2 Vordefinierte Typen und Operatoren

2.1 Kommentare

- `$$` leitet einen *Zeilenkommentar* ein, der bis zum Ende der aktuellen Zeile geht.
- `$(` leitet einen *Blockkommentar* ein, der bis zum nächsten Vorkommen von `)$` auf der gleichen Verschachtelungsebene geht, d. h. Blockkommentare können beliebig geschachtelt werden.
- Zeilen- und Blockkommentare sind jedoch unabhängig voneinander.

2.2 Vorrang und Assoziativität von Operatoren

- Wenn ein Präfix-, Infix- oder Postfix-Operator *stärker bindet* (d. h. *höheren Vorrang* besitzt) als ein anderer, können Anwendungen des stärker bindenden Operators direkt (d. h. ohne Verwendung von Klammern) als Operanden des schwächer bindenden Operators verwendet werden, aber nicht umgekehrt.

  Beispielsweise können Multiplikationen und Divisionen nach der üblichen Punkt-vor-Strich-Regel direkt als Operanden von Additionen und Subtraktionen verwendet werden, während Additionen und Subtraktionen als Operanden von Multiplikationen und Divisionen geklammert werden müssen.

- Wenn ein Infix-Operator oder eine Gruppe solcher Operatoren mit gleichem Vorrang *links-* bzw. *rechtsassoziativ* ist, können direkte Anwendungen dieser Operatoren selbst nur in ihrem linken bzw. rechten Operanden, aber nicht in ihrem anderen Operanden verwendet werden.

  Da die arithmetischen Operatoren linksassoziativ sind, bedeutet `x - y - z` beispielsweise implizit `(x - y) - z` und nicht etwa `x - (y - z)`.

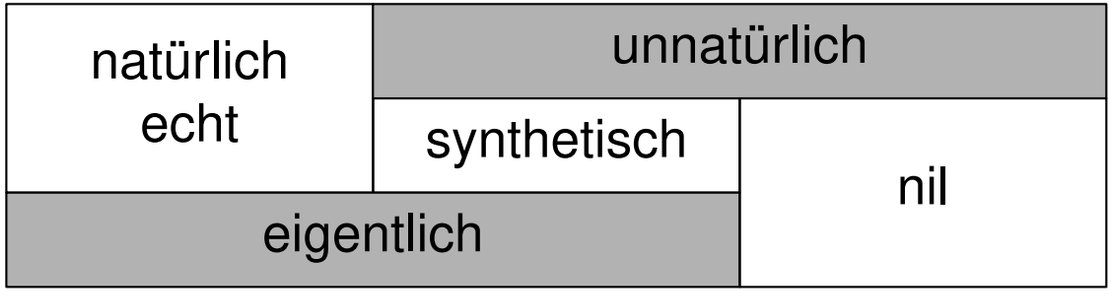
- Wenn ein Infix-Operator oder eine Gruppe solcher Operatoren mit gleichem Vorrang *nichtassoziativ* ist, können direkte Anwendungen dieser Operatoren selbst in keinem ihrer Operanden verwendet werden.

  Da der Gleichheitstest (vgl. § 2.5) nichtassoziativ ist, ist ein Ausdruck wie `x = y = z` fehlerhaft.

2.3 Ganze Zahlen und arithmetische Operationen

- Ganze Zahlen besitzen den Typ `int` und können prinzipiell beliebig groß werden.
- Eine beliebig lange Folge von Ziffern wie z. B. `12345678912345678 9` wird als Dezimalzahl interpretiert (auch wenn sie mit der Ziffer `0` beginnt).
- `print•` gibt eine ganze Zahl als Dezimalzahl, gefolgt von einem Zeilentrenner, auf der Standardausgabe aus (siehe auch § 2.12).
- Addition (`•+•`), Subtraktion (`•-•`), Multiplikation (`•*•`), Division (`•:•`), Divisionsrest (`•-:-•`) und Vorzeichenwechsel (`-•`) haben die aus anderen Programmiersprachen bekannte Bedeutung, und es gelten die üblichen Regeln für Vorrang und Assoziativität.
  `print•` bindet schwächer als diese Operatoren.
- Division schneidet eventuelle Nachkommastellen ab (Rundung in Richtung 0).
- Division durch 0 liefert den speziellen Wert nil (Kurzform des lateinischen „nihil“, das „nichts“ bedeutet), den es für jeden Typ gibt.
- Außerdem kann es für jeden Typ beliebig viele *synthetische* Werte geben, die u. a. durch Konstantendeklarationen ohne Initialisierung entstehen können (vgl. § 2.4).

