universität
wien

# DIPLOMARBEIT

**Titel der Diplomarbeit**

## Das automatische Verarbeiten mathematischer Formeln

**angestrebter akademischer Grad**

**Magister der Naturwissenschaften (Mag. rer. nat.)**


| | |
|---|---|
| Verfasser: | David Langer |
| Matrikel-Nummer: | 0507344 |
| Studienkennzahl: | 405 |
| Studienrichtung: | Mathematik |
| Betreuer: | Ao. Univ.-Prof. Dipl.-Ing. Dr. Hermann Schichl |


Wien, am 11. April 2013

# Vorwort

In der vorliegenden Arbeit wird die Funktionsweise eines Parsers beschrieben, der mathematische Formeln einlesen und in bestimmten Darstellungsformaten ausgeben kann. Die Formeln liegen in dem weit verbreiteten Format LaTeX vor und werden zunächst in (binäre) Syntaxbäume verwandelt. Diese Bäume müssen dann jedoch noch nachbearbeitet und semantisch analysiert werden, damit die Formeln in anderen Formaten ausgegeben werden können. Als zwei Beispiele für solche Ausgabeformate werden OpenMath und Concise behandelt. Der Schwerpunkt der Arbeit liegt darin, konkrete Verfahren zur Umwandlung und Verarbeitung von Formeln und Syntaxbäumen zu beschreiben, und nicht, die Vorgehensweisen zu formalisieren. Da in der Praxis die meisten Formeln relativ kurz sind (Formeln, die länger als eine Zeile sind, werden von den allermeisten Menschen als schwer lesbar empfunden), fallen Aspekte wie Speicherplatz, Aufwand und Laufzeit kaum ins Gewicht. Sie werden hier deshalb auch nicht analysiert.

Mein besonderer Dank gilt meinem Betreuer, Hermann Schichl, Arnold Neumaier und Kevin Kofler für ihre Hinweise zu Concise, dem MathBridge-Team sowie meiner Familie und all meinen Freunden und Freundinnen, die mich während dem Verfassen dieser Arbeit in irgendeiner Form unterstützt haben.

Wien, im März 2013 David Langer

# Inhalt


**1 Einleitung** **1**

1.1 Ziel, Voraussetzungen . . . . . 1
1.2 Verwendete Datentypen und -strukturen . . . . . 3
1.2.1 Elementare Datentypen . . . . . 3
1.2.2 Strings . . . . . 3
1.2.3 Vektoren . . . . . 4
1.2.4 Bäume . . . . . 4
1.2.5 Symboltabellen . . . . . 5
1.3 Beschreibung von Algorithmen . . . . . 5
1.4 Arten von Sprachen . . . . . 6
1.4.1 Formale Sprachen . . . . . 6
1.4.2 Einteilung der Sprachen . . . . . 6
1.5 Ebenen von Grammatik . . . . . 8
1.6 Darstellung von Mathematik und Formeln . . . . . 9
1.6.1 Darstellung von Formeln . . . . . 9
1.6.2 Mathematik und Formeln . . . . . 11
1.7 Der Shunting-Yard-Algorithmus von Dijkstra . . . . . 13
1.7.1 Einschränkungen des Shunting-Yard-Algorithmus . . . . . 15

**2 LaTeX** **19**

2.1 Zerlegung eines Strings in Tokens . . . . . 20
2.1.1 Zeichen . . . . . 20
2.1.2 Einfache Zerlegung . . . . . 20
2.1.3 Sonderfälle . . . . . 21
2.2 Besondere Tokens . . . . . 23
2.2.1 Kategorisierung der Tokens . . . . . 23
2.2.2 Klammern und Gruppen . . . . . 24
2.2.3 Befehle . . . . . 24
2.2.4 TeX-Befehle . . . . . 24
2.2.5 Wurzeln . . . . . 25
2.2.6 Hoch- und tiefgestellte Zeichen . . . . . 25
2.2.7 Transformation . . . . . 25

2.3 Neue Schreibweise für Befehle . . . 26
2.4 Zusammenfassung . . . 27
2.4.1 Beispiele . . . 28

**3 Syntax** **31**
3.1 Syntaktische Funktionen . . . 31
3.1.1 Klammern . . . 32
3.1.2 Operatoren . . . 33
3.1.3 Namen . . . 34
3.1.4 Befehle und geschweifte Klammern . . . 34
3.2 Formeln als formale Sprache . . . 34
3.3 Shunting-Yard-Algorithmus . . . 35
3.3.1 Benötigte Zusatzinformationen . . . 39
3.4 Syntaxbäume . . . 39
3.5 Beispiele . . . 40

**4 Semantik** **49**
4.1 Semantische Typen und Kategorien . . . 49
4.1.1 Semantische Regeln . . . 49
4.1.2 Typbestimmung . . . 50
4.1.3 Beispiele . . . 51
4.2 Nachbearbeitung von Syntax-Bäumen . . . 55
4.2.1 Falsche Assoziativität . . . 55
4.2.2 Zahlen . . . 57
4.2.3 Operatoren aus mehreren Tokens . . . 62
4.2.4 Operatoren, die nicht erkannt werden . . . 65

**5 Math-Bridge** **73**
5.1 Hintergrund . . . 73
5.2 OpenMath . . . 73
5.2.1 OpenMath-Objekte . . . 74
5.2.2 Ganze Formeln . . . 75
5.2.3 Beispiele . . . 75
5.2.4 Umwandeln von Formeln in OpenMath . . . 79
5.2.5 Fazit . . . 84

**6 Concise** **87**
6.1 Die Darstellung von Formeln in Concise . . . 87
6.1.1 Variable und Konstante . . . 87
6.1.2 Operatoren und Funktionen mit einer festen Anzahl von Argumenten . . . 88
6.1.3 Listen . . . 89

6.1.4 Relationen . . . . . 89
6.1.5 Summen . . . . . 90
6.1.6 Mehrfaches Vorkommen von Ausdrücken . . . . . 92
6.2 Die Umwandlung von Semantik-Bäumen in Concise-Ausdrücke . . . . . 92
6.2.1 Grundsätzliche Überlegungen . . . . . 92
6.2.2 Die Algorithmen zur Umwandlung . . . . . 94
6.3 Record Sheets . . . . . 99

**Conclusio** **103**

**Anhang: Eine Implementation in C++** **105**

**Literaturverzeichnis** **179**

# 1 Einleitung

## 1.1 Ziel, Voraussetzungen

Computer sind heutzutage ein unerlässliches Hilfsmittel zur Verarbeitung mathematischer Ausdrücke. Ursprünglich wurden sie zur Automatisierung numerischer Berechnungen eingesetzt. Um den Wert von $\sqrt{3}$ zu bestimmen, kann in einem Computerprogramm, das in der Programmiersprache C geschrieben ist, der Funktionsaufruf `sqrt(3.0)` verwendet werden. Das liefert dann als Ergebnis den Wert `1.732051`. Mit Hilfe von Computeralgebrasystemen ist es weiters auch möglich, symbolische Berechnungen durchzuführen. Um das unbestimmte Integral $\int \frac{x}{x^3+1}\,\mathrm{d}x$ zu bestimmen, muss in dem Computeralgebrasystem Maxima die Zeile

```
integrate(x/(x^3 + 1), x);
```

eingegeben werden. Die Ausgabe des Ergebnisses $\frac{\log(x^2-x+1)}{6} + \frac{\arctan\frac{2x-1}{\sqrt{3}}}{\sqrt{3}} - \frac{\log(x+1)}{3}$ sieht dann so aus:

```
                      2 x - 1
     2            atan(-------)
log(x  - x + 1)        sqrt(3)    log(x + 1)
--------------- + ------------- - ----------
      6              sqrt(3)           3
```

Es gibt noch zahllose weitere Beispiele für die Verwendung von Computern zum Auswerten mathematischer Formeln. Sie haben alle gemeinsam, dass die Formeln irgendwie in einer für den Computer verwendbaren Art dargestellt werden müssen.

In dieser Arbeit geht es darum, wie Formeln, so wie sie von und für Menschen geschrieben sind, für Computer übersetzt werden können. Die Aufgabenstellung ist recht alt, denn schon sehr früh wurden Programmiersprachen so entworfen, dass die Art, wie Formeln geschrieben werden müssen, für Menschen möglichst gut lesbar ist.[1] Wir wollen uns hier aber nicht nur mit Schreibweisen beschäftigen, die „richtigen“ Formeln nur ähnlich sind, sondern versuchen, Formeln, so wie sie in Büchern stehen, für den Computer zu übersetzen. Damit sind Formeln in Standard-Schreibweise gemeint, so wie Naturwissenschaftler/-innen sie normalerweise schreiben. Spezielle Schreibweise wie die Infix-Notation, die manchmal in der Logik verwendet wird, werden hier nicht behandelt. Sie sind allerdings meist absichtlich so gebaut, dass sie ohnehin von Maschinen gut verarbeitet werden können.

[1] Die älteste Referenz zu dem Thema ist [Samelson, Bauer 1960], in [Dijkstra 1961] wird ein entsprechender Algorithmus für ALGOL-Compiler beschrieben und [Pratt 1973] beschäftigt sich noch genauer mit der Rangordnung von Operatoren.

Diese Aufgabenstellung ist aber noch zu allgemein. Zunächst einmal müssen wir uns fragen, wie die Formeln in einem Buch dargestellt werden. Wenn zum Druck des Buches die Seitenbeschreibungssprache PostScript zum Einsatz kommt, wird die einfache Formel $x^2$ so oder so ähnlich dargestellt:

```
/Times-Roman findfont
12 scalefont
setfont
50 80 moveto
(x) show
/Times-Roman findfont
9 scalefont
setfont
57 87 moveto
(2) show
```

Das ist für unsere Zwecke zu kompliziert. Wir beschäftigen uns nur mit Formeln, die in den Formatierungssprachen TEX oder LATEX geschrieben sind. (Aus ihnen kann problemlos eine druckbare Postscript-Ausgabe erzeugt werden, wofür es schon viele Computerprogramme gibt.) Sie haben den großen Vorteil, dass sie sowohl für Menschen als auch für Maschinen gut lesbar sind.

Es gibt jedoch noch ein weiteres Problem, das beim Betrachten folgender Formel deutlich wird:

$$|a|b|c|$$

Hier ist nicht klar, was diese Formel überhaupt heißen soll. Zwei mögliche Interpretationen werden gleich offensichtlich: $|a| \cdot b \cdot |c|$ und $\left|a \cdot |b| \cdot c\right|$. Wir beschäftigen uns hier nur mit Formeln, die für Menschen *eindeutig lesbar* sind. Insbesondere kümmern wir uns nicht um falsche Formeln: mit Ausdrücken wie $x+(y-$ (hier fehlen zumindest eine runde Klammer und ein Operand) beschäftigen wir uns ebensowenig wie mit Formeln, die zwar richtig aussehen, aber das Falsche bedeuten. Wenn eine Formel etwa „$a$ plus $b$“ heißen soll, aber dafür $a*b$ geschrieben ist, dann wird sie so interprertiert, wie sie dasteht, nämlich als „$a$ mal $b$“.

Eine letzte Einschränkung gibt es noch: Aus Rücksicht auf die Länge dieser Arbeit wird die Behandlung von Arrays und Matrizen weggelassen. Diese sind nämlich aus mehreren einfachen Formeln zusammengesetzt, und ihre Verarbeitung ist daher weit komplizierter.

Das Ergebnis ist dann ein Formelparser, der (korrekte) LATEX-Formeln lesen und semantisch analysieren kann. Die Formeln können dann (möglicherweise unter Zuhilfenahme von Zusatzprogrammen) in ein beliebiges Ausgabeformat umgewandelt werden. Während das Einlesen der Formeln immer nach denselben Regeln erfolgt, ist die Erzeugung der Ausgabe natürlich für jedes Format anders. Wir werden uns deshalb zum Schluss zwei Beispiele dafür ansehen, nämlich OpenMath und Concise.

Ich behaupte nicht, dass die Vorgehensweise, die hier beschrieben wird, *die* Lösung des Problems ist. Es gibt sicher noch andere Möglichkeiten, mit denen auch sinnvolle Ergebnisse erzielt werden können. Die Algorithmen und Konzepte, die in dieser Arbeit vorgestellt werden, sind mein Lösungsansatz, der mir nach eingehender Überlegung sinnvoll erscheint und mit dem ich bis jetzt jede Formel (die den oben angeführten Einschränkungen unterliegt) richtig verarbeiten konnte. Damit ist das Ganze zumindest *eine* Lösung.

## 1.2 Verwendete Datentypen und -strukturen

Hier sind zum besseren Verständnis alle Datentypen und Datenstrukturten aufgelistet, die im Weiteren verwendet werden.

### 1.2.1 Elementare Datentypen

#### Zahlen

Mit Zahlen sind hier immer ganze Zahlen gemeint. Wir machen uns dabei jedoch keine Gedanken darüber, ob und wie sie im Computer dargestellt werden können, d. h. die Frage, ob die Zahl $2^{65}$ überhaupt auf einem Computer mit einem 64-Bit-Prozessor dargestellt werden kann, betrifft uns nicht. (Wir werden es auch nicht annähernd mit so großen Zahl zu tun haben, wo solche Probleme auftreten könnten.)

#### Boolesche Werte

Sie können die Werte 1 (wahr) und 0 (falsch) annehmen. Für Boolesche Werte verwenden wir die Operationen $\wedge$ (und), $\vee$ (oder) sowie $\neg$ (Negation).

#### Zeichen

Wir verwenden Zeichen nur als Elemente eines Alphabets. Zeichen werden immer in einfachen Anführungszeichen geschrieben, z. B. $'\mathtt{x}'$. Für Arithmetik (wie etwa in der Programmiersprache C) können sie nicht verwendet werden, d. h. $'\mathtt{a}' + '\mathtt{b}'$ ist hier kein gültiger Ausdruck.

### 1.2.2 Strings

Strings sind Listen von Zeichen, Schreibweise: $''\mathtt{abc}''$. Ist $s$ ein String, dann ist $\mathtt{size}(s)$ die Länge von $s$, und $s_i$ ist das $i$-te Element von $s$ (die Nummerierung beginnt bei 0). Die einzige Operation für Strings ist die Konkatenation: Sind $a = ''\mathtt{aaa}''$ und $b = ''\mathtt{bbb}''$ zwei Strings, dann ist $ab$ der String $''\mathtt{aaabbb}''$.

### 1.2.3 Vektoren

Vektoren sind, wie in der Informatik üblich, Listen von Elementen beliebigen Typs. Ist $v$ ein Vektor, dann bezeichnet $v_i$ das $i$-te Element von $v$. Weiters gibt es folgende Funktionen für Vektoren:

`size`$(v)$ die Länge von $v$

`push`$(v,x)$ hängt das Element $x$ an den Vektor $v$ an

`pop`$(v)$ entfernt das letzte Element von $v$

`back`$(v)$ liefert das letzte Element von $v$

### 1.2.4 Bäume

Wir verwenden zur Darstellung von Formeln Binärbäume.[2] Jeder Binärbaum besteht aus einem Knoten sowie einem Zeiger auf den linken und rechten Unterbaum. Wenn ein Unterbaum fehlt, wird der entsprechende Zeiger auf `NULL` gesetzt. Im Knoten müssen folgende Informationen enthalten sein: ein Vektor von Strings für die Formelzeichen und ein Vektor von Strings für den Typ des Baumes (bzw. die Typen, falls er nicht eindeutig ist; wir gehen im Weiteren der Einfachheit halber jedoch immer davon aus, dass die Typen eindeutig sind). Weiters gibt es folgende Funktionen für Bäume:

`left`$(t)$ Zeiger auf den linken Unterbaum von $t$

`right`$(t)$ Zeiger auf den rechten Unterbaum von $t$

`getName`$(t,i)$ das $i$-te Formelzeichen des Knotens

`getType`$(t,i)$ der $i$-te Typ des Knotens

`setName`$(t,n,i)$ setzt das $i$-te Formelzeichen des Knotens auf $n$

`setType`$(t,n,i)$ setzt den $i$-ten Typ des Knotens auf $n$

`getType`$(n,i)$ der $i$-te Typ des Knotens

Da wir immer davon ausgehen, dass Typen eindeutig sind, verwenden wir auch die Abkürzungen `setType`$(t,n)$ statt `setType`$(t,n,0)$ und `getType`$(n)$ statt `getType`$(n,0)$.

In den Bäumen, die wir verwenden, sind zwar immer die Unterbäume abgespeichert, nicht jedoch der Eltern-Baum. Deshalb definieren wir auch dafür eine Funktion: Ist $N$ ein Baum und $n$ ein Knoten, der irgendwo in dem Baum vorkommt, dann ist `parent`$(N,n)$ der Eltern-Knoten von $n$. Wenn es keinen Knoten in $N$ gibt, der $n$ als Unterbaum hat, dann sagen wir, der Eltern-Knoten ist `NULL`.

