Dokumentation des Compilers für die erweiterbare Programmiersprache

LOWFLEXIPI

Prof. Dr. Christian Heinlein

15. November 2022

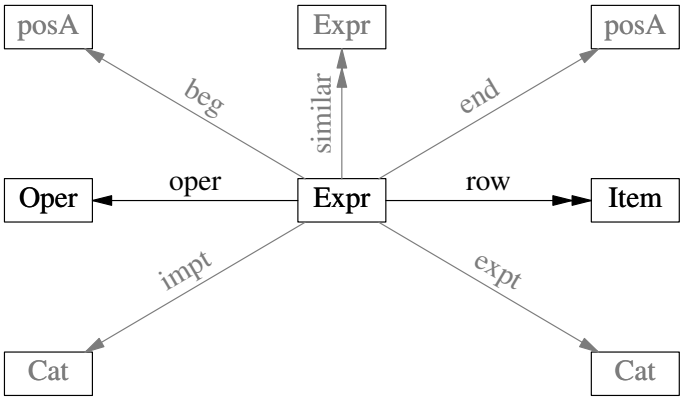
1 Zentrale Datenstrukturen und Hilfsfunktionen

Alle wesentlichen Datenstrukturen des Compilers werden mit offenen Typen und Sequenzen der Bibliothek *lib*$_{\texttt{CH}}$ realisiert. Die Namen aller Attribute enden mit einem Unterstrich, erstens, damit sie im Quelltext besser als Attribute erkennbar sind, und zweitens, um Namenskonflikte mit ansonsten gleichnamigen Variablen zu vermeiden. In der nachfolgenden Beschreibung sowie in den Kommentaren im Quelltext werden diese Unterstriche aber meist weggelassen. In den Abbildungen werden die Namen virtueller Attribute mit Klammern umgeben.

1.1 Ausdrücke

Ein Ausdruck (Typ `Expr`) wie z. B. `a+b` ist die Anwendung eines Operators (Attribut `oper`) auf prinzipiell beliebig viele Teilausdrücke, die als Operanden bezeichnet werden und direkt oder indirekt in einer Reihe von Einträgen (siehe unten) gespeichert werden (Attribut `row`). Die Blätter der so entstehenden Baumstruktur, d. h. Ausdrücke ohne weitere Operanden, heißen atomare Ausdrücke.

Jeder Ausdruck besitzt eine Anfangs- und eine Endposition im Quelltext (Attribute `beg` und `end`), einen Import- und einen Exportkatalog (Attribute `impt` und `expt`) sowie ggf. eine Menge ähnlicher Ausdrücke (Attribut `similar`).



1.2 Operatoren

Ein Operator (Typ `Oper`) besitzt zumindest eine Signatur (Attribut `sig`).

Weitere optionale Bestandteile sind:

- eine Initialisierung (Attribut `init`) oder eine Implementierung (Attribut `impl`);
- die Kennzeichnung als statischer Operator (Attribut `stat`);
- Import-, Export- und Ausschlussangaben (Attribute `impt`, `expt` und `excls`);
- ein Verweis auf den vorigen Operator in einer Kette von Redeklarationen (Attribut `prev`).

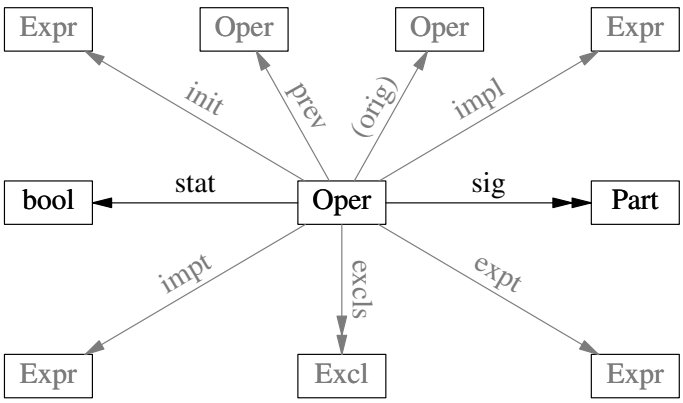
Das virtuelle Attribut `orig`, das nur abgefragt, aber nicht direkt verändert werden kann, liefert den ursprünglichen Operator in einer Kette von Redeklarationen.

Eine Konstante ist ein statischer Operator, dessen Signatur (siehe unten) nur aus Namen besteht und der höchstens eine Initialisierung, aber keine Implementierung besitzt.

Der Unterschied zwischen Initialisierung und Implementierung ist wesentlich: Eine Initialisierung wird zur Laufzeit einmalig ausgewertet, wenn die Deklaration des Operators ausgewertet wird. Eine Implementierung hingegen wird normalerweise bei jeder Anwendung des Operators erneut auswertet (bei einem statischen Operator allerdings für jede Kombination von Parameterwerten nur einmal). Zum Beispiel:

```
x : int = ......;
y -> (int = ......);
z => (int = ......)
```

Die Initialisierung der Konstanten `x` wird einmalig bei der Ausführung ihrer Deklaration ausgewertet; jede spätere Anwendung von `x` liefert dann den hier berechneten Wert. Die Implementierung des dynamischen Operators `y` wird bei jeder Anwendung von `y` erneut ausgewertet (und kann deshalb prinzipiell jedes Mal einen anderen Wert liefern). Die Implementierung des statischen Operators `z` wird einmalig bei der ersten Anwendung von `z` ausgewertet; alle weiteren Anwendungen liefern dann ebenfalls den hier berechneten Wert.



1.3 Signaturen und ihre Bestandteile

Eine Signatur (Typ `Sig`) ist eine Folge von einem oder mehreren Teilen (Typ `Part`), d. h. der Typ `Sig` ist lediglich ein Synonym für `seq<Part>`.

Ein solches Signaturteil ist:

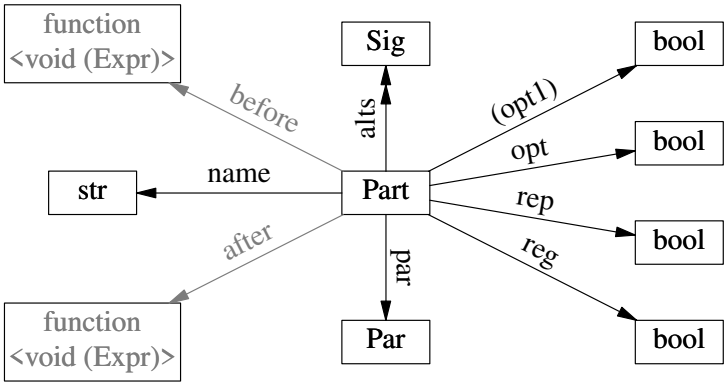
- entweder ein Name (Attribut `name`)
- oder ein Parameter (Attribut `par`), bei dem es sich um einen Operator handelt, d. h. der Typ `Par` ist lediglich ein Synonym für `Oper`,
- oder eine Klammer mit einer oder mehreren Alternativen (Attribut `alts`), die wiederum Signaturen sind. (Da jede dieser Signaturen bereits eine Folge von Teilen ist, ist `alts` faktisch eine Folge von Folgen von Teilen.)

Abhängig vom Wert des Attributs `reg` wird ein Name entweder als feste Zeichenfolge (Attributwert `false`) oder als regulärer Ausdruck (Attributwert `true`) interpretiert.

Eine Klammer kann ggf. optional und/oder wiederholbar sein (Attribute `opt` und `rep`).

- Eine Klammer, die weder optional noch wiederholbar ist (runde Klammern in der konkreten Syntax von MOSTflexiPL), muss bei der Abarbeitung der Signatur durch den Parser genau einmal durchlaufen werden, d. h. es muss genau eine der Alternativen durchlaufen werden.
- Eine optionale, aber nicht wiederholbare Klammer (eckige Klammern) muss null- oder einmal durchlaufen werden, d. h. es darf höchstens eine der Alternativen durchlaufen werden.
- Eine optionale und wiederholbare Klammer (geschweifte Klammern) muss null- oder mehrmals durchlaufen werden, d. h. es dürfen beliebig viele der Alternativen nacheinander durchlaufen werden.
- Klammern, die wiederholbar, aber nicht optional sind, sind momentan in der konkreten Syntax von MOSTflexiPL nicht vorgesehen.
- Da runde Klammern mit nur einer Alternative unnötig sind, muss eine runde Klammer immer mindestens zwei Alternativen enthalten, d. h. runde Klammern werden nur zur Klammerung von Alternativen benötigt. Eckige und geschweifte Klammern enthalten aber häufig nur eine Alternative.
- Das virtuelle Attribut `opt1` das nur abgefragt, aber nicht verändert werden kann, kennzeichnet einfache Optionsklammern, d. h. eckige Klammern mit genau einer Alternative, die an manchen Stellen besonders behandelt werden müssen.

Jedes Signaturteil kann eine Prolog- (Attribut `before`) und eine Epilogfunktion (Attribut `after`) besitzen, die vor bzw. nach der Verarbeitung dieses Teils durch den Parser mit dem entsprechenden Ausdruck als Parameter aufgerufen wird.

