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).

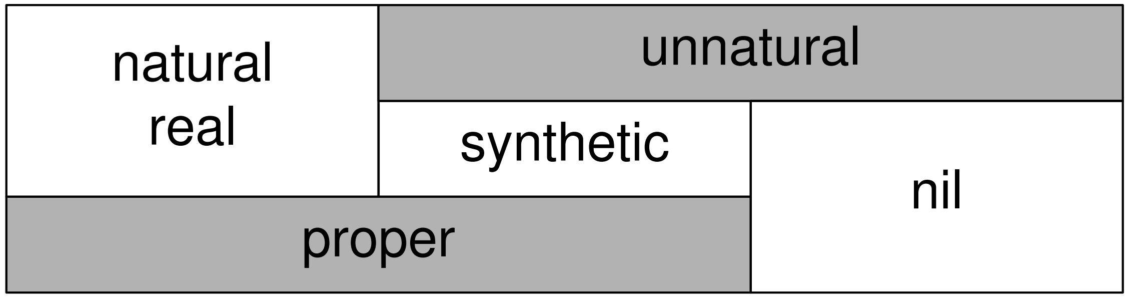
❐ Division by 0 yields the special value nil (abbreviation of the latin “nihil” meaning “nothing”), which is available for any type.

❐ 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:

```
excl (1+2)²; (1−2)²; (1*2)²; (1:2)²; (1−:−2)²; (−1)² 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