[2] Man könnte auch Bäume verwenden, die mehr als nur zwei Unterbäume haben. Es hat sich jedoch in der Sprachwissenschaft gezeigt, dass jeder Satz einer menschlichen Sprache als binärer Syntaxbaum dargestellt werden kann. Der Grund dafür scheint zu sein, dass das menschliche Gehirn ähnlich wie Computer mit einem Binärsystem funktioniert. Da mathematische Formeln sehr eng mit menschlichen Sprachen verwandt sind, eignen sich Binärbäume auch für ihre Darstellung. Sie haben auch den Vorteil, dass sie eine minimale Anzahl von Unterbäumen enthalten. Außerdem ist immer von vornherein klar, wie viele Unterbäume jeder Baum höchstens haben kann.

### 1.2.5 Symboltabellen

Symboltabellen werden gebraucht, um Zusatzinformationen zu bestimmten Bezeichnern von Objekten (den „Symbolen“) zu speichern. Von Compilern oder Interpretern werden sie benutzt, um jedem Symbol im Quellcode einen Datentyp und eine Adresse zuzuordnen. Hier ist ein vereinfachtes Beispiel einer Tabelle, die die Namen von Variablen sowie ihre Typen und Werte enthält:

| Variable | Typ | Wert |
|---|---|---|
| $a$ | int | 1 |
| $y$ | double | 5.3 |
| $x$ | char | `'a'` |

In dieser Arbeit treten zahlreiche Algorithmen auf, bei denen bestimmte Zusatzeigenschaften von Symbolen benötigt werden. Beim Parsen der Formel $a+b$ muss beispielsweise bekannt sein, dass das Zeichen $+$ ein binärer Operator ist. Diese Informationen werden in Symboltabellen abgespeichert, auf die dann die Funktionen zur Typbestimmung zugreifen. Deshalb wird diese wichtige Datenstruktur hier noch einmal ausdrücklich erwähnt, obwohl sie selbst in der Beschreibung der Algorithmen nicht wirklich vorkommt. Ausführlichere Informationen zu dem Thema gibt es in Lehrbüchern über Algorithmen und Datenstrukturen oder Compilerbau, etwa bei [Sedgewick 2002] oder [Aho, Liam, Sethi 2007].

## 1.3 Beschreibung von Algorithmen

Die Beschreibung von Algorithmen erfolgt in dieser Arbeit weder in einer bestimmten Programmiersprache noch in Pseudo-Code. Stattdessen werden die einzelnen Schritte der Algorithmen in sehr einfachen, maschinennahen Anweisungen dargestellt. Dadurch kommen keine sprachspezifischen Elemente wie Schleifen o. Ä. vor, die vom eigentlichen Algorithmus ablenken könnten, und die Implementierung in einer beliebigen Programmiersprache wird so erleichtert. Das Zeichen $=$ wird dabei nur als Vergleichsoperator verwendet. Für Variablenzuweisung wird immer $\leftarrow$ gebraucht. Hier sei als Beispiel der Euklidische Algorithmus beschrieben:

**Algorithmus 1.3.1** Euklidischer Algorithmus

Eingabe: der größte gemeinsame Teiler der Zahlen $a$ und $b$ soll bestimmt werden.

Ausgabe: die Variable $a$ enthält den größten gemeinsamen Teiler.

**0** Falls $a = 0$, setze $a \leftarrow b$ und der Algorithmus ist fertig.

**1** Falls $a > b$, setze $a \leftarrow a - b$, sonst $b \leftarrow b - a$.

**2** Wiederhole Schritt 1, solange $b \neq 0$ ist.

## 1.4 Arten von Sprachen

### 1.4.1 Formale Sprachen

In der theoretischen Informatik spielt der Begriff der formalen Sprache eine große Rolle. Hier sind die wichtigsten Begriffe deshalb kurz erklärt. Die folgenden Definitionen sind dem Vorlesungsskript [Bachmann 2001] entnommen.

**Definition 1.1 (Formale Grammatik)** *Eine* Grammatik $G = (V, \Sigma, S, R)$ *besteht aus*

- *einer endlichen Menge $V$, dem sogenannten* Vokabular*,*
- *einem* Alphabet $\Sigma$ *mit* $\Sigma \subset V$*,*
- *einem Element $S \in V - \Sigma$, dem sogenannten* Startsymbol*, und*
- *einer endlichen Relation $R \subseteq (V^* - \Sigma^*) \times V^*$, der sogenannten* Regelmenge*.*

**Definition 1.2 (Ableitungsrelation)** *Jede Grammatik $G = (V, \Sigma, S, R)$ definiert eine* Ableitungsrelation $\rightarrow_G \subseteq V^* \times V^*$ *durch* $\varphi \rightarrow_G \varphi' :\Longleftrightarrow \exists(\alpha, \beta) \in R\colon\ \varphi = \varphi_1 \alpha \varphi_2 \wedge \varphi' = \varphi_1 \beta \varphi_2$.

**Definition 1.3 (Sprache)** *Die von der Grammatik $G = (V, \Sigma, S, R)$ erzeugte* Sprache $L(G)$ *ist definiert durch* $L(G) = \{\varphi \mid S \rightarrow_G^* \varphi \wedge \varphi \in \Sigma^*\}$.

#### Schreib- und Sprechweisen

- Statt $\rightarrow_G$ wird oft kurz $\rightarrow$ geschrieben.
- Regeln werden oft $\alpha \rightarrow \beta$ statt $(\alpha, \beta)$ geschrieben.
- Die Elemente von $\Sigma$ nennt man oft *Terminale* oder *Grundsymbole*.
- Die Elemente von $V - \Sigma$ nennt man oft *Nichtterminale* oder *Metasymbole*.

**Beispiel** Betrachten wir die Grammatik $G$ mit dem Alphabet $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$, dem Startsymbol $S = z$ und den Regeln $s \rightarrow 0, \ldots, s \rightarrow 9, s \rightarrow s0, \ldots, s \rightarrow s9$. Dann ist erzeugte Sprache die Menge aller Ziffernfolgen, d. h. $L(G) = \{0, 1, \ldots 00, 01, \ldots 10, \ldots, 2854564, \ldots\}$.

### 1.4.2 Einteilung der Sprachen

Der Begriff „Sprache“, mit dem wir hier immer wieder zu tun haben werden, hat eine sehr große Bedeutungsweite. In unterschiedlichen Zusammenhängen kann damit etwas anderes gemeint sein. So wird er beispielsweise in der Linguistik anders gebraucht als in der Mathematik oder Informatik. Deshalb sei hier eine grobe Einteilung zur besseren Begriffsbestimmung:

1. Formale Sprachen
   Bei formalen Sprachen kommt es nur auf die formalen Eigenschaften an, die durch ihre Grammatik definiert werden. Die Wörter haben im Allgemeinen keine besondere Bedeutung. Ein bekanntes Beispiel dafür sind die Palindrome, die aus den Buchstaben des lateinischen Alphabets gebildet werden, z. B. *xyzyx*.

2. Formale Sprachen mit Bedeutung
   Ein weiteres Beispiel für eine formale Sprache ist die dekadische Zifferndarstellung natürlicher Zahlen (siehe Beispiel oben). Hier haben die einzelnen Wörter jedoch sehr wohl eine Bedeutung, weil sie eine bestimmte Zahl darstellen. Diese Bedeutung lässt sich aus den formalen Eigenschaften herleiten. Weitere wichtige Beispiele für formale Sprachen, deren Wörter eine Bedeutung haben, sind Programmiersprachen.

3. Natürliche Sprachen
   Als *natürliche Sprachen* werden die Sprachen bezeichnet, die von Menschen gesprochen werden. Die Forschung der letzten 60 Jahre ist zu dem Ergebnis gekommen, dass sie nicht wie formale Sprachen beschrieben werden können.

Bei formalen Sprachen geht es, wie der Name schon nahelegt, um formale Eigenschaften. Diese formalen Eigenschaften kann man sich zunutze machen, indem man Regeln definiert, was sie bedeuten sollen. So entsteht neben der Syntax eine neue Ebene von Grammatik, nämlich die Semantik. Das ist etwa bei Programmiersprachen sehr praktisch. Davon unterscheiden sich natürliche Sprachen sehr stark. Hier gibt es zwar schon bestimmte Regeln, nach denen diese funktionieren, aber sie wurden nicht explizit festgelegt, sondern existieren implizit in den Gehirnen der Sprecherinnen und Sprecher. Dementsprechend ist es sehr schwierig (wenn nicht sogar unmöglich) zu sagen, nach welchen Regeln etwa die deutsche Sprache funktioniert. Hier sind Mehrdeutigkeiten üblich und sowohl Kontext als auch Vorwissen der Gesprächspartner und -partnerinnen sind sehr wichtig. Im schlimmsten Fall können Missverständnisse noch durch Nachfragen beseitigt werden. Deshalb spielt auch bei der Verarbeitung menschlicher Sprachen Intuition eine wichtige Rolle, die sich nur sehr schwer formalisieren lässt.

Die mathematischen Formeln, mit denen wir uns beschäftigen, fallen nicht eindeutig in eine der drei Kategorien, sondern liegen zwischen formalen Sprachen mit Bedeutung und natürlichen Sprachen. Die Formelschreibweise ist zwar sehr stark formalisiert, da Formeln aber von und für Menschen geschrieben werden und immer irgendwie menschliche Gedanken ausdrücken, müssen auch einige Aspekte aus der Welt der natürlichen Sprachen berücksichtigt werden. Ich habe deshalb als Anregung für diese Arbeit zahlreiche Konzepte aus der Sprachwissenschaft verwendet, ohne jedoch jedes Mal ausdrücklich darauf einzugehen. Eine sehr ausführliche Einführung und Übersicht gibt es z. B. bei [Vater 1994].

## 1.5 Ebenen von Grammatik

Oft sind Fehler aufschlussreicher als korrekte Ausdrücke. Betrachten wir daher einige falsche Formeln:

```
\Sin(x)
```

Diese Formel kann von unserem Parser problemlos verarbeitet werden, und es ist auch klar, was sie heißen soll. Wenn wir sie mit LaTeX verarbeiten wollen, erhalten wir jedoch eine Fehlermeldung, weil der Befehl `\Sin` nicht definiert ist.

```
\sin\cos x
```

Hier erzeugt LaTeX keine Fehlermeldung, und auch der Parser kann die Formel (richtig?) interpretieren. Trotzdem kommt sie uns falsch vor. Wir würden korrekt $\sin(\cos x)$ schreiben.

```
sin(x)
```

Auch hier gibt es bei LaTeX keine Probleme und es ist unmissverständlich klar, was gemeint ist. Der Parser kann die Formel jedoch nicht richtig verarbeiten (er liest sie wie $s \cdot i \cdot n \cdot (x)$ statt wie $\sin(x)$).

An diesen Beispielen wird deutlich, dass die Richtigkeit (oder eben Falschheit) von Formeln auf verschiedenen Ebenen stattfinden kann. Diese sind, wenn wir sie danach bezeichnen, wo die Formel verarbeitet wird, LaTeX, der Parser und der menschliche Leser / die menschliche Leserin. Vergleichbares gibt es durchaus auch in anderen Arten von Sprachen. Bei Programmiersprachen wird etwa zwischen Syntax und Semantik unterscheiden. So ist `a+b;` immer ein (syntaktisch) korrekter Ausdruck in der Sprache C. Er ist aber (semantisch) nur sinnvoll, wenn die Variablen `a` und `b` Datentypen sind, die addiert werden können (z. B. Zahlen). Ist aber `a` vom Typ `double` und `b` vom Typ `char*`, dann kann der Ausdruck nicht richtig interpretiert werden (weil in C keine Addition für Fließkommazahlen und Zeiger definiert ist). In der modernen Sprachwissenschaft unterscheidet man sogar vier verschiedene Ebenen von Grammatik, nämlich Phonologie, Morphologie, Syntax und Semantik.[3]

Das Interessante an diesen Ebenen von Grammatik ist, dass sie ziemlich unabhängig voneinander behandelt werden können. Wie wir an den Beispielen oben gesehen haben, kann eine Formel, die auf einer Ebene falsch ist, auf allen anderen Ebenen richtig sein. Das liegt daran, dass die Sprachverarbeitung in mehreren Phasen erfolgt. Wenn etwa Menschen Sprache hören, dann werden die Schallwellen zuerst phonologisch analysiert und das Gehörte wird in Phoneme gegliedert. Bei der morphologischen Verarbeitung werden anschließend bestimmte Gruppen von Phonemen zu Morphemen zusammengefasst. Dann wird die Syntax des Gesprochenen analysiert. Zum Schluss wird daraus die Semantik abgeleitet (siehe Abb. 1.1). Auch Compiler (oder Interpreter) analysieren zuerst

[3] Eine Darstellung davon ist in Abb. 1.1 zu sehen. Dort ist ein Satz zuerst phonologisch in Lautschrift dargestellt. Dann wird die Morphologie durch einen Klammerausdruck beschrieben, wobei die Morpheme mit tiefgestelltem N, V, oder X gekennzeichnet sind, je nachdem, ob sie zur Bildung von Nomen oder Verben verwendet werden oder gar nicht zur Derivation dienen. Zuletzt ist noch der Syntaxbaum zu sehen, in dem der Satz (S) in verschiedene Phrasen zerlegt ist (Nominalphrase NP und Verbalphrase VP), die wiederum aus kleineren Bestandteilen zusammengesetzt sind (Nomen N, Verb V, Determinator D).

die Syntax des Quellcodes und dann die Semantik. Diese Vorgehensweise werden wir hier übernehmen und die Formeln auf drei Ebenen verarbeiten. Wir lesen zuerst einen String von Zeichen ein. Den zerlegen wir dann so in Einzelteile, dass jedes Einzelteil für LaTeX eine bestimmte Bedeutung hat. Dann erzeugen wir daraus einen Syntaxbaum, und anschließend analysieren wir dessen Semantik.

Sie isst einen Apfel.

[ziː ˈɪst ˌaɪnən ˈapfəl]

$[\text{Sie}]_{\text{N}}\ [[\text{iss}]_{\text{X}}\text{t}]_{\text{V}}\ [[\text{ein}]_{\text{X}}\text{en}]_{\text{N}}\ [\text{Apfel}]_{\text{N}}$

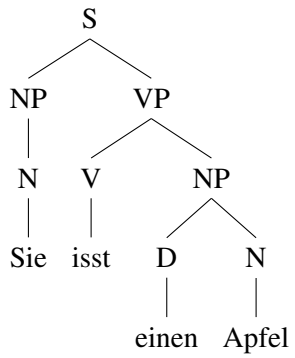



Abbildung 1.1: Phonologie – Morphologie – Syntax

## 1.6 Darstellung von Mathematik und Formeln

### 1.6.1 Darstellung von Formeln

#### Standard-Schreibweise

Die Schreibweise für Formeln, die heutzutage üblicherweise verwendet wird, ist in sich sehr uneinheitlich. Ein Beispiel ist die Definition der ganzzahligen Gamma-Funktion:

$$\Gamma(n) = (n-1)! = \prod_{i=1}^{n-1} i$$

Hier treten Formelzeichen mit unterschiedlichsten Eigenschaften auf: es gibt Funktionen, Variable, arithmetische Operatoren, Klammern, große Operatoren und sogar einen Operator, der hinter dem Operanden steht. Die Gründe für diese Vielfalt an Notationsarten sind historisch.

Man kann diese Notationsart auch in verschiedene „Schichten“ von Schreibweisen einteilen. Die elementarste Schicht sind einfache arithmetische Ausdrücke wie $1+2$. Diese wird dann durch Variablen und Funktionen erweitert, sodass auch Formeln wie $\sin(x)$ geschrieben werden können. Dann gibt es noch grafische Elemente, bei denen etwas nicht duch ein explizites Zeichen sondern durch Schriftarten, Hoch- oder Tiefstellungen, Akzente usw. ausgedrückt wird. Das sind etwa Potenzen

wie $x^2$ oder Brüche wie $\frac{a}{b}$ oder auch $\bar{a} \in \mathbb{N}$. Eine weitere Besonderheit sind die sogenannten „großen Operatoren“, die Ober- und Untergrenzen haben, etwa $\sum_{i=1}^{n} x_i$. Man kann sich das so vorstellen, wie Kinder die Formelschreibweise in der Schule erlernen: Zuerst gibt es nur die Grundrechnungsarten, und die Menge der darstellbaren Formeln wird dann durch Einführen neuer Zeichen und Schreibweisen sukzessive erweitert.

Ein bekanntes Problem dabei sind schon bei elementaren Ausdrücken die Rangordnung und Assoziativität von Operatoren. Etwa gilt $1+2\cdot 3=7$ und $2\uparrow 2\uparrow 3=256$. Dadurch ist die Verwendung von Klammern notwendig, um Ausdrücke wie $(1+2)\cdot 3$ oder $(2\uparrow 2)\uparrow 3$ darstellen zu können.