- Damit gibt es grundsätzlich drei disjunkte Kategorien von Werten:
  - natürliche oder echte Werte (Zahlenwerte)
  - synthetische Werte
  - den Wert nil
- Synthetische Werte und nil werden zusammen auch als *unnatürliche Werte* bezeichnet, natürliche und synthetische Werte werden zusammen auch als *eigentliche Werte* bezeichnet:



- Wenn mindestens ein Operand einer arithmetischen Operation ein unnatürlicher Wert ist, ist das Resultat nil.
- Der Divisionsrest `x -:- y` ist immer gleich `x - x : y * y`.
  (Daraus folgt u. a., dass `x -:- 0` gleich nil ist.
  Für `x >= 0` und `y > 0` ist `x -:- y` gleich `x` modulo `y`, für andere Werte von `x` und `y` jedoch nicht; beispielsweise ist `-5 -:- 3` gleich `-2`, während `-5` modulo `3` gleich `1` ist.)

2.4 Konstantendeklarationen

- Eine Deklaration der Gestalt `name : type = init` definiert eine Konstante mit dem Namen `name` und dem Typ `type`, deren Wert sich durch Auswertung des Initialisierungsausdrucks `init` ergibt, z. B. `N : int = 10 * 10`.

- Tatsächlich kann `name` auch aus mehreren Namen bestehen, z. B. `line length : int = 10 * 10`.

- Jeder einzelne Name ist
  - entweder eine nicht leere Folge von Buchstaben und Ziffern des ASCII-Codes, die mit einem Buchstaben beginnt (und dann genau diese Zeichenfolge beschreibt)
  - oder eine nicht leere Folge beliebiger Unicode-Zeichen außer Zwischenraum innerhalb von doppelten Anführungszeichen (die dann die Zeichenfolge zwischen den Anführungszeichen beschreibt).
    Um ein doppeltes Anführungszeichen in eine solche Zeichenfolge zu integrieren, muss es verdoppelt werden; einfache Anführungszeichen können direkt verwendet werden.

- Zum Beispiel:

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

- Wenn der Typ der Konstanten fehlt, wird er automatisch aus dem Typ der Initialisierung ermittelt, z. B. `N := 10 * 10`.

- Wenn die Initialisierung fehlt (z. B. `S : int`), erhält die Konstante einen neuen synthetischen Wert, der verschieden von jedem anderen synthetischen Wert ist. Das ist primär für Variablentypen (vgl. § 2.7) und benutzerdefinierte Typen (vgl. § 4.3) wichtig, kann aber prinzipiell auch für Typen wie `int` verwendet werden.

- Der Resultatwert einer Konstantendeklaration ist der Wert der Konstanten.

- Eine Konstante ist ein nullstelliger Operator, d. h. ein Operator, der nur Namen, aber keine Operanden besitzt.

- Der Konstantendeklarationsoperator bindet schwächer als der Ausgabeoperator `print•`.

2.5 Wahrheitswerte und Vergleiche

- Die Wahrheitswerte `true` und `false` besitzen den Typ `bool`.
  Tatsächlich ist `true` einfach ein synthetischer Wert und `false` der nil-Wert des Typs `bool`.

- Für zwei Werte `x` und `y` desselben beliebigen Typs überprüft der Gleichheitstest `x = y`, ob die beiden Werte gleich sind, das heißt:
  - Beide stellen den gleichen natürlichen Wert dar
  - oder beide sind nil
  - oder beide stellen denselben synthetischen Wert dar.

  Als Resultat erhält man den entsprechenden Wahrheitswert `true` oder `false`.

- Für einen Wert `x` des Typs `int` überprüft der Negativtest `x -`, ob der Wert negativ ist, d. h. ob es ein natürlicher Wert ist, der kleiner als 0 ist. Als Resultat erhält man wiederum den entsprechenden Wahrheitswert `true` oder `false`.

- Der Gleichheitsoperator •=• ist nichtassoziativ und bindet stärker als der Ausgabeoperator `print`• und schwächer als der Negativtest •-, der wiederum schwächer bindet als die arithmetischen Operatoren.

- Mit diesen Grundoperatoren lassen sich alle weiteren bekannten Vergleichsoperatoren leicht selbst implementieren.

2.6 Ablaufsteuerung

2.6.1 Nacheinanderauswertung

- Semikolon ist ein linksassoziativer Infixoperator mit dem niedrigsten Vorrang zur Aneinanderreihung von Teilausdrücken, die nacheinander ausgewertet werden sollen, das heißt:

- Für Ausdrücke `x` und `y` mit beliebigen Typen `X` und `Y` wertet `x; y` die Ausdrücke `x` und `y` nacheinander aus und liefert den Wert von `y` (d. h. der Resultattyp ist `Y`).

