would be necessary for that does not follow in the input.

- *Completion errors*:
  An error is only detected during the final check of an already complete expression. The reasons for such errors are very specific, e. g., that the type of a subexpression cannot be uniquely determined or that an implicit parameter cannot be assigned.

- The (complete oder incomplete) expressions involved in a possible error are always displayed in the form `12:34(...)56:78`, where `12:34` and `56:78` represent the expression's begin and end position, respectively, each of which consists of a line number and a character position within that line, both counted from 1. (The expression starts at the begin position and ends immediately *before* the end position.)

- The part between the parentheses denotes the expression's operator, e. g., `add/sub: •(+|-)•` for the predefined operator for addition and subtraction or `true` for the predefined constant `true`.

- In case of a user-defined operator, the part between the parentheses is itself of the form `21:43[9]65:87`, where `21:43` and `65:87` represent the begin and end position, respectively, of the operator's definition, while `9` is the number of the source file containing this definition.

- The names of the source files corresponding to these numbers are displayed at the end of the whole error output.

- The readability of the error messages is improved by employing different colors.

- If `flxc` is called with the option `-v`, even more detailed error messages are produced:
  On the one hand, the signature (cf. § 2.8) of user-defined operators is printed in addition; on the other hand, the line(s) of the source file corresponding to each possible error reason are printed in addition.

1.6.4 Display of Ambiguities

❐ If the program is ambiguous, one ore more diagnostic messages of the following form are displayed:

```
ambiguity from position 12:34 to 56:78
```

Here, `12:34` and `56:78` again represent the begin and end position of the ambiguous source code block.

❐ In the subsequent lines, this source code block is displayed (without the white space it contains) twice with different colors for each possible interpretation:

❐ In the first part of each line (before the vertical bar), each color represents a particular (predefined or user-defined) operator.
The operators corresponding to these colors are shown afterwards also with their respective colors.

❐ In the second part of each line (after the vertical bar), each color represents a particular precedence level.
The colors of these precedence levels are red, yellow, green, blue etc. from top (low precedence) to bottom (high precedence) and are shown at the end of the line `ambiguity from ...`

Example

```
ambiguity from position 3:6 to 3:12: operators | precedence levels (🟥🟨🟩)
2+3² | 2+3²
2+3² | 2+3²
+ operator defined from position 1:1 to 1:20 in module ambig5.flx
+ predefined operator add/sub: •(+|-)•
+ predefined operator int literal
```

- Because the precedence of the operator $\bullet^2$ (shown in red before the vertical bar) defined in the source file `ambig5.flx` has not been specified, the expression $2 + 3^2$ can be interpreted in two ways which are shown after the vertical bar:
  - Either the square operator (red) is applied to the addition (yellow) of `2` and `3` (both green).
  - Or the addition (red) is applied to `2` (yellow) and the application of the square operator (also yellow) to `3` (green).

1.6.5 Integration into Visual Studio Code

❐ An extension for Visual Studio Code, which integrates the compiler into this editor, is provided in the directory `/usr/local/share/flxc` of the virtual server.

❐ If this extension is installed in VS Code for Linux, a MOSTflexiPL source file is always checked by the compiler when it is opened or saved (i. e., the code is not checked incrementally). Possible errors or ambiguities are then displayed directly in VS Code.

❐ Furthermore, the following features are provided:

- ❍ Simple syntax coloring, e. g., for comments and parentheses.
- ❍ If a source file does not contain any errors, but at most ambiguities, „Go to Definition“ jumps to the definition of a name and „Go to References“ jumps to the locations where a name is used.

❐ Whenever the compiler is updated, this VS Code extension is updated correspondingly.

1.6.6 Integration into Vi IMproved

❐ A corresponding integration into Vi IMproved is in progress.

1.7 Practical Advice

❒ Forget almost everything you are accustomed to from other programming languages!

❒ Let your imagination run wild!
Design your “programming room” to your heart’s content!

❒ Semicolon is an infix operator. Therefore, it may only appear *between* subexpressions, but not at the end of a „chain“ of expressions.