Diese Standard-Schreibweise hat den Vorteil, dass sie meist als sehr anschaulich und für Menschen gut leserlich empfunden wird. Nachteile sind die schon erwähnte Vielfältigkeit der Notationsarten und die Tatsache, dass sie für die Verarbeitung durch Rechenmaschinen gänzlich ungeeignet ist. Deshalb wurden noch einige andere Darstellungsformen entwickelt, die diese Probleme umgehen. Im Folgenden werden einige Beispiele dafür vorgestellt.

### Postfix-Notation

Eine Schreibweise, die diese Probleme umgeht, ist die sogenannte Postfix-Notation oder umgekehrte polnische Notation (UPN; benannt nach dem polnischen Logiker Jan Łukasiewicz, der sie in den 1920er Jahren entwickelte). Hier ist jedes Zeichen entweder ein Operand oder ein Operator. Die Operatoren stehen immer hinter ihren Operanden, wodurch die Reihenfolge der Berechnung eindeutig ist. Die Formel $\sin(1+2*3)$ würde in Postfix-Notation so aussehen:

$$1\ 2\ 3 * + \sin$$

Der große Vorteil der UPN besteht darin, dass sie eine Stack-basierte Auswertung ermöglicht und daher von Rechenmaschinen sehr schnell und einfach verarbeitet werden kann. Deshalb wurde sie früher als Eingabeformat für HP-Taschenrechner verwendet, und Programmiersprachen wie Forth oder Postscript basieren auf ihr. Nachteile der UPN sind, dass sie für Menschen nur schwer lesbar ist und die Art und Weise, wie algebraische Veränderungen von Formeln (z. B. symbolisches Differenzieren) in ihr durchgeführt werden, sehr umständlich wirkt.

### Funktionen-Schreibweise

Man kann auch jeden Operator als Funktion auffassen. Wenn die Argumente der Funktionen dann eingeklammert sind und mehrere Argumente durch Kommata getrennt werden, sieht die Formel $\sin(1+2*3)$ in Funktionen-Schreibweise so aus:

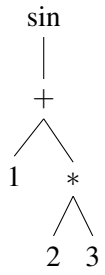
$$\sin(+(1,*(2,3)))$$

Lässt man die Klammern und Kommata weg, bleibt $\sin + 1 * 2\ 3$ übrig. Das ist genau die Umkehrung der UPN. Die Funktionen-Schreibweise (mit Klammern und ohne Kommata) wird in LISP verwen-

det, und auch in der Programmiersprache C wird etwa der Ausdruck $x^y$ nicht durch einen Operator, sondern mit der Funktion `pow(x,y)` dargestellt.

#### Bäume

Formeln können auch als Bäume dargestellt werden. Dafür gibt es verschiedene Möglichkeiten, und wir beschränken uns hier wie schon erwähnt auf Binärbäume. Jeder Baum besteht aus einem Knoten und höchstens zwei Unterbäumen. Der Knoten enthält den Operator und die Unterbäume die Operanden. Handelt es sich um eine Zahl oder Variable, dann sind beide Unterbäume leer. Unsere Formel würde als Baum so aussehen:




Die Baum-Darstellung hat den Vorteil, dass sie sehr gut für das weitere Verarbeiten von Formeln (insbesondere für algebraische Manipulationen) geeignet ist. Der Nachteil an der grafischen Darstellung von Bäumen, wie sie hier gezeigt wurde, ist, dass sie sehr viel Platz braucht. Es gibt jedoch zu jedem Baum einen äquivalenten Klammerausdruck, der den Baum platzsparender darstellt:

$$[\sin[+[1][*[2][3]]]]$$

Diese Schreibweise ist in der Sprachwissenschaft weit verbreitet und wir werden sie auch verwenden, um Bäume kompakter darzustellen.

### 1.6.2 Mathematik und Formeln

Formeln dienen dazu, mathematische Sachverhalte aufzuschreiben. Sind etwa $a$ und $b$ zwei natürliche Zahlen und ist $a$ die größere dieser beiden Zahlen, dann kann man als Formel $a > b$ schreiben. Dies ermöglicht es, etwas schriftlich festzuhalten. Trotzdem ist das nur eine Schreibweise! Die eigentliche Mathematik (also der dargestellte Sachverhalt) ist etwas vollkommen anderes, das dadurch nur beschrieben wird.

Nehmen wir als anderes Beispiel die menschliche Sprache. Ein Mensch geht in ein Geschäft und möchte Brot kaufen. Er wendet sich an die Verkäuferin. Irgendwo in seinem Gehirn gibt es den Gedanken, dass er ein Brot möchte. Diesen Gedanken muss er jetzt der Verkäuferin vermitteln. Deshalb sagt er: „Bitte ein Brot.“ Der Satz „Bitte ein Brot“ ist jetzt keineswegs der Gedanke dieses Menschen, sondern ein sprachlicher Ausdruck, mit dem dieser Gedanke der Verkäuferin mitgeteilt wird.

Ebenso verhält es sich auch mit Mathematik und Formeln. Die Mathematik selbst ist unsichtbar und besteht nur aus abstrakten Sachverhalten. Ein bestimmter Sachverhalt ist sozusagen ein Gedanke im Kopf des Mathematikers / der Mathematikerin. Formeln ermöglichen es uns, diese Gedanken aufzuschreiben. Sie sind aber selbstverständlich nur eine Schreibweise.

Es gibt bestimmte Regeln, wie Mathematik durch Formeln dargestellt wird. Für die Addition der Zahlen $x$ und $y$ schreibt man etwa $x+y$. Das Ergebnis (also die Formeln) lässt sich recht einfach als formale Sprache darstellen. Die formale Grammatik für einfache Arithmetik besteht etwa aus folgenden Regeln:

$$X \to NUM, \quad X \to X+X, \quad X \to X-X, \quad X \to X \cdot X, \quad X \to X:X$$

Das Problem dabei ist, dass diese formale Sprache zu mächtig für die Beschreibung einfacher ganzzahliger Arithmetik ist. Dadurch lässt sich nämlich auch die Formel $1:0$ darstellen, und es gibt keinen arithmetischen Ausdruck, der ihr entspricht (weil Division durch 0 mathematisch nicht möglich ist). Auch Formeln wie $7:3$ sind keine gültigen Ausdrücke der ganzzahligen Arithmetik. Umgekehrt kommt es auch vor, dass ein und derselbe mathematische Ausdruck durch mehrere Formeln dargestellt werden kann. Ist $x$ eine Funktion, dann gibt es für die Ableitung an der Stelle $t$ verschiedene Schreibweisen, die alle dasselbe bedeuten:

$$x'(t) \qquad \dot{x}(t) \qquad \frac{dx}{dt} \qquad \frac{\partial x}{\partial t}$$

Ein wichtiges Problem, mit dem wir uns befassen müssen, ist also, einerseits Mathematik unabhängig von ihrer Formeldarstellung zu behandeln, und andererseits herauszufinden, welcher mathematische Sachverhalt durch eine bestimmte Formel dargestellt wird.

### Semantikbäume

Zu diesem Zweck verwandeln wir die Syntaxbäume, durch die die Formeln dargestellt werden, in Semantikbäume, die die zugrundeliegende Mathematik darstellen. Betrachten wir noch einmal das Beispiel mit der ganzzahligen Arithmetik. Eine arithmetische Operation ist eine Abbildung $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, d. h. sowohl ihr Ergebnis als auch ihre beiden Argumente sind ganze Zahlen. Der Binärbaum, der die Operation darstellt, stellt also eigentlich eine ganze Zahl dar, die durch die Operation erzeugt wird. Um das beschreiben zu können, führen wir *Typen* ein, die angeben, was für ein mathematisches Objekt eine Formel bzw. ein Baum darstellt. Wenn wir ganze Zahlen mit dem Typ `NUM` bezeichnen, dann sind ein Binärbaum, der eine arithmetische Operation darstellt sowie seine beiden Unterbäume von Typ `NUM`.

Diese Typen geben also an, was für mathematische Objekte in einer Formel vorkommen. Außerdem ist aber noch wichtig, was mit diesen Objekten gemacht werden kann. dazu verwenden wir semantische *Kategorien*. Eine solche Kategorie wäre etwa die Addition, bei der zwei ganze Zahlen addiert werden.


Abbildung 1.2: Die Formel $1+2$ als Syntax- und als Semantikbaum.

### Verschiedene Schreibweisen

Es kommt durchaus oft vor, dass es zu einem Semantikbaum mehrere Syntaxbäume gibt. Sind etwa $a$ und $b$ zwei Variable, die Zahlen darstellen, dann gibt es für die Multiplikation von $a$ und $b$ (für die es genau einen eindeutigen Semantikbaum gibt!) zwei verschiedene Schreibweisen, nämlich $a \cdot b$ und $ab$. Beide bedeuten genau dasselbe, obwohl sie unterschiedlich aussehen. Welche Schreibweise verwendet wird, hat also stilistische Gründe wie Verdeutlichung der Multiplikation oder bessere Lesbarkeit.

Umgekehrt kann auch ein und dasselbe Zeichen mit vollkommen verschiedenen Bedeutungen verwendet werden. Ein bekanntes Beispiel dafür ist das Minus-Zeichen, das ein arithmetischer Operator oder ein Vorzeichen sein kann. Durch die Darstellung mit Semantikbäumen werden solche Mehrdeutigkeiten umgangen.

## 1.7 Der Shunting-Yard-Algorithmus von Dijkstra

Im Jahre 1961 veröffentlichte Edsger Dijkstra in [Dijkstra 1961] einen Algorithmus zur Umwandlung von Formeln in Postfix-Notation für die Programmiersprache ALGOL. Da ihn die Funktionsweise an die Arbeit auf einem Verschubbahnhof erinnerte, bezeichnete er ihn als „Shunting Yard Algorithm“. In [Pratt 1973] wurde der Algorithmus noch genauer beschrieben und in einigen Punkten erweitert. Heute ist dieses Verfahren im Compilerbau Standard. Deshalb wird der Algorithmus auch hier in leicht vereinfachter Form kurz beschrieben, weil wir ihn später als Grundlage für den Parser verwenden werden.

Stellen wir uns einen Taschenrechner vor, dessen Eingabe-Tastatur folgende Tasten hat:

```
0 1 2 3 4 5 6 7 8 9 . ( ) + - * / ^ sin cos
```

So lassen sich Formeln wie `3+sin(5.8)` eingeben. Damit diese dann berechnet werden können, müssen sie in Postfix-Form gebracht werden. Vorher muss die Formel noch in Tokens zerlegt werden. Ein solches Token ist ein String, der eine sinntragende Einheit darstellt, also etwa `+` oder `47.9` (die Ziffern einer Zahl werden also immer zu einem einzigen Token zusammengefasst). Diese Tokens werden dann in einem Vektor $v$ abgespeichert. Um die Formel weiter verarbeiten zu können, müssen

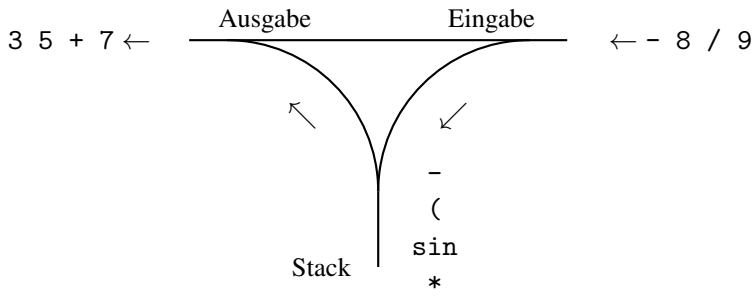



Abbildung 1.3: Shunting-Yard-Algorithmus

wir die Tokens kategorisieren. Jeder String in $v$ muss einen bestimmten Typ haben. Die Funktion `type` ordnet jedem Token seinen Typ zu:

| Typ | Name | Beispiele |
|---|---|---|
| `NUM` | Zahl | `1`, `38`, `0.49` usw. |
| `OP` | Operator | `+`, `-`, `*`, `/`, `^` |
| `FUNC` | Funktion | `sin`, `cos` |
| `LBRACK` | linke Klammer | `(` |
| `RBRACK` | rechte Klammer | `)` |

In den damit erzeugten Formeln sind Tokens vom Typ `NUM` Operanden und Tokens vom Typ `OP` oder `FUNC` sind Operatoren (binäre bzw. unäre). Klammern haben keine eigene Bedeutung, sie dienen nur dazu, Ausdrücke zusammenzufassen, die zuerst berechnet werden. Sie kommen daher in der Postfix-Notation nicht mehr vor. Bei Operatoren gibt es, wie schon erwähnt, eine Rangordnung. Die Funktion `prec` ordnet einem Operator seine Rangordnung zu. Unsere Operatoren haben folgende Rangordnungen:

| Operator | Rangordnung |
|---|---|
| `+` | 1 |
| `-` | 1 |
| `*` | 2 |
| `/` | 2 |
| `^` | 3 |

Wir müssen auch noch berücksichtigen, ob Operatoren links- oder rechtsassoziativ sind.[4] (Diese Assoziativität ist eine Vorschrift, in welcher Reihenfolge mehrere Vorkommnisse eines Operators ausgewertet werden, während das Assoziativgesetz in der Mathematik besagt, dass die Reihenfolge der Ausführung von mehrfachen Verknüpfungen keine Rolle spielt – die beiden Begriffe sind also nur beschränkt verwandt.)

[4] Diese Unterscheidung fehlt bei Dijkstra. Er behandelt alle Operatoren als linksassoziativ, obwohl der Potenzoperator ↑ rechtsassoziativ sein muss.

In unserem Beispiel ist ^ der einzige rechtsassoziative Operator. Wir definieren die Funktion `leftass` mit

$$\texttt{leftass}(x) = \begin{cases} 1 & x \text{ ist linksassoziativ} \\ 0 & x \text{ ist rechtsassoziativ} \end{cases}$$

Mit diesen Informationen können aus dem Vektor $v$ einen Vektor $p$ erzeugen, der dieselbe Formel in Postfix-Notation enthält. Dazu benötigen wir noch einen Zwischenspeicher (Stack), in dem wir Zeichen speichern können. Die Umwandlung erfolgt dann so: Der Vektor wird von Anfang bis Ende abgearbeitet. Zahlen werden in die Ausgabe geschrieben. Alle übrigen Zeichen werden auf dem Stack abgelegt. Ist das Zeichen ein Operator, so wird so lange das letzte Element des Stacks auf die Ausgabe geschrieben und anschließend vom Stack entfernt, solange es eine Funktion ist oder ein Operator mit gleicher oder höherer Rangordnung, wenn das Zeichen linksassoziativ ist, oder ein Operator mit höherer Rangordnung, wenn es rechtsassoziativ ist. Bei einer rechten Klammer werden bis zur nächsten linken Klammer alle Elemente vom Stack in die Ausgabe geschrieben. Wenn so der gesamte Vektor abgearbeitet ist, werden zum Schluss alle übrigen Einträge des Stacks in die Ausgabe geschrieben und anschließend entfernt. Dann ist der Stack leer und die Ausgabe enthält die Postfix-Notation des Ausdrucks.

**Beispiel** Die Formel

$$(3+5)*\sin(7-8/9) \tag{1.1}$$

hat folgende Postfix-Darstellung:

$$3\ 5+7\ 8\ 9/-\sin* \tag{1.2}$$

Die Umwandlung ist in Abb. 1.3 und Abb. 1.4 dargestellt.

### 1.7.1 Einschränkungen des Shunting-Yard-Algorithmus

Dieser Algorithmus eignet sich zwar sehr gut für den Compilerbau, er unterliegt aber zu starken Einschränkungen, als dass wir ihn in dieser Form verwenden könnten. In dieser vereinfachten Form fehlt zunächst einmal das Rechnen mit Variablen. Hier kann der Algorithmus aber sehr leicht erweitert werden, weil sich Variable syntaktisch genau wie Zahlen verhalten. Dafür fehlen aber noch Postfix-Operatoren wie ! für die Fakultät. Der Algorithmus kann so lange erweitert werden, bis alle Arten von syntaktischen Eigenschaften berücksichtigt sind. Dadurch würde er jedoch sehr umfangreich und unübersichtlich werden. Abgesehen davon muss von vornherein bekannt sein, welche Tokens Funktionen, Operatoren usw. sind. Das kann nachträglich nicht mehr geändert werden, was sehr unpraktisch ist (Das Zeichen $x$ kann in der Praxis als Funktion, Variable oder Operator auftreten). Das nächste Problem besteht darin, dass der Algorithmus nur bestimmte Formeln von einer einfachen Bauart verarbeiten kann. Sie lassen sich als formale Sprache mit folgenden Regeln beschreiben:

| Eingabe | Ausgabe | Stack |
|---|---|---|
| ( | | ( |
| 3 | 3 | ( |
| + | | ( + |
| 5 | 5 | ( + |
| ) | + | |
| * | | * |
| sin | | * sin |
| ( | | * sin ( |
| 7 | 7 | * sin ( − |
| − | | * sin ( − |
| 8 | 8 | * sin ( − / |
| / | | * sin ( − / |
| 9 | 9 | |
| ) | / | |
| | − | |
| | sin | |
| | * | |

Abbildung 1.4: Konvertierung von Formel 1.1 in Postfix-Notation

| Name | Regel | Beispiel |
|---|---|---|
| Zahl | $X \rightarrow$ `NUM` | 1 |
| Operation | $X \rightarrow X$ `OP` $X$ | $3+7$ |
| Klammern | $X \rightarrow$ `LBRACK` $X$ `RBRACK` | $(5+9)$ |
| Funktion | $X \rightarrow$ `FUNC` `LBRACK` $X$ `RBRACK` | $\sin(0)$ |

Dadurch lassen sich aber sogar sehr häufige Ausdrücke wie $-1$, $\sin 0$ oder $(1+2)(1-2)$ nicht mehr darstellen. Kompliziertere Formeln wie $\mathbb{R}^+$ oder $\{\}$ würden noch größere Schwierigkeiten machen. Sie verstoßen nämlich gegen die oben definierten Regeln, die der Algorithmus jedoch braucht. Ein letztes Problem besteht noch darin, dass Klammern in dem Algorithmus nur die Reihenfolge der Berechnung steuern und in der Ausgabe nicht mehr vorkommen. Es gibt allerdings auch Formeln, in denen Klammern eine größere Bedeutung haben und auch in der Ausgabe nicht fehlen dürfen (z. B. Matrizen, die von Klammern umschlossen sein müssen).