2.6.2 Klammern

- Klammern können wie üblich zur expliziten Vorrangregelung verwendet werden, z. B. `(x + y) * z`.
  Das ist insbesondere notwendig, wenn ein Semikolon-Ausdruck als Operand eines Operators mit höherem Vorrang verwendet werden soll, z. B. `(print 1; x) + (print 2; y)`.
  (Anhand der Ausgaben von `print` kann man verfolgen, in welcher Reihenfolge die Teilausdrücke der Addition ausgewertet werden.)

- `(•)` ist ein normaler Operator, der seinen Operanden (mit einem beliebigen Typ `X`) auswertet und seinen Wert liefert (d. h. der Resultattyp ist `X`).

2.6.3 Fallunterscheidung

- Für einen Operanden `x` eines beliebigen Typs und zwei Operanden `y` und `z` desselben beliebigen Typs `T` liefert die elementare Fallunterscheidung `x ? y ! z` entweder den Wert von `y` oder den Wert von `z` (d. h. der Resultattyp ist `T`), je nachdem, ob der Wert von `x` ein eigentlicher Wert (also ungleich nil) oder nil ist.

- Nach der Auswertung von `x` wird dementsprechend nur entweder `y` oder `z` ausgewertet.

2.6.4 Schleife

- Für einen Operanden `x` eines beliebigen Typs wertet die elementare Schleife `?* x` den Operanden `x` immer wieder aus, bis seine Auswertung den Wert nil liefert.

- Als Resultatwert mit Typ `int` wird die Anzahl der Auswertungen von `x` geliefert.

2.6.5 Vorrang und Assoziativität

- Die Schleife bindet schwächer als der Ausgabeoperator `print•` und stärker als die Fallunterscheidung, die wiederum stärker als der Gleichheitstest bindet.

- Die Fallunterscheidung ist rechtsassoziativ, um „if-elseif-Ketten“ ohne Verwendung von Klammern zu ermöglichen, d. h. sie kann direkt in ihrem rechten (dritten), aber nicht in ihrem linken (ersten) Operanden verwendet werden.

  Im mittleren (zweiten) Operanden können – wie im Operanden von Klammern – Operatoren mit beliebigem Vorrang (einschließlich Nacheinanderauswertung) direkt verwendet werden.

2.6.6 Weitere Operatoren

- Mit diesen Grundoperatoren und der Möglichkeit, Operanden unausgewertet an Operatoren zu übergeben (vgl. § 3.8), lassen sich leicht beliebige weitere Operatoren zur Ablaufsteuerung wie z. B. `if•then•else•end` oder `while•do•end` definieren.

2.7 Variablen

- Für einen beliebigen Typ `T` bezeichnet der Typ `T?` *Variablen* mit *Inhaltstyp* `T`.

- Dementsprechend bezeichnet ein synthetischer Wert `x` eines solchen Typs `T?`, der gemäß § 2.4 mit einer Konstantendeklaration (!) `x : T?` erzeugt werden kann, eine eindeutige Speicherzelle, die einen veränderlichen Wert des Typs `T` (anfänglich nil) enthält, d. h. eine Variable mit Inhaltstyp `T`.

- Für eine Variable `x` des Typs `T?` liefert `?x` den aktuellen Wert (mit Typ `T`) der Variablen, und `x =! y` ersetzt diesen Wert durch den Wert `y` (ebenfalls mit Typ `T`). Der Resultatwert einer solchen Zuweisung ist der zugewiesene Wert mit Typ `T`.

- Wenn `x` der Wert nil (des Typs `T?`) ist, liefert `?x` ebenfalls nil (mit Typ `T`), und `x =! y` ist wirkungslos. Damit ist eine nil-Variable vergleichbar mit der Sonderdatei `/dev/null` in Unix-Systemen, die bei Leseoperationen EOF (also nichts) liefert und Schreiboperationen ignoriert.

- Weil = den Gleichheitsoperator bezeichnet, wurde für die Zuweisung das Symbol `=!` gewählt. Außerdem verdeutlicht das Ausrufezeichen den imperativen Charakter der Zuweisung. (Wenn es den Zuweisungsoperator nicht gäbe, wäre MOSTflexiPL keine imperative, sondern eine rein funktionale Programmiersprache.)

- Der Zuweisungsoperator •=!• ist rechtsassoziativ (um mehrfache Zuweisungen der Art `x1 =! x2 =! y` ohne Verwendung von Klammern zu ermöglichen) und besitzt den gleichen Vorrang wie die Fallunterscheidung •?•!• (sodass z. B. `x =! 1 + 2` als `x =! (1 + 2)` und nicht als `(x =! 1) + 2` interpretiert wird).