❒ White space between names and operands of an expression is *always* optional. Therefore, `printonlyx` instead of `print only x` is correct, for example, if there is a suitable constant `x`. (However, `pr int on ly x` would not be correct: White space is not permitted inside of a single name.)

❒ If an error or an ambiguity is not obvious, reduce your program step by step until the problem disappears.

❒ On the other hand, develop your programs in many small steps and check every single step by calling the compiler.

❒ If an error remains unclear even after extensive investigation, ask me.
(Unfortunately, the compiler is also not completely free of errors.)

1.8 Bibliography

❒ C. Heinlein: “MOSTflexiPL – An Extremely Flexible Programming Language.” In: T. Noll, I. Fesefeldt (eds.): *22. Kolloquium Programmiersprachen und Grundlagen der Programmierung*. Aachener Informatik-Berichte AIB-2023-02, Department of Computer Science, RWTH Aachen, 2023, 55–72.

The most recent publication.

❒ C. Heinlein: “MOSTflexiPL – Modular, Statically Typed, Flexibly Extensible Programming Language.” In: J. Edwards (ed.): *Proc. ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2012)* (Tucson, AZ, October 2012), 159–178.

The most extensive publication yet.

❒ C. Heinlein: “MOSTflexiPL – Modular, Statically Typed, Flexibly Extensible Programming Language.” In: S. Jähnichen, B. Rumpe, H. Schlingloff (eds.): *Software Engineering 2012 Workshopband* (Fachtagung des GI-Fachbereichs Softwaretechnik; Februar/März 2012; Berlin). Lecture Notes in Informatics P-199, Gesellschaft für Informatik e. V., Bonn, 2012, 45–60.

A shorter description of the language.

❐ C. Heinlein: “Fortgeschrittene Syntaxerweiterungen durch virtuelle Operatoren in MOSTflexiPL.” In: K. Schmid et al. (ed.): *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2014* (Kiel, February 2014). CEUR Workshop Proceedings, 193–212, http://ceur-ws.org/Vol-1129/paper4A.pdf.

A description of so-called virtual operators.

All publications except the first use a syntax of the language which is outdated by now.

All of them are available on `flexipl.info/publications`.

2 Predefined Types and Operators

2.1 Comments

- ❒ `$$` starts a *line comment* that lasts until the end of the current line.
- ❒ `$(` starts a *block comment* that lasts until the next occurrence of `)$` on the same level of nesting, i. e., block comments might be arbitrarily nested.
- ❒ However, line and block comments are independent of each other.

2.2 Precedence and Associativity of Operators

❒ If a prefix, infix, or postfix operator *binds more strongly* (i. e., possesses a *higher precedence*) than another, applications of the more strongly binding operator can be used directly (i. e, without parentheses) as operands of the more weakly binding operator, but not vice versa.

  For example, according to usual mathematical practice, multiplications and divisions can be directly used as operands of additions and subtractions, while additions and subtractions must be parenthesized when used as operands of multiplications and divisions.

❒ If an infix operator or a group of such operators with the same precedence is *left-* or *right-associative*, respectively, direct applications of these operators themselves can only be used in their left or right operand, respectively, but not in their other operand.

  Because the arithmetic operators are left-associative, `x - y - z` implicitly means `(x - y) - z` and not `x - (y - z)`, for example.

❒ If an infix operator or a group of such operators with the same precedence is *non-associative*, direct applications of these operators themselves cannot be used in either of their operands.

  Because the equality test (cf. § 2.5) is non-associative, an expression such as `x = y = z` is erroneous.

2.3 Integral Numbers and Arithmetic Operations

❒ Integral numbers possess type `int` and can become arbitrarily large in principle.

❒ An arbitrarily long sequence of digits such as `123456789123456789` is interpreted as a decimal number (even if its initial digit is `0`).

❒ `print•` writes an integral number in decimal notation, followed by a line separator, to the standard output stream (see also § 2.12).

❒ Addition (`•+•`), subtraction (`•-•`), multiplication (`•*•`), division (`•:•`), remainder (`•-:-•`) and change of sign (`-•`) have the meaning known from other programming languages, and they obey the usual rules for precedence and associativity. `print•` binds more weakly than these operators.