Wir müssen den Algorithmus also so verallgemeinern, dass er sich auch auf Formeln anwenden lässt, die gegen diese restriktiven Regeln verstoßen, und alle Eingabe-Tokens auch irgendwie in die Ausgabe übernimmt. Dieser erweiterte Shunting-Yard-Algorithmus wird in Abschnitt 3.3 behandelt.

**Algorithmus 1.7.1** Shunting-Yard-Algorithmus

Eingabe: der Vektor *in* enthält die Formel.

Ausgabe: der Vektor *out* enthält die Formel in Postfix-Notation.

Sonstige Variable: Zahlen $i, t, p$, String $s$

**0** Setze $i \leftarrow 0$

**1** Setze $s \leftarrow in_i$, $t \leftarrow \texttt{type}(s)$

**2** Falls $t \neq \texttt{NUM}$ weiter bei Schritt 3; $\texttt{push}(out, s)$

**3** Falls $t \neq \texttt{FUNC}$ weiter bei Schritt 4; $\texttt{push}(stack, s)$

**4** Falls $t \neq \texttt{OP}$ weiter bei Schritt 5; $p \leftarrow \texttt{prec}(s)$

**5** Falls $\texttt{type}(\texttt{back}(stack)) \neq \texttt{OP}$ weiter bei Schritt 12.
Falls $(\texttt{leftass}(\texttt{back}(stack)) = 1 \wedge \texttt{prec}(\texttt{back}(stack)) \geq p) \vee (\texttt{leftass}(\texttt{back}(stack)) = 0 \wedge \texttt{prec}(\texttt{back}(stack)) > p)$ weiter bei Schritt 12; sonst $\texttt{push}(out, \texttt{back}(stack))$ und $\texttt{pop}(stack)$.

**6** Wiederhole Schritt 5, solange $\texttt{size}(stack) > 0$ ist.

**7** Falls $t = \texttt{LBRACE}$: $\texttt{push}(stack, s)$

**8** Falls $t = \texttt{RBRACE}$: $\texttt{push}(out, \texttt{back}(stack))$, $\texttt{pop}(stack)$

**9** Wiederhole Schritt 6, wenn $\texttt{type}(\texttt{back}(stack)) \neq \texttt{LBRACE}$.

**10** $\texttt{pop}(stack)$

**11** Falls $\texttt{type}(\texttt{back}(stack)) = \texttt{FUNC}$:
$\texttt{push}(out, \texttt{back}(stack))$ und $\texttt{pop}(stack)$

**12** Setze $i \leftarrow i + 1$; falls $i \leq \texttt{size}(stack)$, gehe zurück zu Schritt 1.

**13** Falls $\texttt{size}(stack) > 0$: $\texttt{push}(out, \texttt{back}(stack))$, $\texttt{pop}(stack)$

**14** Wiederhole Schritt 13, solange $\texttt{size}(stack) > 0$ gilt.

# 2 LaTeX

Die unterste Ebene von Grammatik, mit der wir uns hier beschäftigen, ist die Formatierungssprache LaTeX. Die dadurch definierten Regeln, mit denen Formeln dargestellt werden können, sind jedoch nicht eindeutig; es gibt oft verschiedene Möglichkeiten, ein und dieselbe Formel darzustellen. Wir sagen, zwei Formeln sind für LaTeX gleich, wenn sie (optisch) dieselbe Ausgabe produzieren. So sind etwa `\frac12` und `\frac{1}{2}` gleich, während `1/2` eine andere Formel erzeugt, die aber dasselbe bedeutet. Ein weiteres einfaches Beispiel für Unterschiede in der Darstellung gleicher Formeln sind Leerzeichen: Die Formeln `x+x` und `x␣+␣y` sind natürlich gleich, weil beide die Ausgabe $x+y$ produzieren und die Leerzeichen darauf keinen Einfluss haben. Ein etwas komplizierteres Beispiel sind `x_1^2` und `x^2_1`, die auch gleich sind.

In diesem Abschnitt wird es also darum gehen, die Formeln in eine *eindeutige* Darstellung zu bringen. Das heißt, das Ergebnis ist eine restriktivere Darstellungsform, die keine Mehrdeutigkeiten mehr zulässst. Aus ihr kann dann die Syntax der Formel wesentlich leichter hergeleitet werden.

$$(x_1 + 3x_2^4)^5$$

```
(x_1^{} + 3 x_2^4)^5
```

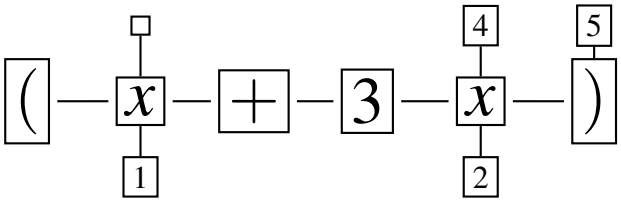



Abbildung 2.1: Interne Darstellung einer Formel in TeX

Leider ist die Art und Weise, wie Formeln in TeX und LaTeX dargestellt werden, für unsere Zwecke vollkommen ungeeignet.[1] Eine Formel ist für TeX eine *mlist*, d. h. eine verkettete Liste, deren Einträge (die sogenannten *noads*) Zeichen sind, die noch eine hoch- und eine tiefgestellte Formel haben können. Für die Formatierung ist es ein Unterschied, ob diese nur leer sind oder fehlen. Bei

[1] Die hier vorgestellten Informationen sind eine Zusammenfassung von [Knuth 2000], Band B: *TeX: The Program, Part 34, Data Structures for Math Mode*.

`x^2` fehlt die tiefgestellte Formel, bei $x^2$ ist sie da, aber leer.

Für die Analyse der Syntax und Semantik einer Formel müssen wir diese jedoch als Baumstruktur darstellen. Da das über die Funktionalität von TeX hinausgeht, beschäftigen wir uns in diesem Kapitel nur damit, eine eindeutige Schreibweise zu erzeugen. Alles Weitere wird dann später behandelt.

## 2.1 Zerlegung eines Strings in Tokens

Wir gehen immer davon aus, dass jede Formel als genau ein String vorliegt und nicht auf mehrere Strings aufgeteilt ist. Das heißt, wir kümmern uns um Schreibweisen wie `$x+$$y$` nicht. Dass so etwas vorkommt, ist auch nur dann sinnvoll, wenn eine Formel auf mehrere Zeilen aufgeteilt wird. Ansonsten würde diese Schreibweise die Abstandsregulierung des Mathematik-Modus zerstören. Für LaTeX sind die einzelnen Zeichen nicht so sehr von Bedeutung, sondern Tokens, also Gruppen von zusammengehörigen Zeichen. In der Formel `\cos x + i \sin x` kommt das Zeichen `i` zwei Mal vor. Es ist jedoch nur einmal eine eigene Einheit, das andere Mal ist es ein Teil des Tokens `\sin`. Unsere erste Aufgabe besteht darin, den String, der die Formel enthält, in einen Vektor von Tokens zu zerlegen. Eine solche Zerlegung sieht etwa so aus:

`\sin x=\cos(x+\pi/2)`

| \sin | x | = | \cos | ( | x | + | \pi | / | 2 | ) |
|---|---|---|---|---|---|---|---|---|---|---|

Diese Tokens können dann weiter verarbeitet werden.

### 2.1.1 Zeichen

Die Zeichen, die zur Darstellung von Formeln im Mathematik-Modus verwendet werden, lassen sich in drei Gruppen gliedern:

1. Leerzeichen: Neben dem klassischen Leerzeichen gehören auch Tabulator und Zeilenumbruch zu dieser Kategorie.
2. Buchstaben: Das sind alle Groß- und Kleinbuchstaben des lateinischen Alphabets.
3. Sonstige Zeichen: Eine besondere Rolle hat hier der Backslash, weil mit ihm Befehle beginnen (siehe unten).

### 2.1.2 Einfache Zerlegung

Zu welchem Token ein Zeichen gehört, wird nach folgenden Prinzipien entschieden:

1. Einzelne Zeichen

   Prinzipiell ist im Mathematik-Modus jedes Zeichen ein eigenes Token; die Formel `abc` besteht etwa aus drei Tokens. Ausnahme werden unten behandelt.

2. Leerzeichen

   Anders als im Textmodus haben Leerzeichen in Formeln keine Bedeutung. Sie können zwischen Tokens stehen und markieren das Ende eines Befehls. Dabei ist es gleichgültig, wieviele Leerzeichen nebeneinander stehen.

3. Befehle

   Befehle beginnen immer mit einem Backslash. Das unmittelbar darauf folgende Zeichen gehört dann auch zu dem Befehl. Ist ein Zeichen in einem Befehl ein Buchstabe, dann gehört das nächste Zeichen auch zu dem Befehl, wenn es ein Buchstabe ist.

Die Zerlegung eines Strings in einen Vektor von Tokens geschieht folgendermaßen: Der String wird von links nach rechts abgearbeitet. Leerzeichen werden ignoriert. Ist das aktuelle Zeichen kein Leerzeichen, dann wird es an das Token angehängt. Ist es kein Backslash, dann ist das Token schon vollständig. Handelt es sich jedoch um einen Backslash, dann wird auch das nächste Zeichen an das Token angehängt. Ist dieses ein Buchstabe, dann wird so lange das nächste Zeichen an das Token angehängt, solange es ein Buchstabe ist. Dann ist das Token vollständig. Das fertige Token wird an den Vektor angehängt, der zum Schluss die Zerlegung der Formel enthält.

**Algorithmus 2.1.1** Zerlegung eines Strings in Tokens

Eingabe: der String $s$ enthält die LaTeX-Formel.

Ausgabe: der Vektor $tokens$ enthält die Tokens, in die die Formel zerlegt worden ist.

Sonstige Variable: Zahlen $i$, $j$, Zeichen $c$

**0** Setze $i \leftarrow 0, j \leftarrow 0$

**1** Falls $s_i$ ein Leerzeichen ist: setze $i \leftarrow i+1$, solange $s_i$ ein Leerzeichen ist.

**2** Setze $c \leftarrow s_i$ und $tokens_j \leftarrow tokens_j c$.

**3** Falls $c$ der Backslash ist: setze $i \leftarrow i+1$ und $tokens_j \leftarrow tokens_j s_i$, solange $s_i$ ein Buchstabe ist.

**4** Setze $j \leftarrow j+1$, $i \leftarrow i+1$ und gehe zurück zu Schritt 1.

### 2.1.3 Sonderfälle

Durch diese Vorgehensweise erhalten wir eine Zerlegung in Tokens, wie sie auch von LaTeX gemacht wird. Bevor LaTeX die endgültige Ausgabe erzeugt, gibt es jedoch noch einige Zwischenschritte, in denen diese Tokens weiter verarbeitet werden. Deshalb müssen wir unsere Zerlegung auch noch etwas verändern, um Tokens zu erhalten, die in etwa den Bestandteilen der fertigen Formel entsprechen.

### Schriftarten und Ähnliches

Formelzeichen können durch die Schriftart unterschieden werden wie etwa hier:

```
N \in \mathbb N
```
$$N \in \mathbb{N}$$

In solchen Fällen ist es günstig, die Buchstaben und den Befehl für die Schriftart zu einem einzigen Token zusammenzufassen. Dabei ist zu berücksichtigen, dass Schriftarten nur für Buchstaben gelten und ein Befehl mehrere Buchstaben verändern kann.

| `\mathbb{N, Z \subset Q}` | `\mathbb{N}` | `,` | `\mathbb{Z}` | `,` | `\subset` | `\mathbb{Q}` |
|---|---|---|---|---|---|---|

Es gibt noch andere Möglichkeiten, die Schriftart zu verändern (z. B. mit `\selectfont`). Wichtig ist hier, dass die Informationen über die Schrift im Token enthalten sind. Andererseits ist auch zu berücksichtigen, dass manche Symbole (etwa griechische Buchstaben) davon nicht betroffen sind.

Die Befehle `\operatorname` und `\text` verhalten sich ähnlich. Der Unterschied ist hier, dass sie gemeinsam mit ihrem Argument ein einziges Token bilden.

```
\operatorname{sin}(x)
```

| `\operatorname{sin}` | `(` | `x` | `)` |
|---|---|---|---|

### Befehle zur Größenregulierung von Klammern

Mit Befehlen wie `\left`, `\right`, `\big` usw. kann die Größe von Klammern verändert werden. Es ist jedoch vorteilhaft, sie nicht als eigene Tokens aufzufassen, sondern sie mit der zugehörigen Klammer zu verbinden. So ist dann etwa `\left(` ein eigenes Token:

```
\left( \frac a b \right)^2
```

| `\left(` | `\frac` | `a` | `b` | `\right)` | `^` | `2` |
|---|---|---|---|---|---|---|

### Styles

Die Style-Befehle wie `\displaystyle` verändern die Größe von Formelzeichen. Das hat aber keinerlei Einfluss auf die Bedeutung, weil Formelzeichen nicht durch die Schriftgröße unterschieden werden können, wie das folgende Beispiel zeigt:

```
\frac a b = \displaystyle \frac a b
```
$$\tfrac{a}{b} = \frac{a}{b}$$

Die Style-Befehle können also genauso gut weggelassen werden, ohne dass sich etwas an der Bedeutung der Formel ändert.

### Explizite Leerzeichen

Gelegentlich treten in Formeln explizite Leerzeichen auf, die zwar in der endgültigen Darstellung auftauchen, aber keine eigene Bedeutung haben, sondern nur der besseren Lesbarkeit dienen. Sie können ebenfalls weggelassen werden.

```
10\,000      10000
ab\in K \quad \forall a, b \in K
```
$$10\,000 \qquad 10000$$
$$ab \in K \quad \forall a, b \in K$$

## 2.2 Besondere Tokens

### 2.2.1 Kategorisierung der Tokens

Es gibt einige Tokens, die eine besondere Bedeutung haben:

```
^        _        {        }
```

Sie besitzen zunächst einmal die Eigenschaft, dass man sie im gedruckten Text nicht sehen kann, weil es nämlich nur Steuerzeichen in LaTeX sind, die Formatierungsattribute wie Hoch- oder Tiefstellung von Text festlegen. Außerdem können sie bzw. ihre Bedeutung nicht verändert werden. Das Zeichen $f$ kann beispielsweise für eine Funktion, ein Element eines Körpers oder irgendetwas anderes stehen, je nachdem, wie es gebraucht wird. Das Token `\\` bewirkt hingegen immer einen Zeilenumbruch. In der Sprachwisenschaft unterteilt man den Wortschatz einer Sprache (das sogenannte Lexikon) in funktionale und lexikalische Kategorie. Das Wort „und“ gehört etwa zur funktionalen Kategorie: Es hat eine bestimmte Funktion in der Syntax, seine Bedeutung kann sich nicht leicht verändern und es lässt sich nicht durch andere Wörter umschreiben. Ein Beispiel für die lexikalische Kategorie ist das Wort „Zug“: Es kann je nach Zusammenhang verschiedene Bedeutungen haben, man kann es leicht umschreiben und durch andere Wörter ersetzen. Ähnlich gibt es in Programmiersprachen sogenannte Schlüsselwörter oder reservierte Wörter. Das sind Wörter, die eine bestimmte vordefinierte Funktion erfüllen. Im Normalfall dürfen sie nicht anders verwendet werden. Beispiele für Schlüsselwörter in C sind `int`, `if` und `return`. Derselbe Effekt tritt auch in LaTeX-Formeln auf, und deshalb ist bei bestimmten Tokens immer von vornherein klar, welche Bedeutung bzw. Funktion sie haben.