- Der Abfrageoperator `?`• besitzt den gleichen Vorrang wie der Vorzeichenwechsel −•.

2.8 Einfache Operatordeklarationen

- Eine *Operatordeklaration* hat die Form `sig -> (type = impl)`.
- Der Typ `type` ist der *Resultattyp* des deklarierten Operators, der Ausdruck `impl`, der den Typ `type` haben muss, stellt die *Implementierung* des Operators dar.
- Die *Signatur* `sig` ist eine nicht leere Folge von Namen (vgl. § 2.4) und *Parameterdeklarationen* der Form `(names : type)`, wobei `names` (wie bei einer Konstantendeklaration) wiederum eine nicht leere Folge von Namen des Parameters und `type` sein Typ ist.
  Die so deklarierten Parameter sind nur in der Implementierung `impl` sichtbar.
  Dasselbe gilt für Konstanten und Operatoren, die innerhalb der Implementierung deklariert werden (lokale Deklarationen).
  Der deklarierte Operator selbst ist bereits in seiner eigenen Implementierung sichtbar, um rekursive Anwendungen zu ermöglichen.
- Wenn man in der Signatur jeden Namen durch die Zeichenfolge ersetzt, die er gemäß § 2.4 beschreibt, und jeden Parameter durch einen zugehörigen Operanden, d. h. durch einen Ausdruck mit dem Typ dieses Parameters, entsteht eine *Anwendung* des Operators, d. h. ein Ausdruck, der als Typ den Resultattyp des Operators besitzt.
  Zwischen den Bestandteilen eines solchen Ausdrucks (Namen des Operators und Operanden) dürfen jeweils beliebig viele (also auch keine) Zwischenraumzeichen und Kommentare stehen.

- Eine solche Operatoranwendung wird ausgewertet, indem zunächst die Operanden von links nach rechts ausgewertet werden und jeder Parameter wie eine Konstante mit dem Wert des jeweiligen Operanden initialisiert wird.
  Anschließend wird die Implementierung des Operators ausgewertet, um das Resultat der Operatoranwendung zu ermitteln.

Beispiele

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

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

Funktionen mit unterschiedlicher 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 Ausschlussdeklarationen

2.9.1 Direkte Ausschlüsse

- Da es für benutzerdefinierte Operatoren keine vordefinierten Regeln für Vorrang und Assoziativität gibt (mit einer Ausnahme, siehe unten), ist ein Ausdruck wie z. B. `2 + 3 ^ 2` mehrdeutig: Er kann entweder als `2 + (3 ^ 2)` oder als `(2 + 3) ^ 2` interpretiert werden.

- Die zweite Interpretation kann mit einer Ausschlussdeklaration `excl (2 + 3) ^ 2 end` ausgeschlossen werden, wobei die Teilausdrücke `2` und `3` willkürliche Platzhalter für beliebige Operanden des Typs `int` sind.

Allgemeine Regeln

- Für jeden Parameter eines Operators gibt es eine Menge ausgeschlossener Operatoren, die nicht an der Spitze und eventuell auch nicht am linken oder rechten Rand (siehe unten) korrespondierender Operanden stehen dürfen.

- Wenn im Ausdruck zwischen `excl` und `end` ein geklammerter Teilausdruck als Operand für einen bestimmten Parameter auftritt, wird der Operator an der Spitze dieses Teilausdrucks zur Ausschlussmenge dieses Parameters hinzugefügt. Wenn der Ausdruck zwischen `excl` und `end` mehrere geklammerte Teilausdrücke enthält, gilt dies für jeden von ihnen.

- Wenn vor oder nach einem Parameter in der Signatur seines Operators kein Name des Operators steht, enthält seine Ausschlussmenge implizit die vordefinierte Nacheinanderauswertung `•;•`, sodass dieser Operator automatisch schwächer bindet als alle anderen Operatoren. Andernfalls ist die Ausschlussmenge eines Parameters zunächst leer.