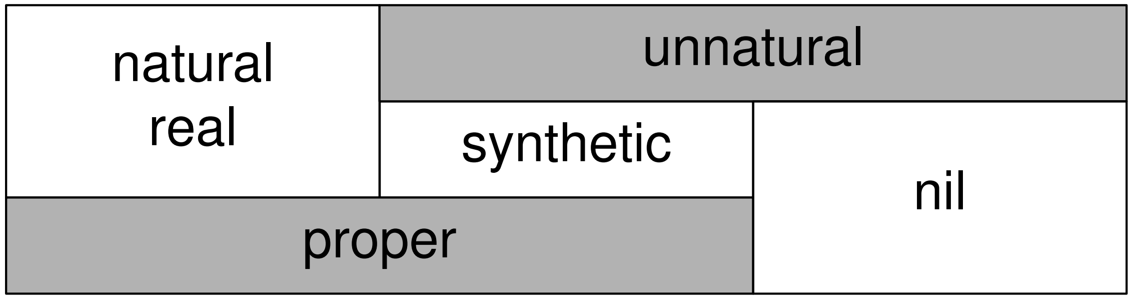
❒ Division truncates possible decimal places (rounding towards 0).

❒ Division by 0 yields the special value nil (abbreviation of the latin “nihil” meaning “nothing”), which is available for any type.

❒ Furthermore, every type can have arbitrarily many *synthetic* values, which arise, amongst others, from constant declarations without an initialization (cf. § 2.4).

- Therefore, there are basically three disjoint categories of values:
  - natural or real values (numbers)
  - synthetic values
  - the nil value

- Synthetic values and nil are collectively also called *unnatural values*,
  natural and synthetic values are collectively also called *proper values*:



- If at least one operand of an arithmetic operation is an unnatural value, the result is nil.

- The remainder `x -:- y` is always equal to `x - x : y * y`.
  (This implies, amongst others, that `x -:- 0` is equal to nil.
  For `x >= 0` and `y > 0`, `x -:- y` is equal to `x` modulo `y`, but not for other values of `x` and `y`; for example, `-5 -:- 3` is equal to `-2`, while `-5` modulo `3` is equal to `1`.)

2.4 Constant Declarations

- A declaration of the form `name : type = init` defines a constant with name `name` and type `type`, whose value results from evaluating the initialization expression `init`, e. g., `N : int = 10 * 10`.

- Actually, `name` might also consist of multiple names, e. g., `line length : int = 10 * 10`.

- Each single name is
  - either a non-empty sequence of letters and digits of the ASCII code starting with a letter (which describes exactly this sequence of characters)
  - or a non-empty sequence of arbitrary Unicode characters except white space inside of double quotation marks (which describes the sequence of characters between the quotation marks).
    To include a double quotation mark into such a sequence of characters, it must be duplicated; single quotation marks can be used directly.

- For example:

```
N : int = 10 * 10;
"N′" : int = N + 1;
"N""" : int = N′ + 1;
print N"
```

- If the type of the constant is omitted, it is automatically deduced from the type of the initialization, e. g., `N := 10 * 10`.

- If the initialization is omitted (e. g., `S : int`), the constant receives a new synthetic value which is different from all other synthetic values.
  This is primarily important for variable types (cf. § 2.7) and user-defined types (cf. § 4.3), but can in principle also be used for types such as `int`.

- The result value of a constant declaration is the value of the constant.

- A constant is a nullary operator, i. e., an operator possessing only names, but no operands.

- The constant declaration operator binds more weakly than the output operator `print•`.

2.5 Truth Values and Comparisons

❒ The truth values `true` and `false` possess the type `bool`.
Actually, `true` is simply a synthetic value and `false` is the nil value of type `bool`.

❒ For two values `x` and `y` of the same arbitrary type, the equality test `x = y` tests whether the values are equal, that means:

- ❍ Both represent the same natural value
- ❍ or both are nil
- ❍ or both represent the same synthetic value.

The result is the corresponding truth value `true` or `false`.

❒ For a value `x` of type `int`, the negative test `x -` tests whether the value is negative, i. e., whether it is a natural value which is less than 0. The result is again the corresponding truth value `true` or `false`.