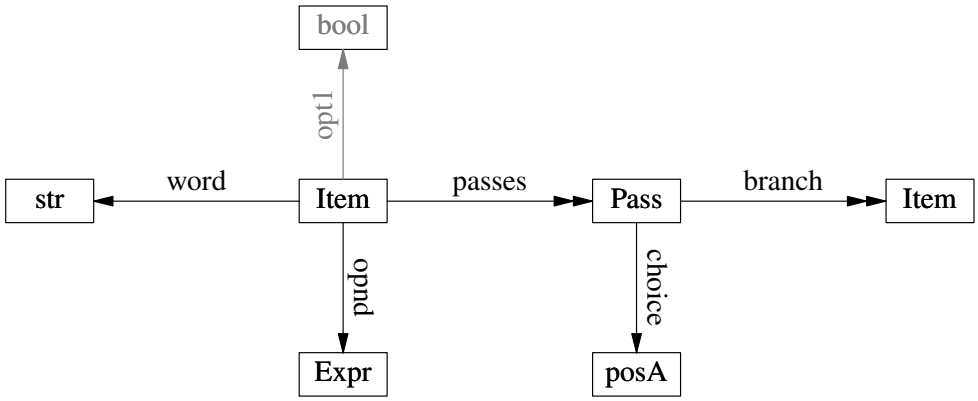


1.4 Reihen und ihre Einträge

Eine Reihe (Typ `Row`) ist eine Folge von einem oder mehreren Einträgen (Typ `Item`), d. h. der Typ `Row` ist lediglich ein Synonym für `seq<Item>`.

Ein solcher Eintrag in einer Reihe eines Ausdrucks mit einem bestimmten Operator bezieht sich immer auf genau ein Teil der Signatur dieses Operators:

- Wenn dieses Teil ein Name ist, enthält der zugehörige Eintrag die Zeichenfolge aus der Anwendung des Operators, die zu diesem Namen gehört (Attribut `word`).
  Wenn der Name als feste Zeichenfolge interpretiert wird, stimmt die Zeichenfolge `word` mit diesem Namen überein, sodass der Eintrag in diesem Fall redundant ist. Aber um die oben genannte Korrespondenz zwischen Einträgen und Signaturteilen aufrechtzuerhalten, wird er trotzdem gespeichert.
  Wenn der Name als regulärer Ausdruck interpretiert wird, enthält `word` eine (möglichst lange) Zeichenfolge, die auf dieses Muster passt.
- Wenn das Teil ein Parameter ist, enthält der zugehörige Eintrag den Operanden aus der Anwendung des Operators, der zu diesem Parameter gehört (Attribut `opnd`).
- Wenn das Teil eine Klammer ist, enthält der zugehörige Eintrag die Informationen über die prinzipiell beliebig vielen Durchläufe durch diese Klammer (Attribut `passes`) sowie die Information, ob es sich um eine einfache Optionsklammer handelt (Attribut `opt1`, das hier nicht virtuell ist).
  Jeder solche Durchlauf (Typ `Pass`) enthält die Position der jeweils durchlaufenen Alternative in der Folge aller Alternativen (Attribut `choice`) sowie rekursiv die zu dieser Alternative gehörende Reihe von Einträgen (Attribut `branch`).
  Eine Besonderheit eckiger Klammern ist, dass für sie auch dann ein „Pseudodurchlauf“ vorhanden sein kann, wenn die Klammer gar nicht durchlaufen wurde, weil die in ihr definierten Parameter in der Implementierung des Operators trotzdem sichtbar sein können. (Diese Parameter besitzen dann ggf. wohldefinierte Ersatzwerte.) In diesem Fall enthält das Attribut `choice` den Wert `0*A`, um anzuzeigen, dass keine der Alternativen durchlaufen wurde.
  Wenn es bei einer geschweiften Klammer keine Durchläufe gibt, ist `passes` jedoch eine leere Sequenz.



1.5 Korrespondenz zwischen Signaturteilen und Einträgen von Reihen

Wie oben bereits angedeutet, sind Signaturen mit Teilen und Reihen mit Einträgen weitgehend gleich aufgebaut:

Korrespondierende Typen und Attribute	
`Sig`	`Row`
`Part`	`Item`
`sig`	`row`
`name`	`word`
`par`	`opnd`

Um mehrere Durchläufe durch eine geschweifte Klammer repräsentieren zu können, benötigt ein Eintrag allerdings zusätzlich die Indirektion über `passes`.

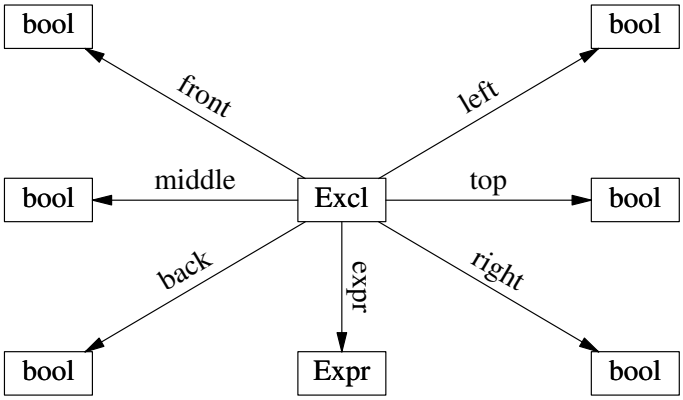
1.6 Ausschlussangaben

Eine Ausschlussangabe (Typ `Excl`) eines Parameters besteht im wesentlichen aus einem Ausdruck (Attribut `expr`), der vom Parser nach bestimmten Regeln interpretiert bzw. expandiert wird, um daraus einen Katalog von Operatoren zu ermitteln, die in den zu diesem Parameter gehörenden Operanden verboten sind. In vielen Fällen ist dieser Katalog der Exportkatalog dieses Ausdrucks, in dem manche Operatoren wiederum nach bestimmten Regeln interpretiert werden.

Die Attribute `front`, `middle` und `back` geben an, ob die Ausschlussangabe für den vorderen und/oder einen mittleren und/oder den hinteren Operanden eines Ausdrucks gilt.

Die Attribute `left`, `top` und `right` geben an, ob sich die Ausschlussangabe auf den linken Rand und/oder die Spitze und/oder den rechten Rand eines Operanden bezieht.

Diese Begriffe sowie weitere Details werden in der Beschreibung des Parsers erläutert.



1.7 Operatorkataloge

Ein Operatorkatalog (kurz Katalog) ist eine teilweise geordnete Menge von Operatoren: Wenn zwei Operatoren gemäß eines noch zu definierenden Kriteriums äquivalent sind, ist ihre Reihenfolge wichtig, weil der früher definierte Operator (der sich dann im Katalog weiter vorne befindet) vom später definierten (der sich weiter hinten befindet) verdeckt wird. Abgesehen davon ist die Reihenfolge der

Operatoren in einem Katalog aber unwichtig, d. h. Kataloge sind (logisch) gleich, wenn sie grundsätzlich die gleichen Operatoren und äquivalente Operatoren in der gleichen Reihenfolge enthalten.

Jeder Ausdruck muss u. a. seinen Importkatalog, d. h. die Menge der in ihm sichtbaren Operatoren kennen. Da diese Kataloge relativ groß sein können und viele Ausdrücke den gleichen Katalog besitzen, wäre es ungeschickt, die Menge dieser Operatoren in jedem Ausdruck redundant zu speichern.

Deshalb werden Kataloge stellvertretend durch Objekte des offenen Typs `Cat` repräsentiert, die nur sehr wenig Speicher benötigen. Über das virtuelle Attribut `opers`, das nur abgefragt, aber nicht verändert werden kann, erhält man die Operatoren eines Katalogs als Sequenz (in der äquivalente Operatoren in der richtigen Reihenfolge enthalten sind).

Außerdem werden Kataloge als Teil des Schlüssels der Tabelle von Archiven verwendet, die vom Parser sehr häufig abgefragt wird, d. h. Kataloge müssen sehr oft auf Gleichheit getestet werden, was grundsätzlich relativ aufwendig ist.

Deshalb gibt es zusätzlich ein globales Verzeichnis aller bis jetzt erzeugten Kataloge, das von der Funktion `cat` wie folgt verwaltet wird: Für eine nichtleere Sequenz `os` von Operatoren liefert `cat(os)` ein eindeutiges Objekt des Typs `Cat`, das einen Katalog mit diesen Operatoren in dieser Reihenfolge repräsentiert, d. h. wenn die Sequenzen `os1` und `os2` logisch den gleichen Katalog darstellen, liefern `cat(os1)` und `cat(os2)` dasselbe Objekt, auch wenn die Sequenzen `os1` und `os2` die Operatoren teilweise in unterschiedlicher Reihenfolge enthalten. Damit können Objekte des Typs `Cat` direkt als (Teil von) Tabellenschlüssel(n) verwendet werden, weil sie nur dann technisch (gemäß `operator==`) gleich sind, wenn sie auch logisch gleich sind.

Das bedeutet aber auch, dass Objekte des Typs `Cat` ausschließlich von der Funktion `cat` erzeugt werden sollten.

Für eine leere Sequenz `os` liefert `cat(os)` (oder einfach `cat()`) ein nil-Objekt des Typs `Cat`, damit der leere Katalog mit einem nicht vorhandenen Katalog (den man z. B. beim Abfragen eines nicht vorhandenen Attributwerts erhält) übereinstimmt.

Für einen Operator `oper` und einen Katalog `cat` überprüft `oper<<cat` (das Symbol `<<` erinnert mit etwas Phantasie an den mathematischen Operator $\in$), ob `oper` im Katalog `cat` enthalten ist oder nicht.