Beispiele

- Quadrat bindet stärker als die Grundrechenarten einschließlich Vorzeichenwechsel, d. h. Anwendungen dieser Operatoren sind als Operanden des Quadratoperators verboten:

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

- Multiplikation, Division und Divisionsrest binden stärker als Addition und Subtraktion, d. h. Additionen und Subtraktionen sind als Operanden von Multiplikation, Division und Divisionsrest verboten (diese Ausschlüsse sind bereits vordefiniert):

```
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 und Subtraktion sowie Multiplikation, Division und Divisionsrest sind zusammen jeweils linksassoziativ, d. h. sie sind in ihren eigenen rechten Operanden verboten (diese Ausschlüsse sind ebenfalls bereits vordefiniert):

```
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
```

Linker und rechter Rand eines Ausdrucks

- Ein direkter oder indirekter *Teilausdruck* eines Ausdrucks befindet sich am *linken* bzw. *rechten Rand* dieses Ausdrucks, wenn er an der gleichen Stelle beginnt bzw. endet wie der gesamte Ausdruck.

  Beispielsweise befindet sich der Teilausdruck `1 * 2` am linken Rand und der Teilausdruck `3 : 4` am rechten Rand des Ausdrucks `1 * 2 + 3 : 4`.

  Am rechten Rand des Ausdrucks `print 1 * 2 + 3 : 4` befindet sich sowohl der Teilausdruck `1 * 2 + 3 : 4` als auch dessen Teilausdruck `3 : 4`, aber nicht der Teilausdruck `1 * 2`. Am linken Rand des Ausdrucks gibt es keinen Teilausdruck.

- Ein *Operator* befindet sich am linken bzw. rechten Rand eines Ausdrucks, wenn ein Teilausdruck am linken bzw. rechten Rand dieses Ausdrucks eine Anwendung dieses Operators ist.

  Dementsprechend befindet sich der Operator •`*`• am linken Rand und der Operator •`:`• am rechten Rand des Ausdrucks `1 * 2 + 3 : 4`.

  Am rechten Rand des Ausdrucks `print 1 * 2 + 3 : 4` befindet sich sowohl der Operator •`+`• als auch der Operator •`:`•, aber nicht der Operator •`*`•.

- Damit der Präfix-Operator `?*•`, wie aus den vorangegangenen Abschnitten hervorgeht, schwächer bindet als die arithmetischen Operatoren, ist er jeweils im linken Operanden der entsprechenden Infix-Operatoren ausgeschlossen.
  (Ein Ausschluss in ihrem rechten Operanden sowie im Operanden von Präfix-Operatoren ist nicht nötig, weil Ausdrücke wie z. B. `1 + ?* 2` oder `– ?* 3` auch ohne diese Ausschlüsse eindeutig sind.)

- Aber wenn sich Ausschlüsse grundsätzlich nur auf die Spitze der jeweiligen Operanden beziehen würden, wäre ein Ausdruck wie z. B. `1 + ?* 2 + 3` immer noch mehrdeutig, weil er sowohl als `1 + ?* (2 + 3)` als auch als `(1 + ?* 2) + 3` interpretiert werden kann. (Der Operator `?*•` befindet sich nicht an der Spitze des Teilausdrucks `1 + ?* 2`, sondern an dessen rechtem Rand.)

- Damit die zweite Interpretation durch den Ausschluss von `?*•` im linken Operanden der Addition ebenfalls ausgeschlossen ist, ist ein ausgeschlossener Operator ggf. auch am linken oder rechten Rand der jeweiligen Operanden verboten:

- Wenn sich der jeweilige Operand am linken Rand seines Ausdrucks befindet, ist der fragliche Operator auch am rechten Rand des Operanden verboten.

  Damit kann z. B. `1 + ?* 2 + 3` nur noch als `1 + ?* (2 + 3)` und nicht als `(1 + ?* 2) + 3` interpretiert werden.

- Wenn sich der jeweilige Operand am rechten Rand seines Ausdrucks befindet, ist der fragliche Operator auch am linken Rand des Operanden verboten.

  Wenn es einen rechtsassoziativen Potenzoperator •^• und einen Postfix-Operator •`@` mit Parameter- und Resultattyp `int` gäbe, der schwächer bindet und dementsprechend im rechten Operanden des Potenzoperators ausgeschlossen ist, dann würde diese Regel dafür sorgen, dass `1 ^ 2 @ ^ 3` nur als `(1 ^ 2) @ ^ 3` und nicht als `1 ^ (2 @ ^ 3)` interpretiert werden kann.

- Wenn sich der jeweilige Operand weder am linken noch am rechten Rand, sondern im „Inneren“ seines Ausdrucks befindet, ist der fragliche Operator nur an der Spitze des Operanden verboten. (Allerdings sind Ausschlüsse für derartige innere Operanden grundsätzlich nicht notwendig, um Vorrang oder Assoziativität von Operatoren festzulegen.)