❒ The equality operator `•=•` is non-associative and binds more strongly than the output operator `print•` and more weakly than the negative test `•-`, which in turn binds more weakly than the arithmetic operators.

❒ With these basic operators, it is easy to implement all other known comparison operators by oneself.

2.6 Control Flow

2.6.1 Sequential Evaluation

- ❒ Semicolon is a left-associative infix operator with lowest precedence used to concatenate subexpressions which shall be evaluated sequentially, that is:

- ❒ For expressions `x` and `y` with arbitrary types `X` and `Y`, the expression `x; y` evaluates the subexpressions `x` and `y` sequentially und returns the value of `y` (i. e., the result type is `Y`).

2.6.2 Parentheses

- ❒ Parentheses can be employed as usual to explicitly control precedence, e. g., `(x + y) * z`.
  This is necessary in particular, if a semicolon expression shall be used as an operand of an operator with higher precedence, e. g., `(print 1; x) + (print 2; y)`.
  (With the output of `print` one can observe the evaluation order of the subexpressions of the addition.)

- ❒ `(•)` is an ordinary operator which evaluates its operand (with any type `X`) and returns its value (i. e., the result type is `X`).

2.6.3 Branch

- For an operand `x` of any type and two operands `y` and `z` of the same arbitrary type `T`, the elementary branch `x ? y ! z` returns either the value of `y` or the value of `z` (i. e., the result type is `T`), depending on whether the value of `x` is a proper value (i. e., different from nil) or nil.

- Therefore, after the evaluation of `x`, only one of `y` and `z` is evaluated.

2.6.4 Loop

- For an operand `x` of any type, the elementary loop `?* x` evaluates the operand `x` repeatedly until its evaluation returns nil.

- The result value of type `int` is the number of evaluations of `x`.

2.6.5 Precedence and Associativity

❒ The loop operator binds weaker then the output operator `print•` and stronger than the branch operator, which in turn binds stronger than the equality test.

❒ The branch operator is right-associative to allow „if-elseif chains“ without using parentheses, i. e., it can be used directly in its right (third), but not in its left (first) operand.

In the middle (second) operand, operators with arbitrary precedence (including sequential evaluation) can be used directly, just as in the operand of parentheses.

2.6.6 Other Operators

❒ With these basic operators and the possibility to pass operands unevaluated to operators (cf. § 3.8), it is easy to define arbitrary other control flow operators, e. g., `if•then•else•end` or `while•do•end`.

2.7 Variables

❒ For an arbitrary type `T`, the type `T?` denotes *variables* with *content type* `T`.

❒ Therefore, a synthetic value `x` of such a type `T?`, which can be created according to § 2.4 with a constant (!) declaration `x : T?`, denotes a unique memory cell that contains a varying value of type `T` (initially nil), i. e., a variable with content type `T`.

❒ For a variable `x` of type `T?`, `?x` returns the current value (with type `T`) of the variable, and `x =! y` replaces that value with the value `y` (also of type `T`).

❒ If `x` is the nil value (of type `T?`), `?x` also returns nil (of type `T`), and `x =! y` has no effect. Therefore, a nil variable is comparable to the special file `/dev/null` in Unix systems, which returns EOF (i. e., nothing) for read operations and ignores write operations.

❒ Because `=` denotes the equality operator, the symbol `=!` has been chosen for the assignment. Furthermore, the exclamation mark emphasizes the imperative nature of the assignment. (Without the assignment operator, MOSTflexiPL would not be an imperative, but a purely functional programming language.)

- ❐ The assignment operator `•=!•` is right-associative (to allow multiple assignments of the form `x1 =! x2 =! y` without using parentheses) and has the same precedence as the branch `•?•!•` (so that e. g. `x =! 1 + 2` is interpreted as `x =! (1 + 2)` and not as `(x =! 1) + 2`).

- ❐ The query operator `?•` has the same precedence as the change sign operator `-•`.