Für Kataloge `cat1` und `cat2` fügt `cat1+=cat2` die Operatoren von `cat2` zum Katalog `cat1` hinzu, d. h. `cat1` enthält anschließend die Vereinigung der beiden Kataloge.

`intersect(cat1, cat2, diff)` bestimmt sowohl die Schnittmenge als auch die Differenzmenge der Kataloge `cat1` und `cat2`. Die Schnittmenge wird als neuer Katalog zurückgeliefert, während die Differenzmenge über den Referenzparameter `diff` als Sequenz von Operatoren zurückgegeben wird.

1.8 Durchlaufen von Reihen

Die Funktion `trav` ist eine sehr flexible Hilfsfunktion für viele verschiedene Zwecke, die alle im Kern ähnlich sind, weil immer eine Reihe von Einträgen inklusive aller direkt und indirekt enthaltenen Teilreihen durchlaufen werden muss. Im Detail gibt es jedoch zahlreiche Varianten:

- Die Reihe kann vorwärts oder rückwärts durchlaufen werden.
- Optional wird gleichzeitig die zur Reihe gehörende Signatur oder aber eine zweite Reihe, die zur gleichen Signatur gehört, durchlaufen.

- Die übergebene Funktion wird häufig nur für die atomaren Einträge der Reihe aufgerufen, gelegentlich aber auch für die Klammereinträge.
- Der übergebenen Funktion wird bei Bedarf auch der Pfad zum jeweiligen Eintrag übergeben, d. h. eine Folge von Positionen mit folgender Bedeutung:
  - Die erste Position bezeichnet den Eintrag in der zu durchlaufenden Hauptreihe.
  - Wenn dieser Eintrag eine Klammer ist, bezeichnet die nächste Position den Durchlauf durch diese Klammer und die übernächste Position wiederum den Eintrag in diesem Durchlauf.
  - Usw.
- Beim Durchlaufen wird entweder nach etwas gesucht und der Durchlauf abgebrochen, sobald es gefunden wurde, oder es wird für jeden Eintrag eine bestimmte Aktion ausgeführt, oder es wird eine teilweise modifizierte Kopie der Reihe erstellt (hierfür gibt es eine weitere Funktion `trans`, die ihrerseits `trav` verwendet).
- Von Klammern werden entweder alle Durchläufe durchlaufen oder jeweils nur der letzte Durchlauf, sofern sich die Klammer am Ende ihrer Reihe befindet oder eine einfache Optionsklammer ist.

Die Funktion `trans` verwendet die Funktion `trav`, um eine teilweise modifizierte Kopie eines Ausdrucks zu erzeugen. Die übergebene Funktion wird normalerweise für alle atomaren Einträge der Hauptreihe des Ausdrucks aufgerufen. Wenn sie nil liefert, wird der Eintrag unverändert in den kopierten Ausdruck übernommen, andernfalls wird er durch den von der Funktion gelieferten Eintrag ersetzt. Einträge für Klammern werden, wenn möglich, unverändert übernommen, aber wenn eine Klammer direkt oder indirekt einen geänderten Eintrag enthält, muss ihr Eintrag kopiert werden. Wenn die übergebene Funktion für alle Einträge nil liefert, wird der Ausdruck unverändert zurückgeliefert.

2 Scanner

Im Vergleich zum Scanner eines Compilers für eine „gewöhnliche“ Programmiersprache, der die Eingabe in zahlreiche unterschiedliche Tokens (wie z. B. Literale für unterschiedliche Typen, vordefinierte Schlüsselwörter und Operatoren) zerlegen muss, ist der Scanner für MOSTflexiPL relativ degeneriert, sodass man eigentlich von „scannerless parsing“ sprechen kann.

Die Funktion `scan_open` erhält als Parameter den Namen einer Quelldatei, öffnet diese Datei und liest ihren gesamten Inhalt in die Variable `scan_str` ein, in der er anschließend bequem verarbeitet werden kann.

Die nachfolgenden Funktionen erhalten als ersten Parameter jeweils eine Position des Typs `posA`, die angibt, ab welcher Stelle der Eingabesequenz `scan_str` etwas gelesen werden soll. Nach einem erfolgreichen Aufruf gibt die Position an, wie weit gelesen wurde. Nach einem nicht erfolgreichen Aufruf ist die Position undefiniert. Die meisten Funktionen zeigen außerdem durch einen Resultatwert des Typs `bool` an, ob sie erfolgreich waren oder nicht, d. h. ob sie das Gewünschte lesen konnten oder nicht.

Die Funktion `scan_white` liest so viel Zwischenraum wie möglich, d. h. eine möglichst lange Folge von Zeichen mit der Eigenschaft `isspace`. Dies ist immer erfolgreich.

Die Funktion `scan_exact` versucht, exakt die als Parameter übergebene Zeichenkette zu lesen.

Die Funktion `scan_match` versucht, eine Zeichenkette zu lesen, die auf den als Parameter übergebenen regulären Ausdruck passt. Wenn dies erfolgreich ist, wird die gelesene Zeichenkette in der als Pa-

rameter übergebenen Variablen `word` gespeichert. Andernfalls ist der Wert dieser Variablen undefiniert.

Die Funktion `scan_eof` versucht sozusagen, das Ende der Eingabe zu lesen, d. h. sie überprüft, ob sich die übergebene Position auf das Ende der Eingabesequenz bezieht.

3 Parser

3.1 Aktuelles Signaturteil und aktueller Eintrag eines Ausdrucks

Für einen Ausdruck, der vom Parser gerade schrittweise konstruiert wird, bezeichnet das Attribut `currpart` bzw. `curritem` das Teil in der Signatur (des Operators) des Ausdrucks bzw. den korrespondierenden Eintrag des Ausdrucks, der als letztes mit Information gefüllt wurde (wenn dies bereits abgeschlossen ist) bzw. der als nächstes gefüllt werden muss (wenn damit noch nicht begonnen wurde) bzw. der momentan gefüllt wird (wenn dies gerade erfolgt).

Diese Objekte können wie folgt gefunden werden (vgl. Funktion `move`):

- Man beginnt mit dem letzten Eintrag `item` in der Hauptreihe `row` des Ausdrucks (die gerade schrittweise aufgebaut wird) und dem entsprechenden Teil `part` der Gesamtsignatur `sig` seines Operators.
  Wenn `part` gleich `currpart` ist, ist man fertig.
- Andernfalls ist `part` eine Klammer. In diesem Fall bezeichnet das Attribut `choice` von `item` die Alternative innerhalb dieser Klammer, die gerade durchlaufen wird, und das Attribut `branch` des letzten Durchlaufs von `item` die zugehörige Teilreihe.
  Dann setzt man die Suche mit dem letzten Eintrag `item` dieser Teilreihe (die ebenfalls erst schrittweise aufgebaut wird) und dem entsprechenden Teil `part` der genannten Alternative fort.
  Diesen Schritt wiederholt man ggf. so lange, bis `part` gleich `currpart` ist.

Die Funktionen `right`, `up` und `down` verschieben die durch `currpart` und `curritem` definierte Stelle um einen Schritt nach rechts bzw. oben bzw. unten, sofern dies möglich ist:

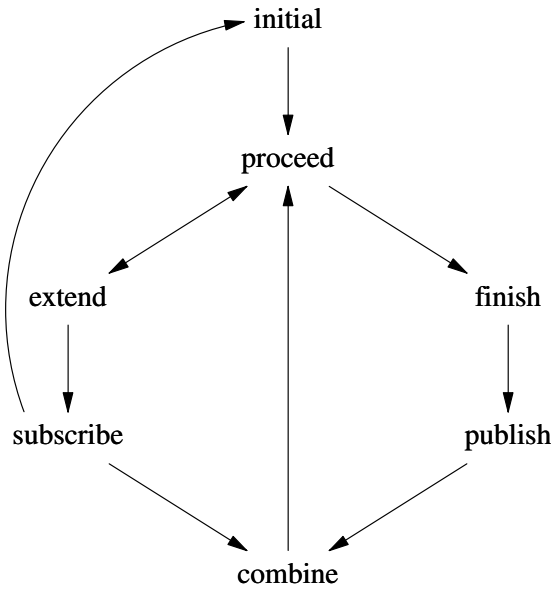
Die Funktion `right` verschiebt die durch `currpart` und `curritem` definierte Stelle um einen Schritt nach rechts, sofern dies möglich ist, das heißt: Wenn es rechts von `currpart` in der zugehörigen (Teil-)Signatur ein weiteres Teil gibt, wird `currpart` auf dieses Teil und `curritem` auf einen neuen leeren Eintrag gesetzt, der am Ende der korrespondierenden (Teil-)Reihe hinzugefügt wird. In diesem Fall ist der Resultatwert der Funktion `true`. Andernfalls ist die Funktion wirkungslos und liefert `false`.

Die Funktion `up` verschiebt die durch `currpart` und `curritem` definierte Stelle um eine Ebene nach oben, sofern dies möglich ist, das heißt: Wenn `currpart` nicht zur Gesamtsignatur `sig`, sondern zu einer Alternative einer Klammer gehört, wird `currpart` auf diese Klammer und `curritem` auf den korrespondierenden Eintrag im letzten Durchlauf durch diese Klammer gesetzt. In diesem Fall ist der Resultatwert der Funktion `true`. Andernfalls ist die Funktion wirkungslos und liefert `false`.