Befehle sind ein Mittelding zwischen diesen beiden Kategorien. Es ist zwar möglich, neue Befehle zu definieren oder vorhandene Befehle umzudefinieren, nicht definierte Befehle sind jedoch unzulässig. Jeder Befehl, der irgendwo auftritt, muss also vorher definiert worden sein. Aus der Definition ist auch bekannt, wieviele Argumente und optionale Argumente er hat. Das ist für die Verarbeitung sehr wichtig, wie beim Vergleich dieser beiden Formeln deutlich wird:

```
\pi x      πx
\sqrt x    √x
```

Im zweiten Fall gehört das `x` zum Befehl `\sqrt`, weil dieser ein Argument haben muss.

Prinzipiell können Befehle beliebig viele Argumente haben. Alle Befehle zur Darstellung von Formeln sind jedoch irgendwie aus elementaren vordefinierten Befehlen zusammengesetzt, und da gibt es folgende Varianten: keine Argumente (z. B. `\pi`), ein Argument (z. B. `\hat`), ein Argument und ein optionales Argument (z. B. `\sqrt`) und zwei Argumente (z. B. `\frac`). Wir beschäftigen uns im Weiteren nur mit diesen vier Typen von Befehlen. Kompliziertere Befehle mit mehr Argumenten müssen dann durch einen Präprozessor durch elementare Befehle ersetzt werden (genau so werden sie auch von LaTeX verarbeitet).

### 2.2.2 Klammern und Gruppen

Mit geschweiften Klammern werden Tokens zu Gruppen zusammengefasst. Diese Gruppierung beeinflusst die Formatierung und regelt, auf welche Tokens Befehle wirken. Optionale Argumente von Befehlen müssen von eckigen Klammern umschlossen sein.

### 2.2.3 Befehle

Wenn LaTeX-Befehle Argumente haben, dann wirken sie auf die darauffolgende Gruppe(n) von Tokens. Eine solche Gruppe besteht entweder nur aus einem einzige Token oder aus allen Tokens, die von einem Paar geschweifter Klammern umschlossen sind. Manche Befehle haben auch ein optionales Argument, das in eckigen Klammern angegeben wird. Hier ist als Beispiel der Befehl `\mathbf`, der auf eine Zeichengruppe wirkt, einmal mit und einmal ohne Klammern:

| `\mathbf` | `x` | `=` | `\mathbf` | `{` | `y` | `z` | `}` |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

$\mathbf{x} = \mathbf{yz}$

Der Befehl `\mathbf` steht in dem Vektor, der die Formel enthält, an 0. und 3. Stelle, und das dazugehörige Argument ist dann die erste Klammergruppe nach dem 0. bzw. 3. Eintrag des Vektors, was hier grau unterlegt ist.

### 2.2.4 TeX-Befehle

Die alten Befehle, die von TeX zur Formatierung verwendet wurden, unterscheiden sich nicht unwesentlich von den typischen LaTeX-Befehlen. Sie gelten immer innerhalb einer Gruppe, die sie gewissermaßen in zwei Hälften teilen. Diese Gruppe ist entweder die ganze Formel oder wird von geschweiften Klammern umschlossen.

```
n \choose k          $n \choose k$
{ 2 \over 2 } = 1    ${ 2 \over 2 } = 1$
```

Statt geschweifter Klammern können auch Klammern mit `\left` und `\right` verwendet werden:

```
\left( a \over b \right)^2          $\left( a \over b \right)^2$
```

Die TeX-Befehle sind nicht-assoziativ, d. h. Ausdrücke wie `a \over b \over c` sind nicht zulässig. Stattdessen muss die Reihenfolge durch geschweifte Klammern geregelt werden, und die zulässige Schreibweise wäre dann

```
{ a \over { b \over c } }
```

Damit verhalten sich diese Befehle syntaktisch wie binäre Operatoren, die zwischen ihren Operanden stehen und deren Rangordnung durch geschweifte Klammer geregelt werden muss.

### 2.2.5 Wurzeln

Zur Darstellung von Wurzeln (z.b. $\sqrt[3]{x}$) gibt es einerseits den Befehl `\sqrt`, der ein Argument und in LaTeX auch ein optionales Argument hat, und andererseits den TeX-Befehl `\root`. Seine Syntax ist etwas außergewöhnlich, denn der Schreibweise `\sqrt[n]{x}` entspricht `\root n \of{x}`.

### 2.2.6 Hoch- und tiefgestellte Zeichen

Mit `^` und `_` werden hoch- und tiefgestellte Zeichen eingegeben. Beide Sonderzeichen wirken immer auf die nächste Gruppe, und die Reihenfolge, in der sie verwendet werden, ist egal, d. h. `x_1^2` und `x^2_1` erzeugen dieselbe Formel. Eine besondere Rolle spielt in diesem Zusammenhang der Apostroph `'`. Die Schreibweise `f'` ist nämlich eine Abkürzung für `f^\prime` und stellt somit dieselbe Formel dar.

### 2.2.7 Transformation

Diese besonderen Tokens stören den klassischen Shunting-Yard-Algorithmus. Es ist jedoch leicht, den Algorithmus so abzuändern, dass TeX-Befehle wie Operatoren und geschweifte Klammern wie Klammern im klassischen Algorithmus behandelt werden. Wir verwandeln daher alle Befehle u. Ä. in TeX-Befehle. So wird dann etwa `\frac{1}{2}` zu `{1 \frac 2}`. Für hoch- und tiefgestellte Zeichen führen wir den neuen Befehl `#subsup ein`, sodass etwa `x_1^2` zu `x{1 #subsup 2}` wird. Wenn eines der Argumente fehlt, dann ersetzen wir es durch `{}`. Außerdem müssen noch Schreibvarianten wie `'` und `\root` statt `^\prime` und `\sqrt` vereinheitlicht werden.

| Standard-Schreibweise | neue Schreibweise |
|---|---|
| `\frac{1}{2}` | `{ 1 \frac 2 }` |
| `x_1^2` | `x { 1 #subsup 2 }` |
| `y^2` | `y { {} #subsup 2 }` |
| `\sqrt[2]{3}` | `{ 2 sqrt 3 }` |
| `\sqrt{3}` | `{ {} sqrt 3 }` |
| `\root n \of x` | `{ n sqrt x }` |
| `f'` | `f { {} #subsup \prime }` |

Dabei ist natürlich wichtig, dass alle überflüssigen geschweiften Klammern entfernt werden, weil sie sonst den Algorithmus stören. Die Klammern in `1 + {2}` werden etwa nicht benötigt. Andererseits gibt es Fälle, wo Klammern wichtig für die Abstandsregulierung sind, wie man am Unterschied zwischen diesen beiden Formeln sehen kann:

| | |
|---|---|
| `\log^2 x` | $\log^2 x$ |
| `\log {}^2 x` | $\log {}^2 x$ |

Hier verhindern im zweiten Fall die Klammern, dass das hochgestellte Zeichen 2 zu dem vorhergehenden `\log` gehört. Es gibt jedoch auch Fälle, wo geschweifte Klammern die richtigen Abstände erzeugen, die wir hier nicht berücksichtigen können:

```
e \approx 2{,}7      e ≈ 2,7
(2,7)                (2,7)
```

Wir nehmen also an, dass der Abstand nach dem Komma in Aufzählungen (der beim Dezimalkomma fehlen muss!) keinen Einfluss auf die Bedeutung hat, welche aus dem Kontext ersichtlich ist. Die Formeln müssen also beide auch richtig interpretiert werden können, wenn die Abstandsregulierung fehlt.

## 2.3 Neue Schreibweise für Befehle

Auf Grund dieser Beobachtungen können wir jetzt die Formeln in eine neue Darstellungsform überführen, die eindeutig ist und sich leichter verarbeiten lässt. Zunächst kategorisieren wir wieder die Tokens:

`FONT` Befehle, die die Schriftart ändern, z. B. `\mathbf`

`TEXT` Befehle, die mehrere Zeichen zu einem Token zusammenfassen , z. B. `\text`, `\operatorname`

`BLANK` explizite Leerzeichen, z. B. `\quad`

`STYLE` Style-Befehle, z. B. `\displaystyle`

`SIZE` Befehle zur Größenregulierung von Klammern, z. B. `\left`, `\big` usw.

`TEXOP` TeX-Befehle, z. B. `\choose`

`FRAC` Befehle mit zwei Argumenten, z. B. `\frac`

`SQRT` Befehle mit einem Argument und einem optionalen Argument, z. B. `\sqrt`

`ACC` Befehle mit einem Argument, z. B. mathematische Akzente wie `\hat`

`LCUR` linke geschweifte Klammer `{`

`RCUR` rechte geschweifte Klammer `}`

`SUP` Dach `^`

`SUB` Underscore `_`

`NONE` sonstige Tokens

Die Umwandlung wird folgendermaßen durchgeführt: Die Token-Zerlegung der Formel ist im Vektor $v$ abgespeichert, der Vektor $w$ soll dann die transformierte Formel enthalten. Der Vektor $v$ wird von Anfang bis Ende abgearbeitet, $t$ bezeichnet immer das aktuelle Token. Zunächst wird der Typ von $t$ bestimmt. Dann gibt es folgende Möglichkeiten:

`FONT` Bestimme die nächste von geschweiften Klammern eingeschlossene Gruppe nach dem Token und speichere sie im Vektor $a1$ ab. Alle Tokens von $a1$, die Buchstaben sind, werden mit der Information über die Schriftart versehen, aus `x` wird etwa `\mathbf{x}`. Dann wird das Verfahren rekursiv auf $a1$ angewendet, und der Vektor wird anschließend an $w$ angehängt. Es geht dann bei dem nächsten Token nach der Klammergruppe weiter.

`TEXT` Bestimme die nächste von geschweiften Klammern eingeschlossene Gruppe und fasse sie zu einem einzigen String zusammen. Dieser wird an $w$ angehängt; es geht bei dem nächsten Token nach der Klammergruppe weiter.

`SIZE` Das Token wird mit dem nächsten Token zusammengefasst und an $w$ angehängt; es geht beim übernächsten Token weiter.

`ACC` oder `SQRT oder FRAC` In den Vektoren $a1$ und $a2$ werden die Argumente bzw. optionalen Argumente der Befehle gespeichert. Wenn diese nicht vorhanden sind, wird an den jeweiligen Vektor einfach `{}` angehängt. Auf $a1$ und $a2$ wird das Verfahren rekursiv angewendet. Anschließend werden $''\{'', a1, t, a2, ''\}''$. an $w$ angehängt. Dadurch entstehen Ausdrücke der Form `{ 3 \sqrt x }`; es geht weiter bei dem nächsten Token nach dem letzten Argument des Befehls.

`LCUR` Die Klammergruppe, die zu $t$ gehört, wird bestimmt und in $a1$ abgespeichert. Dann wird geprüft, ob es einen TEX-Operator gibt, der zu der Klammer gehört und das Verfahren wird rekursiv auf $a1$ angewendet. Wenn ja, dann werden $''\{'', a1, ''\}''$ an $w$ angehängt, sonst wird nur $a1$ ohne die überflüssigen Klammern an $w$ angehängt.

`SUB` oder `SUP` In $a1$ bzw. $a2$ werden die tief- bzw. hochgestellten Zeichen abgespeichert. Dabei ist auf die richtige Reihenfolge zu achten, weil etwa sowohl `x_i^j` als auch `x^j_i` möglich sind. Hat einer der Vektoren Länge 0, dann wird `{}` angehängt; anschließend wird das Verfahren rekursiv auf $a1$ und $a2$ angewendet. Dann wird $''\{'', a1, \texttt{\#subsup}, a2, ''\}''$ an $w$ angehängt.

`TEXOP` oder `NONE` Das Token wird an $w$ angehängt; es geht weiter beim nächsten Token.

`BLANK` oder `STYLE` Das Token wird nicht an $w$ angehängt; es geht weiter beim nächsten Token.

## 2.4 Zusammenfassung

Die LATEX-Verarbeitung einer Formel erfolgt also in folgenden Schritten:

- Einfache Zerlegung: Der String wird in einen Vektor von Tokens zerlegt.
- Verarbeitung von stilistischen Elementen: Tokens zur Änderung der Schrift, Style-Tokens, `\operatorname` u. Ä. werden verarbeitet.

- Transformation: Überflüssige geschweifte Klammern werden entfernt, Befehle sowie Hoch- und Tiefstellungen werden in die Schreibweise von TeX-Operatoren gebracht, die Schreibvarianten `\root` und `'` werden beseitigt.

Dazu brauchen wir eine Symboltabelle, in der alle benötigten Informationen abgespeichert sind. Auf sie wird dann während der Verarbeitung zugegriffen.

### 2.4.1 Beispiele

Betrachten wir als Beispiel folgende Formeln:

```
\hat f(t) = \int_{\mathbb R^n} f(x)\, e^{-\mathrm{i} t \cdot x} \,\mathrm{d} x

\displaystyle { n \choose k } = {n! \over (n-k)!k!}

\operatorname{erf}(z) =
\frac 2{\sqrt\pi} \int_0^z e^{-\tau^2}\,\mathrm d\tau \quad (z\in\mathbb{C})
```

$$\hat{f}(t) = \int_{\mathbb{R}^n} f(x)\, e^{-\mathrm{i}t\cdot x}\,\mathrm{d}x \qquad \binom{n}{k} = \frac{n!}{(n-k)!k!} \qquad \operatorname{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-\tau^2}\,\mathrm{d}\tau \quad (z \in \mathbb{C})$$

Die einfachen Zerlegungen sehen dann so aus:

| \hat | f | ( | t | ) | = | \int | _ | { | \mathbb | R | ^ | n | } | f | ( | x | ) | \, | e | ^ | { |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | \mathrm | { | i | } | t | \cdot | x | } | \, | \mathrm | { | d | } | x | | | | | | | |

| \displaystyle | { | n | \choose | k | } | = | { | n | ! | \over | ( | n | - | k | ) | ! | k | ! | } |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

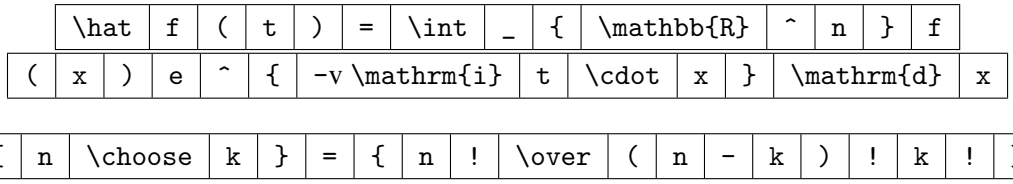
| \operatorname | { | e | r | f | } | ( | z | ) | = | \frac | 2 | { | \sqrt |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| \pi | } | \int | _ | 0 | ^ | z | e | ^ | { | - | \tau | ^ | 2 |
| } | \, | \mathrm | d | \tau | \quad | ( | z | \in | \mathbb | { | C | } | ) |

Zur weiteren Verarbeitung müssen die Typen der Tokens mit besonderen Eigenschaften in einer Symboltabelle abgespeichert sein:

| Token | Typ |
|---|---|
| _ | SUB |
| ^ | SUP |
| { | LCUR |
| } | RCUR |
| \mathrm | FONT |
| \mathbb | FONT |
| \displaystyle | STYLE |
| \choose | TEXOP |

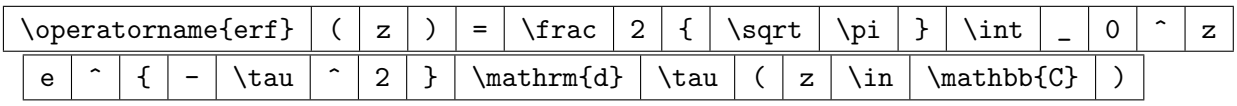
| Token | Typ |
|---|---|
| \over | TEXOP |
| \operatorname | TEXT |
| \hat | ACC |
| \sqrt | SQRT |
| \frac | FRAC |
| \quad | BLANK |
| \, | BLANK |

Im nächsten Verarbeitungsschritt erhalten wir dann

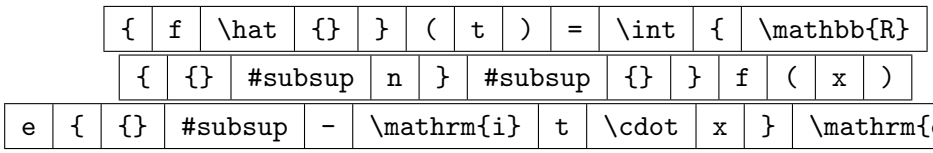


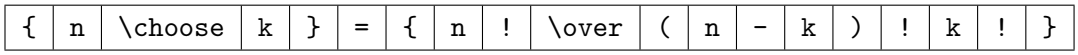


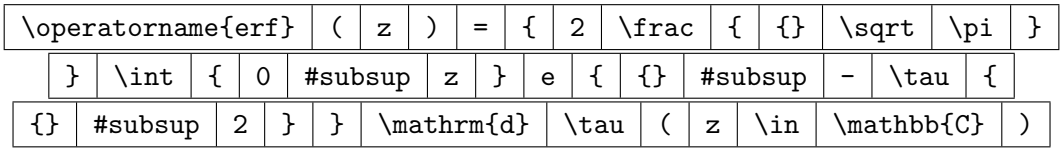





Nach dem letzten Verarbeitungsschritt haben wir die vollständig transformierten Formeln:



# 3 Syntax

## 3.1 Syntaktische Funktionen

Aus den LaTeX-Tokens müssen wir jetzt die Syntax der Formel herleiten und sie als Syntax-Baum darstellen. Es gibt, wie schon erwähnt, sehr viele syntaktische Funktionen, die ein Token in einer Formel haben kann.[1] Durch diese Vielfalt wird eine Verarbeitung stark erschwert. Wir versuchen daher, die Sache so weit wie möglich zu vereinfachen und uns auf möglichst wenige, dafür umso allgemeinere syntaktische Funktionen zu beschränken, aus denen sich dann jede Formel zusammensetzen lässt. Als Motivation kann hierzu Chomskys Minimalistisches Programm[2] aus der Sprachwissenschaft dienen: Dabei werden syntaktische Phänomene auch durch minimalistische Annahmen erklärt und es wird auf kompliziertere Annahmen früherer Modelle verzichtet.