2.9.2 Indirekte Ausschlüsse

- Da die Definition umfangreicher Vorranghierarchien mit direkten Ausschlüssen relativ umständlich ist, können sie mit indirekten Ausschlüssen einfacher, flexibler und modularer definiert werden.

- Ein indirekter Ausschluss definiert entweder, dass die Ausschlussmenge eines Parameters alle Operatoren der Ausschlussmenge eines anderen Parameters enthält, oder dass ein Operator überall ausgeschlossen ist, wo ein anderer Operator ausgeschlossen ist.

- Allgemein gilt:
  - Wenn im Ausdruck zwischen `excl` und `end` ein Teilausdruck der Gestalt `(left) -> (right)` auftritt, ist der Operator an der Spitze des Teilausdrucks `right` überall ausgeschlossen, wo der Operator an der Spitze des Teilausdrucks `left` ausgeschlossen ist.

- Wenn im Ausdruck zwischen `excl` und `end` ein Teilausdruck der Gestalt `left -> right` mit ungeklammerten Teilausdrücken `left` und `right` auftritt, enthält die Ausschlussmenge des durch den Teilausdruck `right` identifizierten Parameters alle Operatoren der Ausschlussmenge des durch den Teilausdruck `left` identifizierten Parameters.

  Damit ein Teilausdruck einen bestimmten Parameter identifiziert, muss der gleiche Teilausdruck bereits weiter vorne im Ausdruck zwischen `excl` und `end` als Operand für diesen Parameter auftreten. Falls er an mehreren Stellen auftritt, gilt das erste Auftreten. Wenn einer der Teilausdrücke `left` oder `right` an keiner Stelle weiter vorne auftritt, ist der gesamte Ausdruck `excl ... end` fehlerhaft.

- Wenn anstelle des Pfeils `->` ein Doppelpfeil `<->` verwendet wird, gelten die obigen Aussagen zusätzlich, wenn die Rollen von `left` und `right` vertauscht werden.

- Die so definierten Beziehungen zwischen Operatoren bzw. Parametern gelten auch für alle Ausschlüsse, die später direkt oder indirekt für diese Operatoren bzw. Parameter definiert werden.

- Die am Ende von § 2.9.1 formulierten Regeln über den Ausschluss von Operatoren am linken oder rechten Rand von Operanden gelten auch für Ausschlüsse, die auf diese Weise indirekt definiert sind.

Beispiele

- Addition ist linksassoziativ:

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

- Multiplikation ist linksassoziativ und bindet stärker als Addition:

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

- Vorzeichenwechsel bindet stärker als Multiplikation:

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

- Subtraktion hat dieselben Bindungseigenschaften wie Addition:

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

- Division und Divisionsrest haben dieselben Bindungseigenschaften wie Multiplikation:

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

- Die obigen Ausschlussangaben können in beliebiger Reihenfolge angegeben werden.

2.9.3 Gültigkeit von Ausschlussdeklarationen

- Eine Ausschlussdeklaration ist nur dort gültig bzw. wirksam, wo ein Operator, der an der gleichen Stelle wie die Ausschlussdeklaration definiert wäre, sichtbar wäre.

- Das bedeutet insbesondere, dass Ausschlussdeklarationen in der Implementierung eines Operators nur dort wirksam sind.

- Außerdem ist bei der rekursiven Verwendung eines Operators zu beachten, dass in seiner Implementierung meist noch keine Ausschlüsse für diesen Operator definiert sind, weil diese normalerweise erst nach der Definition des Operators definiert werden. (Wenn man sie in der Implementierung des Operators definieren würde, müsste man sie nach der Definition des Operators erneut definieren, damit sie auch außerhalb der Implementierung wirksam sind.)

2.10 Zeichen

- Der Typ `char` besitzt als natürliche Werte beliebige Unicode-Zeichen.
- Ein Zeichenliteral mit Typ `char` besteht aus einem beliebigen Unicode-Zeichen innerhalb einfacher Anführungszeichen, z. B. `'x'`, `'ß'`, `'²'`, `'"'` oder `'''` (ein einfaches Anführungszeichen innerhalb von Anführungszeichen), und bezeichnet das Zeichen zwischen den Anführungszeichen.