2.8 Simple Operator Declarations

- An operator declaration is of the form `sig -> (type = impl)`.
- The type `type` is the *result type* of the declared operator, the expression `impl`, which must have type `type`, constitutes the *implementation* of the operator.
- The *signature* `sig` is a non-empty sequence of names (cf. § 2.4) and *parameter declarations* of the form `(names : type)`, where `names` is (just as for a constant declaration) itself a non-empty sequence of names of the parameter and `type` is its type.
  The parameters declared that way are visible only in the implementation `impl`.
  The same holds for constants and operators declared inside of the implementation (local declarations).
  The declared operator itself is already visible in its own implementation to allow recursive applications.
- If every name of the signature is replaced with the character sequence which it describes according to § 2.4, and every parameter is replaced with a corresponding operand, i. e., an expression having the type of this parameter, an *application* of the operator is formed, i. e., an expression whose type is the result type of the operator.
  Between the parts of such an expression (the names of the operator and the operands), zero or more white space characters and comments are permitted.

❒ An operator application of this kind is evaluated by first evaluating the operands from left to right and initializing every parameter like a constant with the value of the corresponding operand.
Afterwards, the implementation of the operator is evaluated to determine the result of the operator application.

Examples

```
(x:int) "²" -> (int = x * x);                 print 5²;

(n:int) "!" -> (int = (n - 1)- ? 1 ! n * ((n-1)!));
                                              print 100!
```

Functions with Varying Syntax

```
avg "(" (x:int) "," (y:int) ")" -> (int = (x + y) : 2);
                                              print avg(10, 20);

avg (x:int) (y:int) -> (int = (x + y) : 2);
                                              print avg 10 20;

avg of (x:int) and (y:int) -> (int = (x + y) : 2);
                                              print avg of 10 and 20
```

2.9 Exclude Declarations

2.9.1 Direct Exclusions

❐ Because there are no predefined rules for precedence and associativity of user-defined operators (with one exception, see below), an expression such as `2 + 3`^{2} is ambiguous: It can be interpreted either as `2 + (3`^{2}`)` or as `(2 + 3)`^{2}.

❐ The second interpretation can be excluded with an exclude declaration `excl (2 + 3)`^{2} `end`, where the subexpressions `2` and `3` are discretionary placeholders for arbitrary operands of type `int`.

General Rules

❒ For every parameter of an operator, there is a set of excluded operators, which must not appear at the top and possibly also not at the left or right border (see below) of corresponding operands.

❒ If the expression between `excl` and `end` contains a parenthesized subexpression as an operand for a particular parameter, the operator at the top of this subexpression is added to the exclude set of this parameter.
If the expression between `excl` and `end` contains multiple parenthesized subexpressions, this holds for each of them.

❒ If there is no name before or after a parameter in the signature of its operator, its exclude set implicitly contains the predefined sequential evaluation `•;•`, causing this operator to automatically bind weaker than all other operators. Otherwise, the exclude set of a parameter is initially empty.

Examples

- Square binds stronger than the basic arithmetic operations including change sign, i. e., applications of these operators are forbidden as operands of the square operator:

$$\texttt{excl (1+2)}^{2}\texttt{; (1-2)}^{2}\texttt{; (1*2)}^{2}\texttt{; (1:2)}^{2}\texttt{; (1-:-2)}^{2}\texttt{; (-1)}^{2}\texttt{ end}$$

- Multiplication, division, and remainder bind stronger than addition and subtraction, i. e., additions and subtractions are forbidden as operands of multiplication, division, and remainder (these exclusions are already predefined):

```
excl (1+2)  *  (3+4); (1-2)  *  (3-4) end;
excl (1+2)  :  (3+4); (1-2)  :  (3-4) end;
excl (1+2) -:- (3+4); (1-2) -:- (3-4) end
```

- Addition and subtraction as well as multiplication, division, and remainder are each collectively left-associative, i. e., they are forbidden in their own right operands (these exclusions are already predefined, too):

```
excl 1  +  (2+3); 1  +  (2-3) end;
excl 1  -  (2+3); 1  -  (2-3) end;
excl 1  *  (2*3); 1  *  (2:3); 1  *  (2-:-3) end;
excl 1  :  (2*3); 1  :  (2:3); 1  :  (2-:-3) end;
excl 1 -:- (2*3); 1 -:- (2:3); 1 -:- (2-:-3) end
```

Left and Right Border of an Expression

❐ A direct or indirect *subexpression* of an expression belongs to the *left* or *right border* of this expression, if it begins or ends at the same position as the entire expression, respectively.