Zunächst einmal kann man zwischen syntaktischen Funktionen in der Ein- und Ausgabe unterscheiden. Ein unärer Operator kann etwa vor oder nach seinem Operanden stehen, je nachdem, ob es ein Infix- oder Postfix-Operator ist. Das hat auf die Semantik jedoch keinerlei Einfluss. Es gibt lediglich eine Konvention, wie der Operator aufgeschrieben wird. Das betrifft also die Ausgabe (d. h. die Regeln, nach denen aus Semantik-Bäumen Formeln erzeugt werden). Uns interessiert jetzt aber die andere Richtung, nämlich die Herleitung von Syntax-Bäumen. Das betrifft also die Eingabe. Für die Eingabe-Syntax ist es etwa egal, ob ein Token, das nach einem anderen Token steht, ein Postfix-Operator oder eine Variable ist. Wesentlich ist nur, dass die Tokens nebeneinander stehen und kein Operator dazwischen steht. Diese Eingabe-Regeln sind für alle Formeln gleich, während die Ausgabe-Regeln natürlich vom Ausgabe-Format abhängen. In Kapitel 5 und 6.2.4 werden wir uns mit zwei Beispielen für solche Ausgabe-Formate beschäftigen, nämlich OpenMath und Concise.

Wir entwickeln hier ein Minimalistisches Programm für die Eingabe-Syntax und erhalten folgende syntaktische Funktionen, die Tokens haben können:

`NAME` Namen: Variable, Funktionennamen, Ziffern usw.

Sie haben keine besondere syntaktische Funktion und verhalten sich wie Variable in den meisten Programmiersprachen.

`OP` Operatoren

[1] Recht detaillierte Beschreibungen mit vielen Beispielen gibt es bei [Ginev 2011] und [Ganesalingam 2009]. Ich verzichte hier auf eine genauere Ausführung und gehe davon aus, dass der Leser / die Leserin mit mathematischen Formeln vertraut ist.

[2] Chomsky, Noam: *A minimalist program for linguistic theory*. MIT occasional papers in linguistics no. 1. Cambridge, MA: Distributed by MIT Working Papers in Linguistics 1993.

Das sind immer binäre Operatoren. Sie verbinden zwei Operanden.

`LBRACK` Linke Klammern

`RBRACK` Rechte Klammern

Mit Klammern können Gruppen zusammengefasst werden. Eine linke und eine rechte Klammer gehören immer zusammen.

`NBRACK` Neutrale Klammern

Sie können sowohl linke als auch rechte Klammern sein (z. B. eckige Klammern bei Intervallen). Deshalb werden sie nicht ineinander verschachtelt.

`VERT` Vertikale Striche

Sie sind wie Klammern, die linke und rechte Klammer sehen aber gleich aus (z. B. Betragsstriche). Wir nehmen hier zur Vereinfachung an, dass sie nicht verschachtelt werden können.[3]

`CMD` Befehle

Das sind die Befehle, die im letzten Kapitel in die Syntax von TEX-Befehlen gebracht wurden (z. B. `\choose`).

`LCUR` linke geschweifte Klammern

`RCUR` rechte geschweifte Klammern

Geschweifte Klammern treten immer paarweise auf und gehören immer zu einem Befehl. Sie dienen zur Gruppierung von Tokens und haben eine ähnliche Funktion wie Klammern im klassischen Shunting-Yard-Algorithmus.

Es fällt auf, dass dabei Postfix-Operatoren, Quantoren o. Ä. fehlt. Wir opfern diese gewissermaßen auf dem Altar des Minimalismus und definieren dafür keine eigenen syntaktischen Funktionen. Stattdessen behandeln wir sie erst später, und zwar auf semantischer Ebene.

### 3.1.1 Klammern

Klammern umschließen immer einen Ausdruck. Sie sind also gewissermaßen Operatoren, die aus zwei Teilen bestehen. Man kann sie weiter in drei Gruppen unterteilen: solche, die eindeutig linke oder rechte Klammer sind, Klammern, die rechts oder links stehen können (aber verschieden aussehen) und Klammern, die links und rechts genau gleich sind.

$$a * (b + c) \qquad ]0,1] \qquad |x|$$

[3] Natürlich können etwa Betragsstriche sehr wohl verschachtelt werden. Das macht die Sache aber dermaßen komplizierter, dass wir diese Möglichkeit hier auslassen.

Wir dürfen Klammern hier jedoch nicht der funktionalen Kategorie zuordnen, weil sie sehr wohl eine Bedeutung haben können und nicht nur zur Gruppierung dienen (z. B. Matrizen oder höhere Ableitungen).

$$f^{(k)}(x) \qquad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

### 3.1.2 Operatoren

Operatoren bilden gewissermaßen das „Gerüst“ der Syntax, indem sie ihre Operanden verbinden. Wir beschäftigen uns hier nur mit binären Infix-Operatoren, d. h. mit Operatoren, die genau zwei Operanden haben, zwischen denen sie stehen. Durch Klammern können längere Ausdrücke zu einem einzigen Operanden zusammengefasst werden. Weiters ist noch besonders auf Rangordnung und Assoziativität zu achten.

**Definition 3.1 (Rangordnung)** *Seien* $\mathrm{op}_1$ *und* $\mathrm{op}_2$ *zwei Operatoren. Dann hat* $\mathrm{op}_1$ *niedere Rangordnung als* $\mathrm{op}_2$*, wenn die Formel* $a\,\mathrm{op}_1\,b\,\mathrm{op}_2\,c$ *die Darstellung* $[\mathrm{op}_1\,a[\mathrm{op}_2\,bc]]$ *hat.*

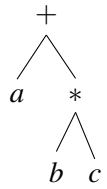



Abbildung 3.1: Der Operator $+$ hat niedere Rangordnung als $*$.

**Definition 3.2 (Assoziativität)** *Sei* op *ein Operator. Wir nennen* op linksassoziativ*, wenn die Formel* $a\,\mathrm{op}\,b\,\mathrm{op}\,c$ *die Darstellung* $[\mathrm{op}[\mathrm{op}\,ab]c]$ *hat, und* rechtsassoziativ*, wenn die Formel* $a\,\mathrm{op}\,b\,\mathrm{op}\,c$ *die Darstellung* $[\mathrm{op}\,a[\mathrm{op}\,bc]]$ *hat.*

*Ist ein Operator linksassoziativ und rechtsassoziativ, dann nennen wir ihn* assoziativ*. Ist der Ausdruck* $a\,\mathrm{op}\,b\,\mathrm{op}\,c$ *nicht definiert, dann ist* op nicht-assoziativ.


Abbildung 3.2: Der Operator $+$ ist linksassoziativ, während $\uparrow$ rechtsassoziativ ist.

In der Praxis lässt sich die Sache jedoch noch etwas vereinfachen. Assoziative Operatoren können nämlich wie linksassoziative behandelt werden („natürliche Assoziativität“). Nicht-assoziative Operatoren dürfen in korrekten Formeln nicht mehrfach hintereinander auftreten, und damit ist es egal, ob sie wie links- oder wie rechtsassoziative Operatoren behandelt werden. Da aber jeder Operator entweder von links oder von rechts verarbeitet werden muss, bleiben dann zwei mögliche Fälle, wobei rechtsassoziative Operatoren eher die Ausnahme sind.

### 3.1.3 Namen

Die Operanden der Operatoren fassen wir einfach als Namen zusammen. Dabei unterscheiden wir nicht zwischen Variablen, Funktionen usw. Der Grund dafür ist der, dass Funktionennamen sich syntaktisch genau wie Variablennamen verhalten, etwa in $(f+g)(x)$ (Addition der Funktionen $f$ und $g$). Die Unterscheidung zwischen Funktionen und Variablen findet also nicht auf syntaktischer, sondern erst auf semantischer Ebene statt. Auch Konstanten und Ziffern verhalten sich syntaktisch wie Variable.

### 3.1.4 Befehle und geschweifte Klammern

Die letzte Klasse von Tokens sind Befehle und die dazugehörigen Klammern. Ein typisches Beispiel dafür ist etwa `{ n \choose k }`. Zur Gruppierung bei den alten TEX-Befehlen können statt geschweifter Klammern auch Klammern mit `\left` und `\right` verwendet werden:

```
\left( a \over b \right)^2 = { a^2 \over b^2 }
```

$$\left(\frac{a}{b}\right)^2 = \frac{a^2}{b^2}$$

Es sei noch einmal daran erinnert, dass wir im vorherigen Kapitel alle Befehle in diese Syntax gebracht haben.

## 3.2 Formeln als formale Sprache

Die Formeln, die sich aus diesen syntaktischen Typen zusammensetzen lassen, können wieder als formale Sprache beschrieben werden. Ihre Grammatik hat folgende Regeln:

$$X \to \texttt{NAME}$$
$$X \to X\ \texttt{OP}\ X$$
$$X \to \texttt{LBRACK}\ X\ \texttt{RBRACK}$$
$$X \to \texttt{NBRACK}\ X\ \texttt{NBRACK}$$
$$X \to \texttt{VERT}\ X\ \texttt{VERT}$$
$$X \to \{\ X\ \texttt{CMD}\ X\ \}$$

Durch diese Regeln lassen sich „ideale Formeln“ darstellen. Das sind etwa $a+b$ oder $2*(3-7)$. Die meisten Formeln, die in der Praxis auftreten, können dadurch jedoch nicht beschrieben werden. In $-1$ fehlt etwa der linke Operand des Operators $-$, in $ab$ oder $\sin(x)$ fehlt jeweils ein Operator zwischen den Operanden und in $\{\}$ müssten die Klammern irgendeinen Ausdruck umschließen. Um diese Probleme zu umgehen, führen wir zwei neue Tokens `{}` und `#link` ein. Überall, wo ein Operator fehlt, wird `#link` eingefügt, und an Stelle eines fehlenden Operanden `{}`. Mit diesem Trick können dann alle Formeln richtig dargestellt werden, ohne dass sich ihr Aussehen ändert. Die Hilfstokens treten nämlich nur in den Bäumen und nicht in den Formeln selbst auf.

| Formel | normale Darstellung | neue Darstellung |
|---|---|---|
| $-1$ | `-1` | `{}-1` |
| $+$ | `+` | `{} + {}` |
| $\{\}$ | `\{ \}` | `\{ {} \}` |
| $ab$ | `ab` | `a #link b` |
| $f(x)$ | `f(x)` | `f #link (x)` |

Diese Hilfstokens müssen also immer dann eingefügt werden, wenn ein Verstoß gegen die oben definierten Regeln vorliegt. In der folgende Tabelle ist zusammengefasst, was eingefügt werden muss, wenn auf einen Typ der ersten Spalte ein Typ der ersten Zeile trifft (# steht für Anfang / Ende einer Formel). Dabei heißt $\times$, dass ein Aufeinandertreffen dieser Typen unmöglich ist.

| | OP | NAME | LBRACK | RBRACK | { | } | CMD | # |
|---|---|---|---|---|---|---|---|---|
| OP | {} | | | {} | | {} | {} | {} |
| NAME | | #link | #link | | #link | | | |
| LBRACK | {} | | | {} | | × | × | × |
| RBRACK | | #link | #link | | #link | | | |
| { | {} | | | × | | | | |
| } | | #link | #link | | #link | | | |
| CMD | {} | | | | | × | × | × |
| # | {} | | | × | | × | × | × |

Die Typen `NBRACK` und `VERT` verhalten sich entweder so wie linke oder wie rechte Klammern, genauer: Wenn man sie in einer Formel von links nach rechts bei 1 beginnend durchnummeriert, sind die mit ungeraden Nummern linke Klammern und die mit geraden rechte.

Wir nehmen `#link` als rechtsassoziativen Operator an, der die höchste Rangordnung hat. Das ist durchaus sinnvoll, wenn man etwa Formeln wie 123, $\sin xy$ oder $\sum_i x_i$ betrachtet. Die unsichtbare Verbindung durch den Operator `#link` erfolgt immer von rechts nach links. Sie bindet auch stärker als jeder andere Operator, etwa in $xy+z$.

## 3.3 Shunting-Yard-Algorithmus

Jetzt können wir eine neue Version des Shunting-Yard-Algorithmus entwerfen. Wir gehen wieder davon aus, dass der Vektor $v$ die Tokens der Formel enthält. Die Formel in Postfix-Notation wird

in einen Ausgabe-Vektor geschrieben, und als Zwischenspeicher wird wieder ein Stack benötigt. Der Vektor $v$ von Anfang bis Ende abgearbeitet, d. h. wir betrachten immer den String $s = v_i$ und beginnen bei $i = 0$. Ist $s$ eine linke geschweifte Klammer, so wird sie an den Stack angehängt. Ist $s$ eine rechte geschweifte Klammer, so wird so lange der letzte Eintrag des Stacks in die Ausgabe geschrieben und anschließend vom Stack entfernt, solange er keine linke geschweifte Klammer ist. Das betrifft die Befehle in TEX-Notation wie `{ n \choose k }`, weil nur bei ihnen geschweifte Klammern verwendet werden. Eine Besonderheit dabei ist `#subsup`, weil tief- bzw. hochgestellte Zeichen immer zu dem vorherigen Token gehören. Deshalb muss in diesem Fall ein zusätzliches `#link` in die Ausgabe geschrieben werden.

Ist das Token $s$ keine geschweifte Klammer, dann wird jetzt sein Typ bestimmt.

- Namen werden in die Ausgabe geschrieben.
- Bei einem Operator müssen Rangordnung und Assoziativität bestimmt werden. Dann wird das letzte Element des Stacks in die Ausgabe geschrieben und vom Stack entfernt, solange es ein Operator ist, der niedere Rangordnung hat als $s$, wenn $s$ linksassoziativ ist, bzw. dessen Rangordnung gleich oder kleiner ist als die von $s$, wenn $s$ rechtsassoziativ ist. Dann wird $s$ an den Stack angehängt.
- Bei Klammern muss bestimmt werden, ob es linke oder rechte Klammern sind. Das ist entweder eindeutig auf Grund der Form der Klammer oder lässt sich an der Reihenfolge der Klammern ablesen (zuerst kommt eine linke Klammer, dann eine rechte, dann wieder eine linke usw.). Eine linke Klammer wird an den Stack angehängt. Bei einer rechten Klammer werden bis zur letzten linken Klammer alle Elemente des Stacks in die Ausgabe geschrieben und vom Stack entfernt. Anschließend werden auch die linke und die dazugehörige rechte Klammer in die Ausgabe geschrieben und das letzte Element wird vom Stack entfernt.
- Wenn $s$ ein Befehl ist, dann wird der letzte Eintrag des Stacks in die Ausgabe geschrieben und vom Stack entfernt, solange es keine linke geschweifte Klammer und auch keine Klammer, die mit `\left` beginnt, ist.

Jetzt müssen ggf. die oben erwähnten Hilfstokens eingefügt werden. Ist $s$ ein Operator, dann wird `{}` an die Ausgabe angehängt, wenn $s$ das erste Element von $v$ ist bzw. wenn das vorherige Element eine linke Klammer, eine linke geschweifte Klammer oder ein Befehl war. Das Token `{}` wird auch dann an die Ausgabe angehängt, wenn $s$ der letzte Eintrag von $v$ ist bzw. wenn der nächste Eintrag ein Operator oder eine rechte Klammer ist. Ist $s$ eine linke Klammer, dann muss `{}` in die Ausgabe geschrieben werden, wenn das nächste Token eine rechte Klammer ist. Wenn $s$ ein Name oder eine rechte (geschweifte) Klammer ist, dann wird `#link` an den Stack angehängt, wenn $s$ nicht am Ende von $v$ steht und der nächste Eintrag kein Operator, kein Befehl und keine rechte (geschweifte) Klammer ist.

Das wird bis zum Ende des Vektors $v$ wiederholt. Dann müssen noch alle Einträge des Stacks in die Ausgabe geschrieben werden, und die Umwandlung ist fertig.

**Algorithmus 3.3.1** Shunting-Yard-Algorithmus

Eingabe: der Vektor $v$ enthält die Tokens Formel.