Die Funktion `down`, die nur aufgerufen werden darf, wenn die durch `currpart` und `curritem` definierte Stelle eine Klammer ist, verschiebt diese Stelle um eine Ebene nach unten, das heißt: `currpart` wird auf das erste Teil derjenigen Alternative innerhalb dieser Klammer gesetzt, die durch den zweiten Parameter von `down` bezeichnet wird. Zu `curritem` wird ein neuer Durchlauf hinzugefügt, dessen Attribut `choice` ebenfalls mit diesem Parameter initialisiert wird. `curritem` wird dann auf einen neuen leeren Eintrag gesetzt, der zur (momentan noch leeren) Reihe `branch` dieses Durchlaufs hinzugefügt wird.

3.2 Grobstruktur des Parsers

Das „Herzstück“ des Parsers besteht aus den sieben Funktionen `initial`, `proceed`, `extend`, `finish`, `subscribe`, `publish` und `combine`, deren Zusammenspiel in der folgenden Abbildung dargestellt ist:



Die meisten Pfeile in dieser Abbildung zeigen sowohl die Aufrufbeziehungen als auch den damit verbundenen Datenfluss zwischen den Funktionen, d. h. wie ein gerade konstruierter Ausdruck von einer Funktion zur nächsten weitergegeben wird.

Der Doppelpfeil zwischen `proceed` und `extend` zeigt an, dass ein Ausdruck von `proceed` durch einen Funktionsaufruf an `extend` weitergegeben wird und nach Beendigung von `extend` eventuell wieder zu `proceed` zurückkommt. In dieser Richtung findet jedoch kein Funktionsaufruf statt.

Beim Pfeil von `subscribe` zu `initial` wird ausnahmsweise kein Ausdruck, sondern (unter anderem) ein Operator weitergegeben.

3.3 Erzeugung initialer Ausdrücke

Die Funktion `initial` erzeugt einen initialen Ausdruck mit einem Operator `oper`, Anfangs- und Endposition `pos` sowie Importkatalog `impt` und übergibt ihn zur weiteren Verarbeitung an `proceed`.

3.4 Archive

Zu potentiell jeder Kombination einer Eingabeposition `pos` und eines Operatorkatalogs `cat` gibt es ein Archiv (Typ `Arch`), das von der Funktion `arch(pos, cat)` geliefert wird, in dem zwei Arten von Ausdrücken gespeichert werden:

- Unvollständige Ausdrücke (Attribut `cons` für „under construction“), die momentan an Position `pos` enden und deshalb als nächsten Operanden einen Ausdruck brauchen, der an dieser Position beginnt und außerdem Importkatalog `cat` besitzt.

- Vollständige Ausdrücke (Attribut `comp` für „complete“), die an Position `pos` beginnen und Importkatalog `cat` besitzen, d. h. die als nächster Operand der o. g. unvollständigen Ausdrücke in Frage kommen.

Man kann die unvollständigen Ausdrücke eines Archivs mit den Abonnenten einer Zeitschrift und die vollständigen Ausdrücke mit den Ausgaben dieser Zeitschrift vergleichen: Wenn ein neuer Abonnent registriert wird (Funktion `subscribe`), erhält er als erstes alle bisherigen Ausgaben der Zeitschrift, d. h. der neue unvollständige Ausdruck wird mit allen bereits vorhandenen vollständigen Ausdrücken kombiniert (Funktion `combine`). Außerdem wird er zur Liste der Abonnenten hinzugefügt, damit er zukünftig neue Ausgaben sofort erhält. Wenn eine neue Ausgabe erscheint (Funktion `publish`), wird sie an alle momentan gespeicherten Abonnenten verschickt, d. h. alle bereits vorhandenen unvollständigen Ausdrücke werden mit diesem neuen vollständigen Ausdruck kombiniert. Außerdem wird sie im Archiv abgelegt, damit neu hinzukommende Abonnenten sie ebenfalls erhalten.

3.5 Kombination von Ausdrücken

Die Funktion `combine` kombiniert einen unvollständigen Ausdruck `cons` (under construction) und einen vollständigen Ausdruck `comp` (complete) eines Archivs zu einem neuen Ausdruck `comb` (combined), sofern dies möglich ist, das heißt: Wenn die unten genannten Bedingungen erfüllt sind, wird `comp` als nächster Operand zu einer Kopie von `cons` hinzugefügt. Anschließend wird die so erweiterte Kopie `comb` an `proceed` weitergeleitet.

In der Formulierung der Bedingungen werden folgende Definitionen verwendet:

- Der aktuelle Parameter `par` ist der Parameter des aktuellen Signaturteils (Attribut `currpart`) von `cons`.
- Wenn sich der erste bzw. letzte Operand eines Ausdrucks ganz am Anfang bzw. ganz am Ende des Ausdrucks befindet (d. h. wenn sich davor bzw. danach keine Namen befinden), wird dieser Operand als vorderer bzw. hinterer Operand des Ausdrucks bezeichnet. (Entscheidend ist nicht, ob sich aufgrund der Signatur des Hauptoperators Namen davor bzw. danach befinden *könnten*, sondern ob sich dort tatsächlich welche befinden oder nicht.)
  Ein Operand wird als mittlerer Operand bezeichnet, wenn er weder der vordere noch der hintere Operand ist.
- Die Spitze eines Ausdrucks ist der Wurzelknoten des Operatorbaums, der diesem Ausdruck entspricht. Der Hauptoperator eines Ausdrucks ist der Operator in diesem Wurzelknoten.
- Der linke bzw. rechte Rand eines Ausdrucks besteht aus der Spitze des Ausdrucks sowie aus dem linken bzw. rechten Rand des vorderen bzw. hinteren Operanden des Ausdrucks, sofern es einen solchen Operanden gibt.

Damit die Ausdrücke `cons` und `comp` kombiniert werden können, darf keine Ausschlussangabe verletzt werden, das heißt:

- Eine Ausschlussangabe ist relevant, d. h. sie muss überprüft werden,
  a) wenn sie für den vorderen Operanden gilt (Attribut `front`) und `comp` der vordere Operand von `comb` werden würde, was genau dann der Fall ist, wenn Anfangs- und Endposition von `cons` gleich sind;
  b) wenn sie für einen mittleren Operanden gilt (Attribut `middle`) und `comp` ein mittlerer Operand von `comb` werden könnte, was genau dann der Fall ist, wenn Anfangs- und Endposition von `cons` verschieden sind und die Signatur des Hauptoperators von `cons` nach dem aktuellen Teil `currpart` noch Teile enthält (d. h. der Wert des Attributs `back` des aktuellen Teils ist höchstens `maybe`);

c) wenn sie für einen hinteren Operanden gilt (Attribut `back`) und `comp` der hintere Operand von `comb` werden könnte, was genau dann der Fall ist, wenn die Signatur des Hauptoperators von `cons` nach dem aktuellen Teil `currpart` höchstens noch optionale Teile enthält (d. h. der Wert des Attributs `back` des aktuellen Teils ist mindestens `maybe`).

Die Hilfsfunktion `relevant` ermittelt anhand dieser Kriterien, ob eine Ausschlussangabe relevant ist.

- Wenn das der Fall ist, wird aus der Ausschlussangabe mittels `expand` ein Operatorkatalog gebildet. Je nachdem, ob sich die Ausschlussangabe auf die Spitze, den linken und/oder den rechten Rand des Operanden bezieht, wird überprüft, ob einer der betreffenden Operatoren von `comp` in diesem Katalog enthalten ist. Wenn dies nicht der Fall ist, wird die Ausschlussangabe mit Sicherheit nicht verletzt.
- Andernfalls:
  a) Wenn der obige Fall a zutrifft, wird die Ausschlussangabe mit Sicherheit verletzt.
  b) Wenn der obige Fall b zutrifft und `comp` mit Sicherheit ein mittlerer Operand von `comb` werden würde, ist die Ausschlussangabe ebenfalls mit Sicherheit verletzt.
  Wenn `comp` eventuell ein mittlerer Operand von `comb` werden könnte, muss genau dies verhindert werden, damit die Ausschlussangabe nicht verletzt wird. Dies wird dadurch erreicht, dass bei der Weitergabe von `comb` an `proceed` der Hinweis `Finish` übergeben wird, der anzeigt, dass der Ausdruck nicht mehr weiter fortgesetzt, sondern nur noch an `finish` übergeben werden darf.
  c) Wenn der obige Fall c zutrifft und `comp` mit Sicherheit der rechte Operanden von `comb` werden würde, ist die Ausschlussangabe wieder mit Sicherheit verletzt.
  Wenn `comp` eventuell der rechte Operand von `comb` werden könnte, muss wiederum genau dies verhindert werden, damit die Ausschlussangabe nicht verletzt wird. Dies wird dadurch erreicht, dass bei der Weitergabe von `comb` an `proceed` der Hinweis `Extend` übergeben wird, der anzeigt, dass der Ausdruck noch weiter fortgesetzt werden muss, bevor er an `finish` übergeben werden darf.
- Wenn sich aus einer oder mehreren Ausschlussangaben ergeben würde, dass an `proceed` beide Hinweise `Extend` und `Finish` übergeben werden müssten, wäre mit Sicherheit eine der Ausschlussangaben verletzt.