For example, the subexpression `1 * 2` belongs to the left border and the subexpression `3 : 4` to the right border of the expression `1 * 2 + 3 : 4`.

The right border of the expression `print 1 * 2 + 3 : 4` contains the subexpression `1 * 2 + 3 : 4` as well as the subexpression `3 : 4` of it, but not the subexpression `1 * 2`. The left border of the expression does not contain any subexpression.

❐ An *operator* belongs to the left or right border of an expression, if some subexpression belonging to the left or right border of the expression, respectively, is an application of this operator.

According to that, the operator `•*•` belongs to the left border and the operator `•:•` belongs to the right border of the expression `1 * 2 + 3 : 4`.

The right border of the expression `print 1 * 2 + 3 : 4` contains the operator `•+•` as well as the operator `•:•`, but not the operator `•*•`.

- To make the prefix operator `?*•` bind weaker than the arithmetic operators, as described in the previous sections, it is excluded in the left operand of each respective infix operator.
  (Excluding it in their right operand or in the operand of prefix operators is not necessary, because expressions such as `1 + ?* 2` or `- ?* 3` are unambiguous even without these exclusions.)

- But if exclusions would apply only to the top of the respective operands, an expression such as `1 + ?* 2 + 3` would still be ambiguous, because it could be interpreted either as `1 + ?* (2 + 3)` or as `(1 + ?* 2) + 3`. (The operator `?*•` is not at the top of the subexpression `1 + ?* 2`, but belongs to its right border.)

- To make sure that the second interpretation is excluded by the exclusion of `?*•` in the left operand of the addition, too, an excluded operator is possibly also forbidden at the left or right border of the respective operands:

❍ If the respective operand belongs to the left border of its expression, the relevant operator is also forbidden at the right border of the operand.

  This implies, for example, that `1 + ?* 2 + 3` can now only be interpreted as `1 + ?* (2 + 3)` and no longer as `(1 + ?* 2) + 3`.

❍ If the respective operand belongs to the right border of its expression, the relevant operator is also forbidden at the left border of the operand.

  If there would be a right-associative power operator `•^•` and a weaker binding postfix operator `•@` with parameter and result type `int`, which is therefore excluded in the right operand of the power operator, the above rule would imply that `1 ^ 2 @ ^ 3` can only be interpreted as `(1 ^ 2) @ ^ 3` and not as `1 ^ (2 @ ^ 3)`.

❍ If the respective operand does neither belong to the left nor to the right border, but to the „interior“ of its expression, the relevant operator is forbidden only at the top of the operand. (However, exclusions for such inner operands are generally not necessary to define precedence and associativity of operators.)

2.9.2 Indirect Exclusions

- Because the definition of extensive precedence hierarchies with direct exclusions is rather cumbersome, indirect exclusions can be used to define them more easily, flexibly, and modularly.

- An indirect exclusion defines either that the exclude set of some parameter contains all operators of the exclude set of some other parameter,
  or that an operator is excluded anywhere where another operator is excluded.

- In general:
  - If the expression between `excl` and `end` contains a subexpression of the form `(left) -> (right)`, the operator at the top of the subexpression `right` is excluded anywhere where the operator at the top of the subexpression `left` is excluded.

- If the expression between `excl` and `end` contains a subexpression of the form `left -> right` with unparenthesized subexpressions `left` and `right`, the exclude set of the parameter identified by the subexpression `right` contains all operators of the exclude set of the parameter identified by the subexpression `left`.

  To make a subexpression identify a particular parameter, the same subexpression must appear earlier in the expression between `excl` and `end` as an operand for that parameter. If it appears multiple times, its first appearance is relevant.

- If a double arrow `<->` is used instead of the single arrow `->`, the above rules also apply when the roles of `left` and `right` are reversed.

- The relationships between operators or parameters which are defined in that way, are also effective for all exclusions which are later defined directly or indirectly for these operators or parameters, respectively.