Ausgabe: der Vektor *out* enthält die Formel in Postfix-Schreibweise.

sonstige Variable: Zahlen $i, n, \texttt{type}, p, \texttt{prec}$, Strings $s, t, b$, Boolesche Variable $nbrack, vert$

**0** Setze $n \leftarrow \texttt{size}(v), i \leftarrow 0, nbrack \leftarrow 0, vert \leftarrow 0$.

**1** Setze $s \leftarrow v_i$.

**2** Falls $s = "\{"$, $\texttt{push}(\texttt{stack}, s)$, sonst weiter bei Schritt 3.

**3** Falls $s \neq "\}"$, weiter bei Schritt 4.

$\texttt{push}(out, \texttt{back}(stack))$ und $\texttt{pop}(stack)$, solange $\texttt{back}(stack) \neq "\{"$ ist.

$\texttt{pop}(stack)$

Falls $\texttt{back}(out) = "\#subsup"$: $\texttt{push}(out, "\#link")$

Falls $\texttt{size}(stack) > 0$ und $\texttt{back}(stack) = "\texttt{\#link}"$: $\texttt{pop}(stack)$

**4** Setze $type \leftarrow \texttt{type}(s)$.

**5** Falls $type \neq \texttt{NAME}$, weiter bei Schritt 6, sonst $\texttt{push}(out, s)$.

**6** Falls $type \neq \texttt{OP}$, weiter bei Schritt 7.

Setze $prec \leftarrow \texttt{precedence}(s)$, $l \leftarrow \texttt{leftass}(s)$, $b \leftarrow \texttt{back}(stack)$

Setze $p \leftarrow \texttt{precedence}(b)$, solange $\texttt{type}(b) = \texttt{OP}$ ist.

Falls $l = 1 \wedge prec > p$ oder $l = 0 \wedge prec \geq p$ weiter bei Schritt 7.

Sonst $\texttt{push}(out, b)$, $\texttt{pop}(stack)$.

**7** Falls $type \neq \texttt{LBRACK}$, weiter bei Schritt 8, sonst $\texttt{push}(stack, s)$.

**8** Falls $type \neq \texttt{RBRACK}$, weiter bei Schritt 9.

Sonst $\texttt{push}(out, \texttt{back}(stack))$, $\texttt{pop}(stack)$, solange $\texttt{type}(\texttt{back}(stack)) \neq \texttt{LBRACK}$.

**9** Falls $type \neq \texttt{NBRACK}$, weiter bei Schritt 10.

Setze $nbrack \leftarrow \neg nbrack$.

Falls $nbrack = 1$, $\texttt{push}(stack, s)$, sonst setze $b \leftarrow \texttt{back}(stack)$, $t \leftarrow \texttt{type}(b)$.

$\texttt{push}(out, b)$ und $\texttt{pop}(stack)$, solange $t \neq \texttt{NBRACK}$.

$\texttt{push}(out, s)$

**10** Falls $type \neq$ `VERT`, weiter bei Schritt 11. Setze $vert \leftarrow \neg vert$.

Falls $vert = 1$, $\texttt{push}(stack, s)$.

Falls $vert = 0$, setze $b \leftarrow \texttt{back}(stack)$, solange $b \neq s$.

$\texttt{push}(out, b)$, $\texttt{pop}(stack)$

$\texttt{push}(out, s)$

**11** Falls $type \neq$ `CMD`, weiter bei Schritt 12.

Setze $b \leftarrow \texttt{back}(stack)$, $\texttt{push}(out, b)$ und $\texttt{pop}(stack)$, solange $b \neq ''\{''$ und $\texttt{size}(b) < 5 \vee b_0 \neq' \backslash' \vee b_0 \neq' l' \vee b_0 \neq' e' \vee b_0 \neq' f' \vee b_0 \neq' t'$.

$\texttt{push}(stack, s)$

**12** Falls $type \neq$ `OP`: weiter bei Schritt 13.

Falls $i = 0$: $\texttt{push}(out, ''\{\}'')$

Falls $i > 0$: setze $b \leftarrow v_{i-1}$ und $t \leftarrow \texttt{type}(b)$, falls $t = \texttt{CMD} \vee t = \texttt{LBRACK} \vee (nbrack = 1 \wedge t = \texttt{NBRACK}) \vee (vert = 1 \wedge t = \texttt{VERT}) \vee b = ''\{''$:

$\texttt{push}(out, ''\{\}'')$

Falls $i \geq n-1$ oder $v_i \neq ''\}''$: $\texttt{push}(out, ''\{\}'')$

Falls $i < n-1$: setze $b \leftarrow v_{i+1}$ und $t \leftarrow \texttt{type}(b)$, falls $t = \texttt{CMD} \vee t = \texttt{OP} \vee t = \texttt{LBRACK} \vee (nbrack = 1 \wedge t = \texttt{NBRACK}) \vee (vert = 1 \wedge t = \texttt{VERT}) \vee b = ''\}''$: $\texttt{push}(out, ''\{\}'')$

**13** Falls $type \neq$ `LBRACK`: weiter bei Schritt 14.

Falls $\texttt{type}(v_{i+1}) = \texttt{RBRACK}$: $\texttt{push}(out, ''\{\}'')$

**14** Falls $type \neq$ `NBRACK`: weiter bei Schritt 15.

Falls $\texttt{type}(v_{i+1}) = \texttt{NBRACK}$: $\texttt{push}(out, ''\{\}'')$

**15** Falls $type \neq$ `VERT`: weiter bei Schritt 16.

Falls $\texttt{type}(v_{i+1}) = \texttt{NBRACK}$: $\texttt{push}(out, ''\{\}'')$

**16** Falls $s \neq ''\{'' \wedge type \neq \texttt{Name} \wedge type \neq \texttt{RBRACK} \wedge (type \neq \texttt{NBRACK} \vee nbrack = 1) \wedge (type \neq \texttt{VERT} \vee vert = 1)$: setze $t \leftarrow \texttt{NONE}$.

Falls $i < n-1$: setze $b \leftarrow v_{i+1}$, $t \leftarrow \texttt{type}(b)$.

Falls $t = \texttt{NAME} \vee t = \texttt{LBRACK} \vee s = ''\{''$: $\texttt{push}(stack, ''\texttt{\#link}'')$.

**17** Falls $i \geq n-1$: weiter bei Schritt 18.

Setze $i \leftarrow i+1$, zurück zu Schritt 1.

**18** Solange $\texttt{size}(stack) > 0$: $\texttt{push}(out, \texttt{back}(stack)), \texttt{pop}(stack)$

### 3.3.1 Benötigte Zusatzinformationen

Damit der Algorithmus funktionieren kann, werden bestimmte Zusatzinformationen zu den einzelnen Tokens benötigt. Sei $T$ die Menge der Tokens und $O \subseteq T$ die Menge der Operatoren. Zunächst muss der Typ jedes Tokens bestimmt werden. Dazu brauchen wir eine Funktion `type`: $T \to \{$`NAME`, `OP`, `LBRACK`, `RBRACK`, `NBRACK`, `VERT`, `CMD`, `LCUR`, `RCUR`$\}$, die jedem Token seinen Typ zuordnet. Für Operatoren brauchen wir auch noch Funktionen `precedence`: $O \to \mathbb{N}$, die jedem Operator seine Rangordnung zuordnet, und `leftass`: $O \to \{0, 1\}$, die angibt, ob der Operator linksassoziativ ist. Diese Funktionen greifen auf globale Symboltabellen zu, in denen diese Informationen abgespeichert sind. Es muss also eine Tabelle geben, in der die Typen aller Tokens abgespeichert sind, die nicht `NAME` als Typ haben. Weiters wird für die Operatoren noch eine Tabelle für die Rangordnungen benötigt und ein Vektor, in dem alle rechtsassoziativen Operatoren abgespeichert sind.

#### Beispiele

Hier sind einige typische Beispiele dafür, wie Formeln in dieser Postfix-Schreibweise aussehen:

| Standard-LaTeX | Postfix |
|---|---|
| `1 + 2` | `1 2 +` |
| `{ n \choose k }` | `n k \choose` |
| `a b` | `a b #link` |
| `( )` | `{} ( )` |
| `x_1^2` | `x 1 2 #subsup #link` |
| `x^2 y` | `x {} 2 #subsup #link y #link` |
| `a b c` | `a b c #link #link` |
| `-1` | `{} 1 -` |
| `\sin x y` | `\sin x y #link #link` |

## 3.4 Syntaxbäume

Aus dieser Postfix-Darstellung sollen nun Syntaxbäume erzeugt werden. Das ist (im Vergleich zur Umwandlung in die Postfix-Darstellung) ziemlich einfach. Wir verwenden nur Binärbäume, d. h. jeder Baum besteht aus einem Knoten sowie einen linken und rechten Unterbaum. Der Vektor, der die Postfix-Darstellung der Formel enthält, wird vom Ende bis zum Anfang abgearbeitet. Das letzte Token wird zum Kopf des Baumes. Jedes Token kann entweder 0, 1 oder 2 Argumente haben, die dann die Unterbäume werden. Dazu wird das Verfahren bei einem Argument rekursiv auf das nächste und bei zwei Argumenten auf die beiden nächsten Tokens angewendet. Hat das Token kein Argument, ist der Baum fertig. So wird der Vektor bis zum Anfang verarbeitet, bis der Baum vollständig ist. Dieses Verfahren ist [Sedgewick 2002] entnommen und in Algorithmus 3.4.1 zusammengefasst. Wir definieren hierfür die Funktion `parse`, die den Algorithmus auf einen Vektor anwendet und so einen Syntaxbaum erzeugt.

**Algorithmus 3.4.1** Umwandlung eines Postfix-Ausdrucks in einen Binärbaum

Eingabe: Vektor $v$, Adresse einer Zahl $i$

Ausgabe: Baum $x$

sonstige Variable: Zahl $t$, String $s$

**0** Setze $i \leftarrow i-1$ und $s \leftarrow v_i$

**1** Falls $s = ''\{\}''$: setzte $x \leftarrow$ `NULL` und der Algorithmus ist fertig.

**2** Setzte $t \leftarrow \texttt{type}(s)$, erzeuge einen neuen Knoten $x$ und $\texttt{setName}(x,s,0)$.

**3** Falls $t = \texttt{OP} \vee t = \texttt{CMD}$: setze $\texttt{right}(x) \leftarrow \texttt{parse}(v,i)$, $\texttt{left}(x) \leftarrow \texttt{parse}(v,i)$.
Falls $t = \texttt{RBRACK} \vee t = \texttt{NBRACK} \vee t = \texttt{VERT}$:
setze $i \leftarrow i-1$ und $\texttt{setName}(x, v_i, 1)$, $\texttt{left}(x) \leftarrow \texttt{parse}(v,i)$.

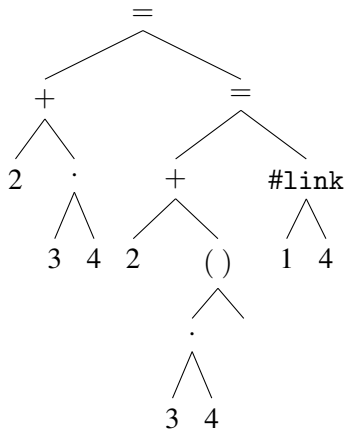
## 3.5 Beispiele

Hier sind einige typische Beispiele für Formeln und die zugehörigen Syntaxbäume. Im Folgenden sind immer die Formel (so wie sie in einem Buch steht), der LaTeX-Code, eine Tabelle mit Informationen zu den Tokens und der Syntaxbaum angegeben. Die Tabelle enthält von jedem Token, das kein Name ist, den Typ, und weiters Rangordnung und Assoziativität der Operatoren. Prinzipiell ist es egal, welche Zahl für die Rangordnung gewählt wird, wichtig ist nur, ob ein Operator höhere oder niedere Rangordnung hat als ein anderer. Hier ist die Wahl so getroffen, dass $+$ und $-$ Rangordnung 1 und $\cdot$ und : Rangordnung 2 haben. Für die übrigen Operatoren wurden Rangordnungen gewählt, die in möglichst vielen Formeln, die auch noch mehrere andere Operatoren enthalten können, gültig sind. Deshalb kommen manchmal auch negative Zahlen vor.

### Beispiel 1

$$2+3\cdot 4 = 2+(3\cdot 4) = 14$$

```
2 + 3 \cdot 4 = 2 + ( 3 \cdot 4 ) = 14
```



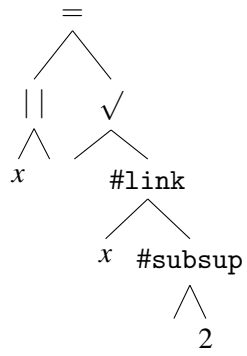

| | | | | |
|---|---|---|---|---|
| $+$ | + | OP | 1 | l |
| $\cdot$ | \cdot | OP | 2 | l |
| $=$ | = | OP | 0 | r |
| ( | ( | LBRACK | | |
| ) | ) | RBRACK | | |

## Beispiel 2

$$|x| = \sqrt{x^2}$$

```
|x| = \sqrt{x^2}
```

| | | | | |
|---|---|---|---|---|
| \| | \| | VERT | | |
| = | = | OP | 0 | r |
| · | \sqrt | CMD | | |
| | #subsup | CMD | | |

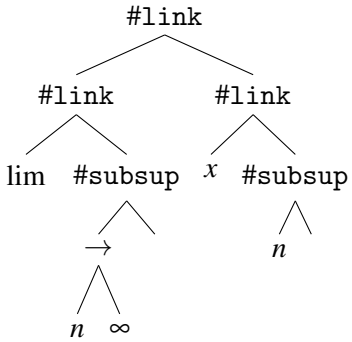



## Beispiel 3

$$\lim_{n \to \infty} x_n$$

```
\lim_{n \to \infty} x_n
```

| | | | | |
|---|---|---|---|---|
| → | \to | OP | 0 | l |
| | #subsup | CMD | | |

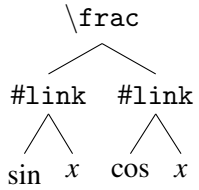



## Beispiel 4

$$\frac{\sin x}{\cos x}$$

```
\frac{\sin x}{\cos x}
```

| | | | | |
|---|---|---|---|---|
| | \frac | CMD | | |
| | #link | OP | 64 | r |

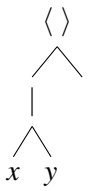



## Beispiel 5

$$\langle x \mid y \rangle$$

```
\left< x \mid y \right>
```

| | | | | |
|---|---|---|---|---|
| ⟨ | \left< | LBRACK | | |
| ⟩ | \right> | RBRACK | | |
| | \mid | OP | −4 | l |

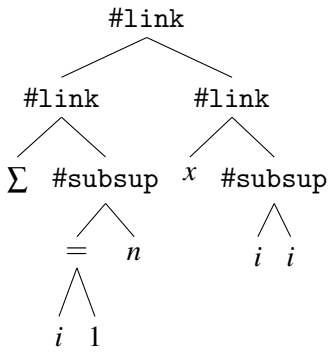


## Beispiel 6

$$\sum_{i=1}^{n} x_i^i$$

```
\sum_{i=1}^n x_i^i
```

| | | | | |
|---|---|---|---|---|
| | #link | OP | 64 | r |
| | #subsup | CMD | | |
| = | = | OP | 0 | r |

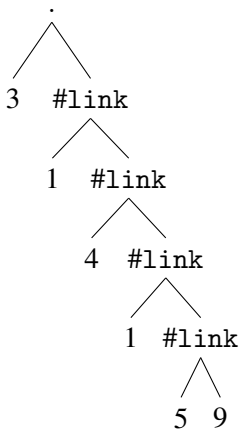



## Beispiel 7

3.14159

```
3.14159
```

| | | | | |
|---|---|---|---|---|
| . | . | OP | 8 | l |

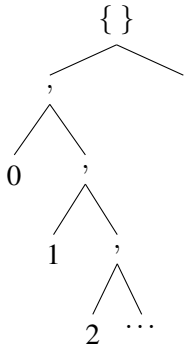



## Beispiel 8

$$\{0,1,2,\dots\}$$

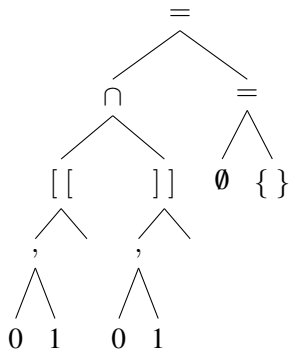
```
\{0,1,2, \dots\}
```

| | | | | |
|---|---|---|---|---|
| { | \{ | LBRACK | | |
| } | \} | RBRACK | | |
| , | , | OP | −1 | r |



## Beispiel 9

$$[0,1[\cap]0,1] = \emptyset = \{\}$$

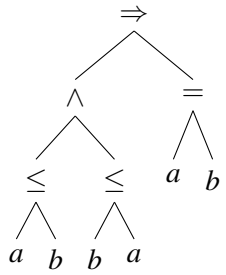



```
[0,1[ {} \cap {} ]0,1] = \emptyset = \{\}
```

| | | | | |
|---|---|---|---|---|
| { | \{ | LBRACK | | |
| } | \} | RBRACK | | |
| [ | [ | NBRACK | | |
| ] | ] | NBRACK | | |
| ∩ | \cap | OP | 1 | l |
| , | , | OP | −1 | r |