Weil die Überprüfung von Ausschlussangaben relativ aufwendig sein kann, wird zuvor anhand der Folgemenge (Attribut `follow`) des aktuellen Signaturteils überprüft, ob die Kombination von `cons` und `comp` anschließend überhaupt korrekt fortgesetzt werden könnte (vgl. §3.10). Wenn dies nicht der Fall ist, bricht `combine` sofort ab.

Beispiele zur Verwendung von Ausschlussangaben

- Um zu erreichen, dass ein Ausdruck wie z. B. `1 * 2 + 3` gemäß der üblichen Punkt-vor-Strich-Regel als `(1 * 2) + 3` und nicht als `1 * (2 + 3)` interpretiert wird, kann man Anwendungen des Operators + im hinteren (und entsprechend dann auch im vorderen) Operanden von `*` ausschließen.
  Auf den ersten Blick genügt es in solchen Fällen, dass sich die Ausschlussangabe auf die Spitze (Attribut `top`) des jeweiligen Operanden bezieht. Wie ein Beispiel weiter unten jedoch zeigt, ist es grundsätzlich sinnvoll, dass sich eine Ausschlussangabe für den vorderen bzw. hinteren Operanden immer auf den gesamten rechten bzw. linken Rand (Attribut `right` bzw. `left`) des Operanden bezieht.
- Um zu erreichen, dass ein Infixoperator wie − oder / linksassoziativ ist, d. h. dass ein Ausdruck wie z. B. `1 - 2 - 3` als `(1 - 2) - 3` und nicht als `1 - (2 - 3)` interpretiert wird, kann man entsprechend Anwendungen des Operators selbst in seinem hinteren Operanden ausschließen.

Wenn ein Infixoperator rechtsassoziativ sein soll, kann man entsprechend Anwendungen des Operators selbst in seinem vorderen Operanden ausschließen.

- Um zu erreichen, dass ein Präfixoperator wie z. B. `print` schwächer bindet als arithmetische Operatoren wie z. B. `+` und `*`, d. h. dass ein Ausdruck wie z. B. `print 1 + 2` als `print (1 + 2)` und nicht als `(print 1) + 2` interpretiert wird, kann man Anwendungen von `print` im vorderen Operanden der arithmetischen Operatoren ausschließen. Anwendungen von `print` im hinteren Operanden dieser Operatoren wie z. B. `1 + print 2` können und sollten aber erlaubt sein. (Hier wird jeweils angenommen, dass `print` seinen Operanden nicht nur ausgibt, sondern auch wieder als Resultatwert liefert.)
  Wenn sich der Ausschluss von `print` im vorderen Operanden von `+` nur auf die Spitze des Operanden beziehen würde, wäre der Ausdruck `1 + print 2 + 3` immer noch mehrdeutig, weil er als `1 + print (2 + 3)` und als `(1 + print 2) + 3` interpretiert werden kann. Um zu erreichen, dass `print` auch hier wirklich schwächer bindet als `+`, muss sich der Ausschluss auf den gesamten rechten Rand des Operanden beziehen. Dann ist von den beiden genannten Interpretationen nur noch die erste möglich, weil der Teilausdruck `1 + print 2` der zweiten Interpretation eine Anwendung von `print` an seinem rechten Rand enthält.
- Aus den vorhergehenden Beispielen können folgende allgemeine Regeln abgeleitet werden:
  - Infix- und Postfix-Operatoren sollten am rechten Rand ihres vorderen Operanden alle Infix- und Präfix-Operatoren mit niedrigerem Vorrang verbieten.
  - Infix- und Präfix-Operatoren sollten am linken Rand ihres hinteren Operanden alle Infix- und Postfix-Operatoren mit niedrigerem Vorrang verbieten.
  - Links- bzw. rechtsassoziative Infix-Operatoren sollten zusätzlich sich selbst am linken Rand ihres hinteren bzw. am rechten Rand ihres vorderen Operanden verbieten.
- Wenn ein Operator optionale oder wiederholbare Signaturteile enthält, kann ein Parameter in seiner Signatur je nach konkreter Anwendung des Operators unterschiedliche Arten von Operanden bezeichnen. Beispielsweise kann ein Operand, der zu einem der Parameter `y` oder `z` in der Signatur `(x:bool) "=" (y:bool) { "=" (z:bool) }` eines variadischen Gleichheitsoperators für Wahrheitswerte gehört, entweder ein hinterer oder ein mittlerer Operand sein. Wenn es sich um einen hinteren Operanden handelt, sollen – wie beim hinteren Operanden eines normalen linksassoziativen Infix-Operators – neben dem Gleichheitsoperator selbst alle Infix- und Postfix-Operatoren ausgeschlossen sein, die schwächer als dieser Gleichheitsoperator binden (z. B. Konjunktion und Disjunktion). Wenn es sich um einen mittleren Operanden handelt, sollen zusätzlich – wie beim vorderen Operanden eines normalen Infix-Operators – auch alle schwächer bindenden Präfix-Operatoren ausgeschlossen sein (z. B. die logische Negation `~`), damit ein Ausdruck wie z. B. `a = ~b = c` eindeutig als `a = ~(b = c)` (mit zwei Anwendungen des Gleichheitsoperators mit jeweils zwei Operanden) und nicht als `a = (~b) = c` (mit einer einzigen Anwendung des Gleichheitsoperators mit drei Operanden) interpretiert wird. Dies kann für beide Parameter durch zwei unterschiedliche Ausschlussangaben erreicht werden, von denen eine für den hinteren und die andere für mittlere Operanden gilt.

3.6 Archivierung und Registrierung von Ausdrücken

Die Funktion `publish` fügt einen vollständigen Ausdruck `comp` normalerweise zum passenden Archiv hinzu, das durch die Anfangsposition (Attribut `beg`) und den Importkatalog (Attribut `impt`) des Ausdrucks bestimmt ist. Hierbei wird jeder bereits im Archiv enthaltene unvollständige Ausdruck `cons` mittels `combine` mit `comp` kombiniert.

Wenn das Archiv jedoch bereits einen ähnlichen Ausdruck `other` enthält, d. h. einen Ausdruck mit gleicher Endposition (die Anfangspositionen aller vollständigen Ausdrücke eines Archivs sind ohnehin immer gleich) und gleichem Exportkatalog, liegt eine Mehrdeutigkeit in der Eingabe vor, die im weiteren Verlauf des Parse-Vorgangs nicht mehr aufgelöst werden kann. (Wenn sich zwei Ausdrücke

noch in mindestens einem dieser Merkmale unterscheiden, kann die Mehrdeutigkeit prinzipiell noch aufgelöst werden.) Deshalb ist es in diesem Fall nicht sinnvoll, den Ausdruck `comp` zum Archiv hinzufügen und mit allen bereits enthaltenen unvollständigen Ausdrücken zu kombinieren, weil die daraus resultierenden Ausdrücke wiederum ähnlich zu bereits früher gebildeten Ausdrücken wären und sich die Mehrdeutigkeit dadurch fortpflanzen würde. Tatsächlich könnte sich der Aufwand für den Parse-Vorgang sonst durch jede solche Mehrdeutigkeit verdoppeln und damit exponentiell wachsen. Stattdessen wird der Ausdruck `comp` in diesem Fall zur Menge der ähnlichen Ausdrücke (Attribut `similar`) des Ausdrucks `other` hinzugefügt. Damit ist die Mehrdeutigkeit zwar grundsätzlich registriert, sodass sie später vom Hauptprogramm ggf. ausgegeben werden kann, bewirkt aber keinen unnötigen Zusatzaufwand des Parsers.

Es ist jedoch aus zwei Gründen notwendig und sinnvoll, den Parse-Vorgang nicht sofort mit einer Fehlermeldung abzubrechen: Erstens kann es sein, dass der Ausdruck `other`, der jetzt sozusagen stellvertretend für sich selbst und den Ausdruck `comp` (und eventuelle weitere ähnliche Ausdrücke) steht, im weiteren Verlauf des Parse-Vorgangs (z. B. aufgrund eines Typfehlers) noch ausgesondert wird und die Mehrdeutigkeit damit wieder verschwindet (ohne dass sie zuvor aufgelöst wurde). Zweitens kann der Parser im weiteren Verlauf eventuell noch weitere Mehrdeutigkeiten oder Fehler entdecken, die dann alle zusammen im Hauptprogramm ausgegeben werden können.

Die Funktion `subscribe` fügt einen unvollständigen Ausdruck `cons` zum passenden Archiv hinzu, das durch die aktuelle Endposition (Attribut `end`) des Ausdrucks und den Importkatalog seines aktuellen Parameters (gemäß Definition in §3.5), der mittels `expand` aus der Importangabe (Attribut `impt`) dieses Parameters gebildet wird, bestimmt ist. Hierbei wird `cons` mit jedem bereits im Archiv enthaltenen vollständigen Ausdruck `comp` mittels `combine` kombiniert.

Wenn das Archiv durch diesen Aufruf von `subscribe` neu entsteht, wird außerdem für jeden in diesem Katalog enthaltenen Operator mittels `initial` ein initialer Ausdruck erzeugt und „auf die Reise geschickt“.