- The rules given at the end of § 2.9.1 regarding the exclusion of operators at the left or right border of operands also hold for exclusions which are defined indirectly in that way.

Examples

❒ Addition is left-associative:

```
excl 1 + (2 + 3) end
```

❒ Multiplication is left-associative and binds stronger than addition:

```
excl 1 * (2 * 3) end;
excl 1 + 2; 3 * 4; 2 -> 3; 2 -> 4 end
```

❒ Change sign binds stronger than multiplication:

```
excl 1 * 2; -3; 2 -> 3 end
```

❒ Subtraction has the same binding properties as addition:

```
excl (1 + 2) <-> (3 - 4); 1 <-> 3; 2 <-> 4 end
```

❒ Division and remainder have the same binding properties as multiplication:

```
excl (1 * 2) <-> (3  :  4); 1 <-> 3; 2 <-> 4 end;
excl (1 * 2) <-> (3 -:- 4); 1 <-> 3; 2 <-> 4 end
```

❒ The above exclude declarations can be given in any order.

2.9.3 Scope of Exclude Declarations

❒ An exclude declaration is valid or effective only where an operator defined at the same location as the exclude declaration would be visible.

❒ This means in particular, that exclude declarations in the implementation of an operator are effective only there.

❒ Furthermore, if an operator is used recursively in its own implementation, it must be kept in mind that normally no exclusions for that operator have been defined yet, because they are usually only defined after the definition of the operator.

2.10 Characters

❒ The natural values of the type `char` are all Unicode characters.

❒ A character literal with type `char` consists of an arbitrary Unicode character between single quotation marks, e. g., `'x'`, `'ß'`, `'²'`, `'"'` oder `'''` (a single quotation mark between quotation marks).

❒ Even characters such as tab or line separator can be used directly in character literals.

❒ If a line break is encoded as two characters (`\r\n`) in the input file, however, such a line break between single quotation marks is not a correct character literal.

❒ Using the equality operator described in § 2.5, it is possible to test whether two `char` values are equal.

Conversions between Characters and Numbers

❒ For a `char` value `c`, `int c` returns the position of character `c` in the Unicode character set, while `char i` for an `int` value `i` returns the character at position `i` of the character set.

❒ If `c` or `i`, respectively, is an unnatural value or if there is no character at position `i` of the character set (especially if `i` is negative), the result is nil in each case.

❒ The operator `int`• has the same precedence as the change sign operator `-`•`*`, while the operator `char`• binds stronger than the negative test •`-` and weaker than the arithmetic operators.

2.11 Import of Modules

- ❐ `import modname` imports the module from the source file `filename.flx`.
  Here, `modname` is a name according to § 2.4 (e. g., `util` or `"../util"` with quotation marks) and `filename` is the character sequence described by it (`util` and `../util` without quotation marks, respectively, in the example).
  By giving multiple module names separated by commas, multiple modules can be imported immediately one after the other.
- ❐ The filename `filename.flx` (as long as it is not an absolute pathname) is always interpreted relatively to the directory in which the source file containing the import statement is located.

  If, for example, the file `A.flx` contains `import "lib/B"` and the file `lib/B.flx` contains in turn `import C`, then, from the viewpoint of `A.flx`, `lib/C.flx` is actually imported in `B.flx`.
- ❐ If different names denote the same file (e. g., `util` and `"./util"` or because one file is a link to another file or different names denote the same file because of the aforementioned interpretation of filenames), it is actually the same module.
- ❐ If there is a binary file `filename.flx.bin` which is newer than the source file `filename.flx`, it is used; otherwise the source file is compiled and – if it is correct and unambiguous – used and the binary file is updated.