- Selbst Zeichen wie Tabulator oder Zeilentrenner können direkt in Zeichenliteralen verwendet werden.
- Wenn ein Zeilenumbruch in der Eingabedatei durch zwei Zeichen (`\r\n`) kodiert wird, ist ein solcher Zeilenumbruch innerhalb einfacher Anführungszeichen jedoch kein korrektes Zeichenliteral.
- Mit dem in § 2.5 beschriebenen Gleichheitsoperator kann überprüft werden, ob zwei `char`-Werte gleich sind.

Umwandlungen zwischen Zeichen und Zahlen

- Für einen `char`-Wert `c` liefert `int c` die Position des Zeichens `c` im Unicode-Zeichensatz, während `char i` für einen `int`-Wert `i` das Zeichen an Position `i` des Zeichensatzes liefert.

- Wenn `c` bzw. `i` ein unnatürlicher Wert ist oder wenn es kein Zeichen an Position `i` des Zeichensatzes gibt (insbesondere wenn `i` negativ ist), wird jeweils nil geliefert.

- Der Operator `int•` besitzt den gleichen Vorrang wie der Vorzeichenwechsel `-•`, während der Operator `char•` stärker bindet als der Negativtest `•-` und schwächer als die arithmetischen Operatoren.

2.11 Import von Modulen

- `import` `modname` importiert das Modul aus der Quelldatei `filename.flx`. Dabei ist `modname` ein Name gemäß § 2.4 (z. B. `util` oder `"../util"` mit Anführungszeichen) und `filename` die Zeichenfolge, die von diesem Namen beschrieben wird (im Beispiel `util` bzw. `../util` ohne Anführungszeichen). Durch Angabe mehrerer Modulnamen, die durch Kommas getrennt sind, können mehrere Module unmittelbar nacheinander importiert werden.
- Der Dateiname `filename.flx` wird (sofern es sich nicht um einen absoluten Pfadnamen handelt) immer relativ zu dem Verzeichnis interpretiert, in dem sich die Quelldatei befindet, die die Importangabe enthält.

  Wenn die Datei `A.flx` z. B. `import "lib/B"` und die Datei `lib/B.flx` wiederum `import C` enthält, wird in `B.flx` aus Sicht von `A.flx` tatsächlich `lib/C.flx` importiert.
- Wenn unterschiedliche Namen dieselbe Datei bezeichnen (z. B. `util` und `"./util"` oder weil eine Datei ein Verweis auf eine andere Datei ist oder weil durch die zuvor genannte Interpretation von Dateinamen verschiedene Namen dieselbe Datei bezeichnen), handelt es sich um dasselbe Modul.
- Wenn es eine Binärdatei `filename.flx.bin` gibt, die neuer als die Quelldatei `filename.flx` ist, wird sie verwendet; andernfalls wird die Quelldatei übersetzt und – sofern sie korrekt und eindeutig ist – verwendet und die Binärdatei aktualisiert.

- Das heißt, dass beim Übersetzen eines Programms alle direkt und indirekt importierten Module bei Bedarf automatisch neu übersetzt werden. Wenn dabei irgendeine Quelldatei fehlerhaft oder mehrdeutig ist oder wenn sich ein Modul direkt oder indirekt selbst importiert, wird der gesamte Übersetzungsvorgang abgebrochen.
- Wenn man beim Aufruf von `flxc` die Option `-v` (verbose) angibt, wird für jedes Modul ausgegeben, ob seine Binärdatei geladen oder seine Quelldatei neu übersetzt wird.
- `import modname` bedeutet:
  - Anschließend sind alle (Konstanten und) Operatoren sichtbar, die am Ende der Quelldatei `filename.flx` sichtbar sind, wie wenn der Inhalt dieser Datei anstelle von `import modname` stehen würde.
    Wenn ein Modul direkt oder indirekt mehrmals importiert wird, wird sein Quellcode aber nicht repliziert, weil sonst beim Übersetzen dieses Codes an jeder Importstelle neue (Konstanten und) Operatoren entstehen würden.
  - Bei der Auswertung des Ausdrucks `import modname` zur Laufzeit wird der Ausdruck, der durch den Code der Datei `filename.flx` definiert wird, ausgewertet, sofern er nicht bereits früher ausgewertet wurde.
    Das heißt, der Code eines Moduls wird höchstens einmal ausgewertet, selbst wenn das Modul direkt oder indirekt mehrmals importiert wird oder wenn der Ausdruck `import modname` (z. B. in einer Schleife) mehrmals ausgewertet wird.