Um die Erzeugung und Weiterverarbeitung von Ausdrücken zu vermeiden, die später sowieso nicht mit den unvollständigen Ausdrücken des Archivs kombiniert werden können, wird hierbei jedoch folgende Optimierung angewandt:

- Für jeden unvollständigen Ausdruck `cons`, der zum Archiv hinzugefügt wird, wird aus den Ausschlussangaben (Attribut `excls`) seines aktuellen Parameters ein (Gesamt-)Ausschlusskatalog gebildet, der alle Operatoren enthält, die an der Spitze des Operanden, auf den `cons` „wartet“, garantiert verboten sind. Hierfür werden die Ausschlussangaben ähnlich wie in der Funktion `combine` verarbeitet.
- Im Attribut `excl` des Archivs wird jeweils die Schnittmenge aller dieser Ausschlusskataloge gespeichert.

Initiale Ausdrücke werden dann tatsächlich nur für folgende Operatoren erzeugt:

- Wenn das Archiv neu entsteht, stimmt die Schnittmenge `excl` mit dem Ausschlusskatalog des ersten unvollständigen Ausdrucks `cons` überein.
  In diesem Fall werden initiale Ausdrücke für alle Operatoren erzeugt, die im Importkatalog des aktuellen Parameters, aber nicht in seinem Ausschlusskatalog enthalten sind.
- Wenn ein weiterer unvollständiger Ausdruck `cons` zum Archiv hinzugefügt wird, wird einerseits die Schnittmenge `excl` aktualisiert (wodurch sie eventuell kleiner wird) und andererseits die Differenzmenge zwischen alter und neuer Schnittmenge gebildet.
  In diesem Fall werden zusätzliche initiale Ausdrücke für alle Operatoren erzeugt, die in dieser Differenzmenge enthalten sind.

3.7 Erweiterung und Fertigstellung von Ausdrücken

Die Funktion `finish` schließt die Verarbeitung eines vollständigen Ausdrucks `comp` ab und fügt ihn anschließend mittels `publish` zum passenden Archiv hinzu.

Zu den abschließenden Tätigkeiten gehört:

- Die Ermittlung des Exportkatalogs des Ausdrucks (Attribut `expt` des Typs `Expr`), der mittels `expand` aus der Exportangabe seines Operators (Attribut `expt` des Typs `Oper`) gebildet wird.

- Die Ersetzung des Ausdrucks durch seine „Realisierung“, sofern sein Hauptoperator virtuell ist. Solange es noch keine virtuellen Operatoren gibt, entfällt dieser Schritt.

Die Funktion `extend` erhält als Parameter einen Ausdruck, dessen Attribute `currpart` und `curritem` das „atomare“ Signaturteil (d. h. keine Klammer) bzw. den Eintrag bezeichnen, der als nächstes mit Information gefüllt werden muss. Die Funktion versucht dann, genau das zu tun, das heißt:

Wenn dieses Signaturteil ein Name ist (Attribut `name`), wird versucht, einen zu diesem Namen passenden Text in der Eingabe zu lesen, die sich unmittelbar hinter der aktuellen Endposition des Ausdrucks befindet. Abhängig vom Wert des Attributs `reg` muss dieser Text, nach eventuellem Zwischenraum, der überlesen wird, entweder exakt mit diesem Namen übereinstimmen (Funktion `scan_exact`) oder auf den dadurch definierten regulären Ausdruck passen (Funktion `scan_match`, wobei hier ein möglichst langer passender Text verwendet wird).
Wenn dies erfolgreich ist, wird der gelesene Text im entsprechenden Eintrag gespeichert, die Endposition des Ausdrucks entsprechend weitergesetzt und Resultatwert `true` geliefert. Andernfalls ist der Resultatwert `false`.

Wenn das Signaturteil ein Parameter ist (Attribut `par`), wird der Ausdruck an die Funktion `subscribe` übergeben und Resultatwert `false` geliefert.

Der Resultatwert zeigt der aufrufenden Funktion `proceed` jeweils an, ob sie die Verarbeitung des Ausdrucks fortsetzen soll (Resultatwert `true`) oder nicht.

3.8 Fortsetzung von Ausdrücken

Die Funktion `proceed` erhält als Parameter einen Ausdruck, dessen Attribute `currpart` und `curritem` normalerweise das Signaturteil bzw. den Eintrag bezeichnen, der zuletzt mit Information gefüllt wurde. Bei einem frisch erzeugten Ausdruck sind beide Attribute nil. Die Funktion versucht dann, sämtliche möglichen Fortsetzungen dieses Ausdrucks zu konstruieren, das heißt:

Wenn `currpart` das Teil am Ende seiner (Teil-)Signatur bezeichnet, d. h. wenn die Funktion `right` Resultatwert `false` liefert:

- Wenn diese Signatur die Gesamtsignatur (des Operators) des Ausdrucks ist, d. h. wenn die Funktion `up` ebenfalls Resultatwert `false` liefert, ist der Ausdruck fertig und wird normalerweise an die Funktion `finish` übergeben.
  Ausnahme: Wenn das Bitmuster `flags` den Wert `Extend` enthält, müsste der Ausdruck zwingend weiter fortgesetzt werden, was aber offenbar nicht möglich ist.

- Andernfalls, d. h. wenn die Funktion `up` Resultatwert `true` liefert:
  Wenn sich die zuvor genannte (Teil-)Signatur in einer wiederholbaren Klammer befindet (Attribut `rep`), werden als mögliche Fortsetzungen neue Durchläufe durch die Alternativen dieser Klammer begonnen (siehe unten).

Unabhängig davon werden die hier beschriebenen Schritte anschließend mit dem neuen aktuellen Teil wiederholt.

Andernfalls hat die Funktion `right` das Signaturteil rechts vom aktuellen Teil zum neuen aktuellen Teil gemacht.

Wenn dieses neue aktuelle Signaturteil atomar, d. h. ein Name oder ein Parameter ist, wird der Ausdruck an die Funktion `extend` übergeben, die versucht, den zu diesem Teil gehörenden Eintrag mit Information zu füllen, und den Ausdruck dann ggf. an `proceed` zurückgibt.

Andernfalls ist dieses neue aktuelle Signaturteil eine Klammer.

In diesem Fall werden für jede ihrer Alternativen mögliche Fortsetzungen wie folgt konstruiert (dieselben Schritte werden auch ausgeführt, um, wie weiter oben beschrieben, einen neuen Durchlauf durch eine wiederholbare Klammer zu beginnen):

- Zu einer Kopie des Ausdrucks (Funktion `dupl`) wird mittels `down` ein (erster bzw. nächster) Durchlauf durch die Klammer hinzugefügt und `currpart` bzw. `curritem` auf das erste Teil dieser Alternative gesetzt.
- Diese Kopie des Ausdrucks wird zur weiteren Verarbeitung an einen rekursiven Aufruf von `proceed` übergeben.
  Weil `currpart` und `curritem` in diesem Fall nicht den Eintrag bezeichnen, der zuletzt mit Information gefüllt wurde, sondern der als nächstes gefüllt werden muss, wird bei diesem Aufruf der Wert `Next` im Parameter `flags` übergeben.

Wenn die Klammer optional ist (Attribut `opt`), d. h. komplett übersprungen werden kann, werden als weitere mögliche Fortsetzung außerdem alle bis jetzt beschriebenen Schritte erneut ausgeführt. Bei einer einfachen Optionsklammer, d. h. wenn die Klammer optional, aber nicht wiederholbar ist und genau eine Alternative besitzt, erhält sie in diesem Fall einen Pseudodurchlauf mit `choice` gleich `0*A` und folgenden Einträgen:

- leer, wenn der Eintrag zu einem Parameter gehört;
- ein Eintrag, der wiederum einen solchen Pseudodurchlauf enthält, wenn er ebenfalls zu einer einfachen Optionsklammer gehört;
- nil sonst, d. h. wenn der Eintrag zu einem Namen oder einer anderen Art von Klammer gehört; außerdem sind nil-Einträge am Ende des Durchlaufs unnötig.

Dieser Pseudodurchlauf ist notwendig, weil die in einer einfachen Optionsklammer definierten Parameter, anders als bei allen anderen Arten von Klammern, auch über die Klammer hinaus sichtbar sind und deshalb in der Deklaration anderer Parameter auftreten können, die nicht in dieser Klammer definiert sind (im Typ, in der Initialisierung oder in einer Import- oder Ausschlussangabe solcher Parameter). Deshalb muss die Funktion `search` zu einem in der Klammer definierten Parameter einen zugehörigen Eintrag finden.

3.9 Rekursiver und iterativer Parse-Vorgang

Die Hauptfunktion `parse` des Parsers erhält als Parameter die Menge aller vordefinierten Operatoren und ruft für jeden dieser Operatoren die Funktion `initial` auf.

Nach der bisherigen Beschreibung läuft damit der gesamte Parse-Vorgang rekursiv ab:

- Jeder Aufruf von `initial` ruft `proceed` auf.
- `proceed` ruft seinerseits entweder `extend` oder `finish` auf, die wiederum `subscribe` bzw. `publish` aufrufen.
- Beide zuletzt genannten Funktionen rufen schließlich `combine` auf, was seinerseits wieder `proceed` aufruft.
- Außerdem generiert `subscribe` u. U. rekursive Aufrufe von `initial` usw.

Nach Beendigung des gesamten Vorgangs befinden sich im „Hauptarchiv“ alle vollständigen Ausdrücke, die am Anfang der Eingabe beginnen und als Importkatalog die Menge der vordefinierten Operatoren besitzen.