- This means, that during the compilation of a program, all directly and indirectly imported modules are also (re)compiled if necessary. If any of the associated source files is erroneous or ambiguous, or if some module imports itself directly or indirectly, the whole compilation is aborted.
- If `flxc` is called with the option `-v` (verbose), it prints for each module whether its binary file is loaded or its source file is (re)compiled.
- `import modname` means:
  - Afterwards all (constants and) operators are visible which are visible at the end of the source file `filename.flx`, just as if the content of that file would appear instead of `import modname`.
    If a module is imported multiple times directly or indirectly, its source code is not replicated, however, because then new (constants and) operators would be created during the compilation of that code at every import point.
  - If the expression `import modname` is evaluated at run time, the expression defined by the code of the file `filename.flx` is evaluated, if it has not already been evaluated earlier.
    This means, that the code of a module is evaluated at most once, even if the module is imported multiple times directly or indirectly or if the expression `import modname` is evaluated multiple times (e. g., in a loop).
    The result value of an import expression is `true` if and only if at least one of the imported modules is actually evaluated.

2.12 Miscellaneous

❒ The output operator `print•` prints either an `int` or a `char` value and a terminating line separator.
A natural `int` value is printed in decimal notation, possibly with a leading minus sign; for a natural `char` value, the corresponding character is printed.
For an unnatural value, nothing (except the terminating line separator) is printed.

`print only x` prints only the `int` or `char` value `x` (possibly nothing) without the terminating line separator.

The result value is `true` if the output was successful, otherwise `false`.

❒ The nullfix operator `nil` represents a generic nil value whose type is deduced (and must be deducible) from the context of its use.
It this type cannot be unambiguously deduced, the program is either erroneous or ambiguous.

For example, `print nil` is ambiguous, because in this case the type of `nil` could be either `int` or `char`.
`nil = nil` is erroneous, because there are basically infinitely many possibilities for the type of the two uses of `nil`.

❒ According to § 1.5 „Everything is an expression“, even types are expressions, whose own type is the type `type`, which is therefore also called *meta type*.

For example, `int` and `bool` are constants of type `type`, while `int?` is an application of the postfix operator •`?` to the type `int` which returns the corresponding variable type.

2.13 Summary of Predefined Operators

2.13.1 Prefix, Infix, and Postfix Operators

❒ In the following tables, the precedence of the operators increases from group to group, i. e., in the operands of a particular operator the operators from all groups above are excluded.

Operator	*Meaning*	*Associativity*	*See §*
`•;•`	sequential evaluation	left	2.6.1

`•:•=•`
`•:•`
`•:=•`
	constant declaration		2.4

`print•`
`print only •`
	output		2.3

`?*•`	loop		2.6.4

`•?•!•`
`•=!•`
	branch
	assignment
		right	2.6.3
			2.7

`•=•`	equality test	non	2.5
`•-`	negative test		2.5

Operator	*Meaning*	*Associativity*	*See §*
`char•`	conversion from `int` to `char`		2.10
`•+•` `•-•`	addition subtraction	left	2.3

`•*•`
`•:•`
`•-:-•`
	multiplication
	division
	remainder
		left	2.3

`-•`
`int•`
`?•`
	change sign
	conversion from `char` to `int`
	variable query
			2.3
			2.10
			2.7

`•?`	variable type		2.7

2.13.2 Circumfix Operators

❒ Because the operands of the following operators are each surrounded by names of the operator and therefore no conflicts with other operators can arise, no operators are excluded in them (not even the sequential evaluation). Conversely, these operators are not excluded in the operands of any other operator.

❒ Note: The signature of an operator declaration, which is denoted here by the character • before the arrow, is actually not an operand of the declaration operator, but a sequence of names and parameter declarations.

Operator	*Meaning*	*See §*
`(•)`	parentheses	2.6.2
`•->(•=•)`	operator declaration	2.8
`excl•end`	exclude declaration	2.9
`vis•end`	visibility declaration	

2.13.3 Nullfix Operators

❒ The following operators do not have any operands and therefore again no conflicts with other operators can arise.
(Even though there is the character • after `import`, this does not actually denote an operand, but one or more names.)

Operator	*Meaning*	*See §*
`123` etc. `'x'` etc.	integer literals character literals	2.3 2.10
`nil`	generic nil value	2.12
`true` `false`	truth values	2.5

`int`
`char`
`bool`
`type`
	type of all integral numbers
	type of all Unicode characters
	type of all truth values
	meta type, i. e., type of all types
		2.3
		2.10
		2.5
		2.12

`import`•	import	2.11

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.