    Der Resultatwert eines Importausdrucks ist genau dann `true`, wenn mindestens eines der importierten Module tatsächlich ausgewertet wurde.

2.12 Sonstiges

- Der Ausgabeoperator `print•` gibt entweder einen `int`- oder einen `char`-Wert und einen abschließenden Zeilentrenner aus.
  Ein natürlicher `int`-Wert wird dezimal, ggf. mit vorangestelltem Minuszeichen ausgegeben, für einen natürlichen `char`-Wert wird das entsprechende Zeichen ausgegeben.
  Für einen unnatürlichen Wert wird nichts (außer dem abschließenden Zeilentrenner) ausgegeben.

  `print only x` gibt nur den `int`- bzw. `char`-Wert `x` (ggf. nichts) ohne abschließenden Zeilentrenner aus.

  Der Resultatwert ist `true`, wenn die Ausgabe fehlerfrei funktioniert hat, sonst `false`.

- Der Nullfix-Operator `nil` repräsentiert einen generischen Nilwert, dessen Typ sich aus dem Verwendungskontext ergibt bzw. eindeutig ergeben muss.
  Wenn sich dieser Typ nicht eindeutig ermitteln lässt, ist das Programm entweder mehrdeutig oder fehlerhaft.

  Beispielsweise ist `print nil` mehrdeutig, weil `nil` hier sowohl Typ `int` als auch Typ `char` haben könnte.
  `nil = nil` ist fehlerhaft, weil es für den Typ der beiden Verwendungen von `nil` prinzipiell unendlich viele verschiedene Möglichkeiten gibt.

- Auch Typen sind – gemäß § 1.5 „Alles ist ein Ausdruck“ – Ausdrücke, die ihrerseits den Typ `type` besitzen, der deshalb auch als *Metatyp* bezeichnet wird.

  Beispielsweise sind `int` und `bool` Konstanten mit Typ `type`, während `int?` eine Anwendung des Postfix-Operators •`?` auf den Typ `int` darstellt, die als Resultat den zugehörigen Variablentyp liefert.

2.13 Zusammenstellung der vordefinierten Operatoren

2.13.1 Präfix-, Infix- und Postfix-Operatoren

❒ In den nachfolgenden Tabellen nimmt der Vorrang der Operatoren gruppenweise von oben nach unten zu, d. h. in den Operanden eines Operators sind alle Operatoren weiter oben stehender Gruppen ausgeschlossen.

Operator	*Bedeutung*	*Assoziativität*	*Siehe §*
`•;•`	Nacheinanderauswertung	links	2.6.1

`•:•=•`
`•:•`
`•:=•`
	Konstantendeklaration		2.4

`print•` `print only •`	Ausgabe		2.3
`?*•`	Schleife		2.6.4
`•?•!•` `•=!•`	Fallunterscheidung Zuweisung	rechts	2.6.3 2.7
`•=•`	Gleichheitstest	nicht	2.5
`•-`	Negativtest		2.5

Operator	*Bedeutung*	*Assoziativität*	*Siehe §*
`char•`	Umwandlung von `int` nach `char`		2.10
`•+•` `•-•`	Addition Subtraktion	links	2.3

`•*•`
`•:•`
`•-:-•`
	Multiplikation
	Division
	Divisionsrest
		links	2.3

`-•`
`int•`
`?•`
	Vorzeichenwechsel
	Umwandlung von `char` nach `int`
	Variablenabfrage
			2.3
			2.10
			2.7

`•?`	Variablentyp		2.7

2.13.2 Zirkumfix-Operatoren

- Da die Operanden der nachfolgenden Operatoren jeweils von Namen des Operators umgeben sind und somit bei ihrer Verwendung keine Konflikte mit anderen Operatoren entstehen können, sind in ihnen keine Operatoren ausgeschlossen (nicht einmal die Nacheinanderauswertung). Umgekehrt sind diese Operatoren auch nicht in Operanden anderer Operatoren ausgeschlossen.

- Beachte: Die Signatur einer Operatordeklaration, die hier durch das Zeichen • vor dem Pfeil angedeutet wird, ist tatsächlich kein Operand des Deklarationsoperators, sondern eine Folge von Namen und Parameterdeklarationen (vgl. § 2.8).

Operator	*Bedeutung*	*Siehe §*
`(•)`	Klammern	2.6.2
`•->(•=•)`	Operatordeklaration	2.8
`excl•end`	Ausschlussdeklaration	2.9
`vis•end`	Sichtbarkeitsdeklaration	

2.13.3 Nullfix-Operatoren

❒ Die nachfolgenden Operatoren haben keine Operanden, sodass auch hier keine Konflikte mit anderen Operatoren entstehen können.
(Obwohl nach `import` das Zeichen • angegeben ist, steht dort kein Operand, sondern ein oder mehrere Namen.)

Operator	*Bedeutung*	*Siehe §*
`123` etc.	Ganzzahlige Literale	2.3
`'x'` etc.	Zeichenliterale	2.10
`nil`	Generischer Nilwert	2.12
`true` `false`	Wahrheitswerte	2.5
`int`	Typ aller ganzen Zahlen	2.3
`char`	Typ aller Unicode-Zeichen	2.10
`bool`	Typ aller Wahrheitswerte	2.5
`type`	Metatyp, d. h. Typ aller Typen	2.12
`import•`	Import	2.11