Wenn das Programm korrekt und eindeutig ist, gibt es unter diesen Ausdrücken genau einen, der bis zum Ende der Eingabe geht und damit das Programm repräsentiert. Bei einem fehlerhaften Programm gibt es keinen solchen Ausdruck, während es bei einem mehrdeutigen Programm eventuell mehrere gibt. Obwohl diese Ausdrücke dann nicht vollständig ähnlich sind – sie unterscheiden sich in ihrem Exportkatalog, weil sie sonst bereits in der Funktion `publish` zu einem einzigen Ausdruck zusammengefasst worden wären –, werden sie dann auf die gleiche Weise wie dort zusammengefasst, indem alle außer dem ersten zur Menge der ähnlichen Ausdrücke (Attribut `similar`) des ersten hinzugefügt werden, um diese Mehrdeutigkeit, die ja nun nicht mehr aufgelöst werden kann, auf die gleiche Art und Weise wie dort zu registrieren.

Obwohl sich die oben beschriebene Rekursion auf natürliche Weise ergibt und prinzipiell auch korrekt funktioniert, hat sie den entscheidenden Nachteil, dass die Rekursionstiefe bei längeren Programmen extrem groß und damit irgendwann zu groß werden kann.

Deshalb sollte diese Rekursion normalerweise „aufgebrochen“ werden, um einen iterativen Algorithmus zu erhalten. In diesem Fall führt `proceed` seine „Arbeit“ nicht direkt aus, sondern speichert den übergebenen Ausdruck (und die zugehörigen `flags`) in einer Warteschlange. Eine weitere Funktion `process` (die beim rekursiven Algorithmus nicht gebraucht wird und deshalb dort einfach leer ist) arbeitet diese Warteschlange dann „verzögert“ ab, indem sie die darin enthaltenen Ausdrücke sukzessive wieder an die „eigentliche“ Funktion `proceed` übergibt.

Ein weiterer Vorteil dieses iterativen Algorithmus ist, dass er vermutlich relativ leicht parallelisiert werden kann, indem die Warteschlange gleichzeitig von mehreren Teilprozessen abgearbeitet wird.

3.10 Anfangs-, End- und Folgemengen

Eine Klammer ist überspringbar, wenn sie optional ist oder eine Alternative enthält, die ihrerseits nur aus überspringbaren Klammern besteht. Dies wird von der Funktion `skip` überprüft.

Die Anfangsmenge (oder first-Menge) einer Signatur oder Teilsignatur enthält alle Wörter, die sich direkt oder indirekt am Anfang dieser Signatur befinden, das heißt:

- Wenn das erste Teil der Signatur ein Name ist, besteht die Anfangsmenge der Signatur aus diesem Namen.

- Wenn das erste Teil der Signatur ein Parameter ist, besteht die Anfangsmenge der Signatur aus einem leeren Wort (siehe unten).
- Wenn das erste Teil der Signatur eine Klammer ist, ist die Anfangsmenge der Signatur die Vereinigung der Anfangsmengen aller Alternativen der Klammer.
  Wenn die Klammer überspringbar ist, kommt außerdem die Anfangsmenge der Signatur ohne ihr erstes Teil hinzu.

Ein leeres Wort in der Anfangsmenge einer Signatur (das als echtes Wort nie vorkommt) bedeutet, dass sich direkt oder indirekt am Anfang einer zugehörigen Reihe prinzipiell beliebige Wörter befinden können.

Die Endmenge (oder last-Menge) einer Signatur oder Teilsignatur enthält alle Parameter-Teile der Signatur, die sich direkt oder indirekt am Ende dieser Signatur befinden, das heißt:

- Wenn das letzte Teil der Signatur ein Name ist, ist die Endmenge der Signatur leer.
- Wenn das letzte Teil der Signatur ein Parameter ist, besteht die Endmenge der Signatur aus diesem Teil.
- Wenn das letzte Teil der Signatur eine Klammer ist, ist die Endmenge der Signatur die Vereinigung der Endmengen aller Alternativen der Klammer.
  Wenn die Klammer überspringbar ist, kommt außerdem die Endmenge der Signatur ohne ihr letztes Teil hinzu.

Die Folgemenge (oder follow-Menge) eines Parameter-Teils einer Signatur oder Teilsignatur enthält alle Wörter, die in einer Anwendung des Operators, zu dessen Signatur dieses Teil direkt oder indirekt gehört, nach einem Operanden kommen können, der zu diesem Parameter gehört.

Das heißt: Nur wenn nach einem solchen Operanden eines dieser Wörter in der Eingabe kommt, ist es sinnvoll und notwendig, den Operanden in die Operatoranwendung einzusetzen; wenn nicht, kann der Aufwand zur Überprüfung seiner Ausschlussangaben in der Funktion `combine` entfallen.

Ausnahme: Wenn die Folgemenge ein leeres Wort enthält, können nach einem solchen Operanden prinzipiell beliebige Wörter kommen (z. B. weil ein weiterer Operand kommen kann). In diesem Fall muss die Einsetzung des Operanden immer versucht werden.

Die Funktion `setfollow` setzt für alle Parameter-Teile einer Signatur das Attribut `follow`, das die Folgemenge des Teils enthält. Wenn die Folgemenge ein leeres Wort enthält, befindet es sich garantiert am Anfang der Sequenz `follow`.

3.11 Weitere Funktionen

Die Funktion `dupl` erzeugt ein Duplikat eines Ausdrucks, bei dem alle direkten und indirekten Einträge dupliziert sind, die sich bei der Fortsetzung des Ausdrucks noch verändern können.

Die Funktion `search` liefert den Eintrag eines Ausdrucks, der zu einem bestimmten Parameter gehört.

Die Funktion `setback` setzt für alle Teile einer Signatur das Attribut `back` auf den passenden Wert `true`, `maybe` oder `false`, je nachdem ob sich dieses Teil garantiert, eventuell oder garantiert nicht ganz am Ende eines zugehörigen Ausdrucks befindet.

Die Funktion `front` bzw. `back` liefert den vorderen bzw. hinteren Operanden eines Ausdrucks, sofern es einen solchen Operanden gibt, oder andernfalls nil.

Die Funktion `expand` expandiert eine Import-, Export- oder Ausschlussangabe `spec`, indem sie i. w. den Exportkatalog dieses Ausdrucks liefert. Die folgenden Operatoren werden jedoch besonders interpretiert:

- Der Pseudooperator `All` wird durch alle Operatoren der übergebenen „Grundmenge“ `all` ersetzt.
- Der Pseudooperator `Self` wird durch alle Operatoren des Exportkatalogs des ebenfalls übergebenen Ausdrucks `expr` ersetzt.
- Ein Parameter des Operators dieses Ausdrucks `expr` wird durch alle Operatoren des Exportkatalogs des korrespondierenden Operanden von `expr` ersetzt.
- Der Pseudooperator `AllBut` bewirkt, dass das Komplement bezüglich der „Grundmenge“ `all` gebildet wird.

3.12 Visualisierung des Parse-Vorgangs

Die Visualisierungskomponente, die im wesentlichen von Lukas Pietzschmann im Rahmen seiner Projektarbeit entwickelt wurde, erweitert einige offene Funktionen des Parsers, um wesentliche Ereignisse während des Parse-Vorgangs zu protokollieren, beispielsweise die Erstellung eines neuen Archivs (Funktion `arch`), das Hinzufügen eines vollständigen oder unvollständigen Ausdrucks zu einem Archiv (Funktion `publish` bzw. `subscribe`) oder das Kombinieren eines unvollständigen und eines vollständigen Ausdrucks (Funktion `combine`).

Nach Beendigung des Parse-Vorgangs kann dieser dann anhand der gesammelten Daten schrittweise nachvollzogen werden, indem bei jedem Schritt die momentan vorhandenen Archive mit ihren vollständigen und unvollständigen Ausdrücken sowie jeweils der aktuelle Inhalt der Warteschlange auf einfache Weise (mit Hilfe der Bibliothek `ncurses`) graphisch in einem Terminalfenster dargestellt werden.

Zu Beginn der Visualisierung wird ganz oben der Inhalt der Quelldatei und ganz unten eine kurze Erklärung der wichtigsten Tastendrücke angezeigt. Der Inhalt der Warteschlange wird anschließend am rechten Rand dargestellt, der übrige Platz wird zur Anzeige von Archiven verwendet.

Jedes Archiv zeigt auf der linken Seite die unvollständigen Ausdrücke an, die an der Eingabeposition, zu der das Archiv gehört, enden und auf der rechten Seite die vollständigen Ausdrücke, die an dieser Position beginnen. Die Trennlinie zwischen diesen beiden Seiten befindet sich vertikal genau unterhalb der entsprechenden Position im Quelltext.

Wenn die Funktionen `subscribe` und `publish` einen Ausdruck auf der entsprechenden Seite eines Archivs hinzugefügt haben und dann diesen neuen Ausdruck mit jedem vorhandenen Ausdruck auf der anderen Seite kombinieren, werden die beiden Ausdrücke, die hierfür an die Funktion `combine` übergeben werden, jeweils in einem Schritt farbig hervorgehoben und im nächsten Schritt wieder normal dargestellt.

Damit derartige Farbausgaben korrekt funktionieren, muss das verwendete Terminalfenster grundsätzlich farbige Ausgaben unterstützen, und die Umgebungsvariable `TERM` muss einen passenden Wert enthalten, z. B. `xterm-256color` anstelle von `xterm`.

Aufgrund des begrenzten Platzes in einem typischen Terminalfenster, sollte die Quelldatei nur einen relativ kurzen einzeiligen Ausdruck enthalten. Die wesentliche Aspekte des Parse-Vorgangs können erfahrungsgemäß anhand solcher kurzen Eingaben nachvollzogen werden.

Die wichtigsten Tastendrücke sind:

- `n` (next) bzw. `p` (previous) geht einen Schritt vorwärts bzw. rückwärts.
  Mit einer vorangestellten Zahl werden (wie im Texteditor Vim) entsprechend viele Schritte auf einmal gegangen.
  Wenn es keinen nächsten bzw. vorigen Schritt mehr gibt, wird ein optisches und akustisches Signal ausgegeben.
- `h` (help) zeigt eine vollständige Liste aller möglichen Tastendrücke an, die durch erneutes Drücken von `h` wieder verschwindet.
- `o` (operators) zeigt eine Liste aller bis jetzt aufgetretenen Operatoren jeweils mit einer laufenden Nummer an, die durch erneutes Drücken von `o` wieder verschwindet.
  Diese Operatornummern werden bei der Darstellung von Ausdrücken in der Warteschlange und in den Archiven verwendet, um den Hauptoperator eines Ausdrucks (der aus Platzgründen u. U. verkürzt angezeigt werden muss) jeweils eindeutig zu identifizieren.
- Wenn der vertikal zur Verfügung stehende Platz zur Darstellung der Archive oder der Operatorliste nicht ausreicht, wird jeweils ein einfacher „Rollbalken“ angezeigt, und die Anzeige kann mit den entsprechenden Pfeiltasten aufwärts bzw. abwärts gerollt werden.
  Analog kann die Anzeige der Warteschlange bei Bedarf mit den Tasten `w` und `s` aufwärts bzw. abwärts gerollt werden. (Die Buchstaben `w` und `s` haben keine besondere Bedeutung, außer dass sie auf der Tastatur untereinander angeordnet sind.)
  Eine horizontale Rollmöglichkeit wird nicht angeboten.

4 Auswertung von Ausdrücken (Laufzeitsystem)

4.1 Auswertungsfunktionen

Das Laufzeitsystem besteht „nach außen“ im wesentlichen aus einer einzigen Funktion `eval`, die als Parameter einen Ausdruck erhält, diesen auswertet (bzw. ausführt) und seinen Wert als Resultat liefert.

Intern gibt es sinnvollerweise jedoch für jeden vordefinierten Operator von MOSTflexiPL eine eigene Auswertungsfunktion wie z. B. `add_eval`, `print_eval` etc., die im Attribut `eval` dieses Operators gespeichert wird und die für die Auswertung von Anwendungen dieses einen Operators verantwortlich ist.

Falls das Verhalten mehrerer Operatoren sehr ähnlich ist, kann es im Einzelfall aber auch sinnvoll sein, dass diese Operatoren die gleiche Auswertungsfunktion besitzen.

Darüber hinaus gibt es einige wenige allgemeine Auswertungsfunktionen wie z. B. `const_eval` und `oper_eval`, die für die Auswertung benutzerdefinierter Konstanten und Operatoren zuständig sind.

Die o. g. Hauptfunktion `eval` delegiert die Arbeit dann einfach an die Auswertungsfunktion des Hauptoperators des übergebenen Ausdrucks, die ihrerseits typischerweise rekursive Aufrufe von `eval` für die Operanden des Ausdrucks ausführt und deren Ergebnisse geeignet verarbeitet.

Achtung: In C++ ist für Ausdrücke wie z. B. `expr1 + expr2` oder `f(expr1, expr2)` nicht festgelegt, in welcher Reihenfolge die Teilausdrücke `expr1` und `expr2` ausgewertet werden. Um sicherzustellen, dass die Operanden eines MOSTflexiPL-Ausdrucks garantiert von links nach rechts ausgewertet werden, müssen solche C++-Ausdrücke (wenn sie z. B. rekursive Aufrufe von `eval` enthalten) ggf. „auseinandergenommen“ werden, zum Beispiel:

```
auto x1 = expr1;
auto x2 = expr2;
x1 + x2 /* bzw. */ f(x1, x2)
```

Zur Auswertung eines Hauptausdrucks, d. h. eines gesamten Programms, muss die Funktion `exec` anstelle von `eval` verwendet werden, weil sie vor dem Aufruf von `eval` notwendige globale Initialisierungen ausführt, beispielsweise die Erzeugung des globalen Kontexts (vgl. §4.4) sowie die Definition globaler Konstanten in diesem Kontext.

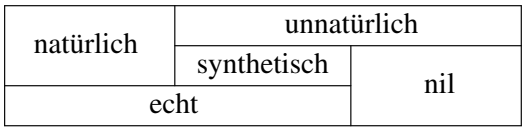
4.2 Laufzeitwerte

Der offene Typ `Value` kann MOSTflexiPL-Laufzeitwerte unterschiedlicher Art repräsentieren, beispielsweise `bool`-, `int`- oder `float`-Werte (Attribute `intval` etc.).

Der nil-Wert von `Value` repräsentiert entsprechend einen MOSTflexiPL-nil-Wert, der u. a. beim Abfragen einer Variablen entsteht, der noch kein Wert zugewiesen wurde. Außerdem liefern undefinierte arithmetische Operationen, insbesondere eine ganzzahlige Division durch 0, als Resultat nil.

Ein `Value`-Objekt, dessen Attribut `synth` den Wert `true` besitzt, repräsentiert einen synthetischen Wert, der u. a. durch Konstantendeklarationen ohne explizite Initialisierung wie z. B. `Person : type`, `p : Person` oder auch `i : int` entstehen kann. `Person` ist damit eine eindeutige Konstante des Metatyps `type` und damit ein neuer Typ. `p` ist eine eindeutige Konstante des Typs `Person` und damit logisch ein Objekt dieses Typs, vergleichbar mit Objekten offener Typen, die mit `uniq` initialisiert wurden. `i` ist vollkommen analog eine eindeutige Konstante des Typs `int`, d. h. `i` ist verschieden von allen anderen natürlichen oder synthetischen Werten des Typs `int`. Anders als bei Typen wie `Person`, die neben dem nil-Wert nur solche synthetischen Werte besitzen, sind synthetische Werte für Typen wie `int` eher ungewöhnlich und selten, aber prinzipiell möglich.

Grundsätzlich können folgende Arten von MOSTflexiPL-Werten unterschieden werden:



4.3 Operationen mit unnatürlichen Werten

Wenn mindestens ein Operand einer arithmetischen Operation synthetisch oder nil ist, ist das Ergebnis der Operation nil.

Für zwei Werte `x` und `y` desselben MOSTflexiPL-Typs gilt die Gleichheitsrelation `x = y` genau dann,

- wenn beide den gleichen natürlichen Wert darstellen
- oder wenn beide nil sind
- oder wenn beide denselben synthetischen Wert darstellen.

Für zwei Werte `x` und `y` desselben numerischen MOSTflexiPL-Typs gilt die Kleinerrelation `x < y` genau dann, wenn beides natürliche Werte sind, für die die Kleinerrelation gilt. Das bedeutet umgekehrt, dass `x < y` immer `false` ist, wenn mindestens einer der Werte unnatürlich ist.

Die übrigen Relationen werden wie folgt auf die Gleichheits- und Kleinerrelation zurückgeführt:

- `x =/ y` gilt genau dann, wenn `x = y` nicht gilt.
- `x > y` gilt genau dann, wenn `y < x` gilt.
- `x <= y` gilt genau dann, wenn `x < y` oder `x = y` gilt.
- `x >= y` gilt genau dann, wenn `y <= x` gilt.

Durch diese Definitionen ist es möglich, dass für zwei Werte `x` und `y` die Vergleiche `x < y`, `x = y` und `x > y` alle drei nicht gelten – woraus u. a. zum Beispiel auch folgt, dass `x >= y` nicht immer gleichbedeutend mit der Negation von `x < y` ist.

4.4 Kontexte

Ein Kontext (Typ `Context`) speichert in einer Tabelle (Attribut `tab`) die Werte aller Konstanten eines bestimmten Gültigkeitsbereichs. Dazu zählen auch die Parameter einer Operatoranwendung sowie Variablen, bei denen es sich technisch um Konstanten handelt, deren Wert die Adresse einer Speicherzelle ist, in der sich der aktuelle Wert der Variablen befindet.

Alle Kontexte außer dem globalen Kontext besitzen außerdem einen Verweis auf einen statisch umschließenden Kontext (Attribut `encl`).

Der globale Kontext, der zu Beginn der Programmausführung von der Funktion `exec` erzeugt und in der globalen (bzw. Thread-lokalen) Variablen `curr` gespeichert wird, speichert die Werte aller globalen Konstanten.

Wenn die Funktion `oper_eval` eine Anwendung eines benutzerdefinierten Operators auswertet, erzeugt sie hierfür einen neuen Kontext, der für die Dauer der Auswertung zum aktuellen Kontext `curr` wird. In ihm werden die Werte der Parameter und lokalen Konstanten des Operators gespeichert. Sein umschließender Kontext ist im allgemeinen nicht der aktuelle Kontext, sondern der Kontext, in dem der Operator definiert wurde.