## Beispiel 10

$$a \leq b \wedge b \leq a \Rightarrow a = b$$



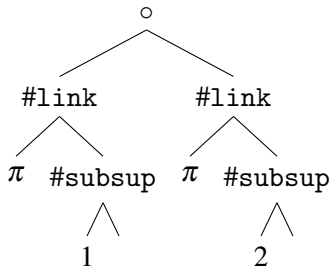

```
a \leq b \land b \leq a \Rightarrow a = b
```

| | | | | |
|---|---|---|---|---|
| ≤ | \leq | OP | 0 | r |
| ∧ | \land | OP | −3 | l |
| ⇒ | \Rightarrow | OP | −8 | l |
| = | = | OP | 0 | r |

## Beispiel 11

$$\pi_1 \circ \pi_2$$

```
\pi_1 \circ \pi_2
```

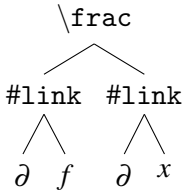



| | | | | |
|---|---|---|---|---|
| ∘ | \circ | OP | 1 | l |
| | #subsup | CMD | | |

### Beispiel 12

$$\frac{\partial f}{\partial x}$$

```
\frac{\partial f}{\partial x}
```

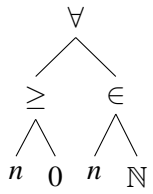



| | | | | |
|---|---|---|---|---|
| | \frac | CMD | | |
| | #link | OP | 64 | r |

### Beispiel 13

$$n \geq 0 \quad \forall n \in \mathbb{N}$$

```
n \geq 0 \quad \forall n \in \mathbb{N}
```

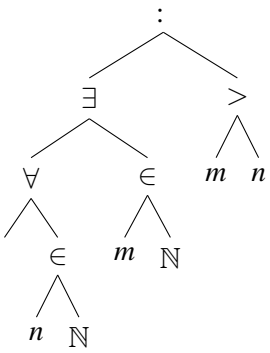



| | | | | |
|---|---|---|---|---|
| $\geq$ | \geq | OP | | r |
| $\forall$ | \forall | OP | $-6$ | l |
| $\in$ | \in | OP | $-2$ | l |

### Beispiel 14

$$\forall n \in \mathbb{N} \exists m \in \mathbb{N}\colon m > n$$

```
\forall n \in \mathbb{N}
\exists m \in \mathbb{N}: m > n
```

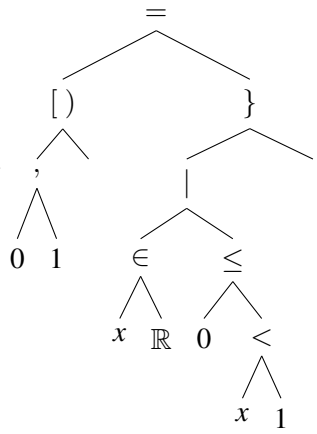



| | | | | |
|---|---|---|---|---|
| $>$ | > | OP | 0 | r |
| $\in$ | \in | OP | $-2$ | l |
| $\forall$ | \forall | OP | $-6$ | l |
| $\exists$ | \exists | OP | $-7$ | l |
| : | \colon | OP | $-8$ | l |

## Beispiel 15

$$[0,1) = \{x \in \mathbb{R} \mid 0 \leq x < 1\}$$

```
[0,1) = \{ x \in \mathbb{R} \mid 0 \leq x < 1 \}
```

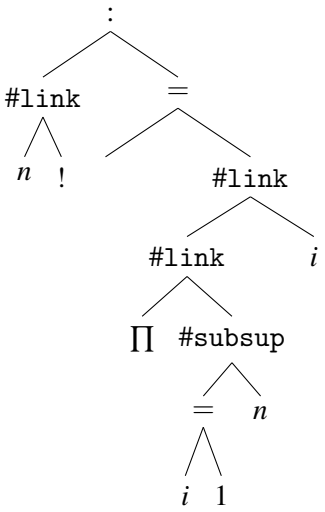



| | | | | |
|---|---|---|---|---|
| [ | [ | NBRACK | | |
| ) | ) | NBRACK | | |
| , | , | OP | −1 | r |
| = | = | OP | 0 | r |
| < | < | OP | 0 | r |
| ∈ | \in | OP | −2 | l |
| \| | \mid | OP | −4 | l |
| { | \{ | LBRACK | | |
| } | \} | RBRACK | | |

## Beispiel 16

$$n! := \prod_{i=1}^{n} i$$

```
n! := \prod_{i=1}^n i
```

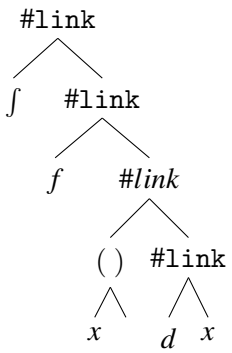



| | | | | |
|---|---|---|---|---|
| | #link | OP | 64 | r |
| : | : | OP | −1 | l |
| = | = | OP | 0 | r |
| | #subsup | CMD | | |

## Beispiel 17

$$\int f(x)\,dx$$

```
\int f(x)\,dx
```



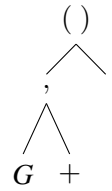

| | | | | |
|---|---|---|---|---|
| | #link | OP | 64 | r |

### Beispiel 18

$$(G,+)$$

```
(G,+)
```

| | | | | |
|---|---|---|---|---|
| ( | ( | LBRACK | | |
| ) | ) | RBRACK | | |
| , | , | OP | −1 | r |
| + | + | OP | 1 | l |

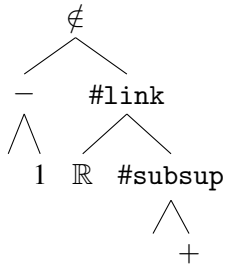



### Beispiel 19

$$-1 \notin \mathbb{R}^+$$

```
-1 \notin \mathbb R^+
```

| | | | | |
|---|---|---|---|---|
| + | + | OP | 1 | l |
| − | - | OP | 1 | l |
| ∉ | \notin | OP | −1 | l |
| | #link | OP | 64 | r |
| | #subsup | CMD | | |

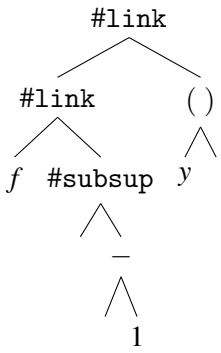



### Beispiel 20

$$f^{-1}(y)$$

```
f^{-1}(y)
```

| | | | | |
|---|---|---|---|---|
| | #link | OP | 64 | r |
| | #subsup | CMD | | |
| − | - | OP | 1 | l |
| ( | ( | LBRACK | | |
| ) | ) | RBRACK | | |

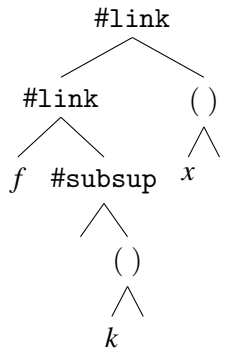


## Beispiel 21

$$f^{(k)}(x)$$

```
f^{(k)}(x)
```

| | | | | |
|---|---|---|---|---|
| | #link | OP | 64 | r |
| | #subsup | CMD | | |
| ( | ( | LBRACK | | |
| ) | ) | RBRACK | | |

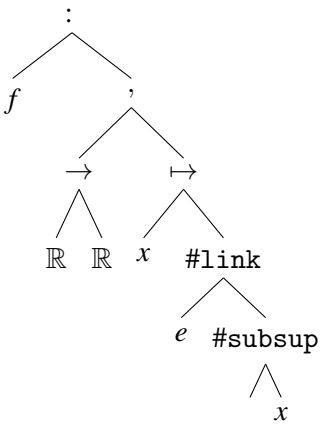



## Beispiel 22

$$f\colon \mathbb{R} \to \mathbb{R}, x \mapsto e^x$$

```
f\colon \mathbb{R} \to \mathbb{R}, x \mapsto e^x
```

| | | | | |
|---|---|---|---|---|
| : | \colon | OP | −8 | r |
| → | \to | OP | 0 | l |
| , | , | OP | −1 | r |
| ↦ | \mapsto | OP | 0 | l |
| | #link | OP | 64 | r |
| | #subsup | CMD | | |




Im letzten Beispiel wäre es auch möglich, dass der Doppelpunkt höhere Rangordnung als das Komma hat.

# 4 Semantik

## 4.1 Semantische Typen und Kategorien

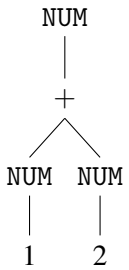
### 4.1.1 Semantische Regeln

Wie wir schon in der Einleitung gesehen haben, beschreibt jede Formel ein bestimmtes mathematisches Objekt. Diese Objekte lassen sich in semantische Typen einteilen, z. B. Zahlen, Vektoren, Matrizen usw. Dementsprechend hat auch jeder Binärbaum, der eine Formel darstellt, einen bestimmten Typ. Jeder Baum lässt sich also durch die Ableitungsregel einer formalen Grammatik beschreiben:

$$t_h \rightarrow n\ t_l\ t_r$$

Dabei ist $t_h$ der Typ des Baumes, $t_l, t_r$ sind die Typen der Unterbäume und $n$ ist der Vektor, der die Tokens des Baumes enthält. Als Beispiel ist hier die Formel $1+2$ als Baum und als äquivalenter Klammerausdruck mit Typen dargestellt:

$$[+[1]_{\texttt{NUM}}[2]_{\texttt{NUM}}]_{\texttt{NUM}}$$




Dieser Darstellung liegen die folgenden Regeln zugrunde:

$$\texttt{NUM} \rightarrow 1$$
$$\texttt{NUM} \rightarrow 2$$
$$\texttt{NUM} \rightarrow + \texttt{ NUM NUM}$$

**Definition 4.1 (Semantische Kategorie)** *Eine* semantische Kategorie *ist ein Quadrupel* $(n, t_h, t_l, t_r)$*, wobei n ein Vektor von Strings ist und* $t_h, t_l, t_r$ *Strings sind. Zu jedem korrekten Semantikbaum gibt es eine semantische Kategorie, sodass* $t_h$ *der Typ des Baumes ist,* $t_l$ *der Typ des linken Unterbaumes,* $t_r$ *der Typ des rechten Unterbaumes und n die Tokens des Baumes enthält.*

Die drei oben erwähnten Regeln sind also die Kategorien $(+, \texttt{NUM}, \texttt{NUM}, \texttt{NUM})$, $(1, \texttt{NUM}, \texttt{NONE}, \texttt{NONE})$ und $(2, \texttt{NUM}, \texttt{NONE}, \texttt{NONE})$.

Wir nehmen im Weiteren an, dass alle Kategorien in einem Vektor `rules` abgespeichert sind, auf den global zugegriffen werden kann. Wenn dieser Vektor nur die drei obigen Regeln enthält, würde er so aussehen:

| | | | |
|---|---|---|---|
| $(\texttt{rules}_0)_n = ''1''$ | $(\texttt{rules}_0)_{t_h} = \texttt{NUM}$ | $(\texttt{rules}_0)_{t_l} = \texttt{NONE}$ | $(\texttt{rules}_0)_{t_r} = \texttt{NONE}$ |
| $(\texttt{rules}_1)_n = ''2''$ | $(\texttt{rules}_1)_{t_h} = \texttt{NUM}$ | $(\texttt{rules}_1)_{t_l} = \texttt{NUM}$ | $(\texttt{rules}_1)_{t_r} = \texttt{NUM}$ |
| $(\texttt{rules}_2)_n = ''+''$ | $(\texttt{rules}_2)_{t_h} = \texttt{NUM}$ | $(\texttt{rules}_2)_{t_l} = \texttt{NUM}$ | $(\texttt{rules}_2)_{t_r} = \texttt{NUM}$ |

Es ist zu beachten, dass durch die Festlegung der Kategorien entschieden wird, welche Formeln semantisch korrekt sind und welche nicht. Werden etwa die Kategorien $(/, \texttt{NUM}, \texttt{NUM}, \texttt{NUM})$, $(1, \texttt{NUM}, \texttt{NONE}, \texttt{NONE})$ und $(0, \texttt{NUM}, \texttt{NONE}, \texttt{NONE})$ definiert, dann ist $1/0$ eine semantisch korrekte Formel vom Typ `NUM`. Das ist hier insofern unproblematisch, weil wir ja voraussetzen, dass alle Formeln, die wir verarbeiten, korrekt sind (wodurch $1/0$ ausgeschlossen ist). Sollte jedoch irgendwo eine Textstelle wie „der Ausdruck $1/0$ ist nicht definiert“ auftreten, dann müssen die Kategorien so gewählt sein, dass die Formel nicht den Typ `NUM` hat. Es kann also auch vorkommen, dass Kategorien semantisch falsche Formeln beschreiben. Solche Kategorien dürfen nur dann verwendet werden, wenn ein Auftreten solcher Formeln ausgeschlossen werden kann.

### 4.1.2 Typbestimmung

Wenn wir die Tokens und die Typen der Unterbäume eines korrekten Semantikbaumes kennen, dann gibt es eine Kategorie, die uns den Typ des Baumes selbst verrät. Die Typbestimmung erfolgt mittels Post-order-Traversal des Baumes: Zuerst werden die Typen der Unterbäume bestimmt. Der Typ eines fehlenden Baumes ist immer `NONE`. Dann wird eine Kategorie gesucht, die die Tokens des Baumes und die Typen der Unterbäume enthält. Sie verrät uns den Typ des Baumes. Das Verfahren wird in Algorithmus 4.1.1 beschrieben. Wir benutzen dazu die Funktion `rulecheck`, die den Algorithmus auf einen Baum anwendet.

Problematisch wird es, wenn es mehrere passende Kategorien gibt. Dann müssen alle möglichen Typen in einem Vektor gespeichert werden – die Auswahl erfolgt später. Ein Beispiel dafür wäre etwa $(0,1)$, weil damit sowohl ein offenes Intervall als auch ein Punkt im $\mathbb{R}^2$ gemeint sein könnte. Wir gehen hier der Einfachheit halber jedoch davon aus, dass die Typen immer eindeutig sind.

**Algorithmus 4.1.1** Typbestimmung

Eingabe: der Baum $n$ enthält die Formel ohne Typen.

Ausgabe: im Baum $n$ ist in jedem Knoten ein Typ eingetragen.

Sonstige Variable: Bäume $l, r$, Strings $\texttt{ltype}, \texttt{rtype}$, ein String-Vektor $\texttt{name}$, Zahl $i$

**0** Setze $l \leftarrow \texttt{left}(n), r \leftarrow \texttt{right}(n), \texttt{type} \leftarrow \texttt{getType}(n,0), \texttt{name} \leftarrow \texttt{getName}(n)$.

**1** Wende den Algorithmus auf die Unterbäume an: $l \leftarrow \texttt{rulecheck}(l)$, $r \leftarrow \texttt{rulecheck}(r)$.

**2** Falls $l \neq \texttt{NULL}$, setze $\texttt{ltype} \leftarrow \texttt{getType}(l,0)$, sonst $\texttt{ltype} \leftarrow \texttt{NONE}$;
falls $r \neq \texttt{NULL}$, setze $\texttt{rtype} \leftarrow \texttt{getType}(r,0)$, sonst $\texttt{rtype} \leftarrow \texttt{NONE}$.

**3** Setze $i \leftarrow 0$.

**4** Falls $(\texttt{rules}_i)_n = \texttt{name} \wedge (\texttt{rules}_i)_{t_l} = \texttt{ltype} \wedge (\texttt{rules}_i)_{t_r} = \texttt{rtype}$,
setze $\texttt{setType}(n, (\texttt{rules}_i)_{t_h}, 0)$ und der Algorithmus ist fertig.

**5** Setze $i \leftarrow i+1$ und wiederhole Schritt 4, falls $i < \texttt{size}(\mathit{rules})$ ist.

### 4.1.3 Beispiele

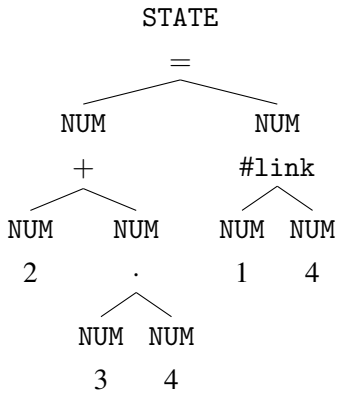
Die Wahl der Typen hängt sehr stark vom Kontext und vom Inhalt der Formel ab. Aus diesem Grund sind die hier vorgestellten Beispiele jeweils nur eine von vielen Möglichkeiten, die Formeln zu interpretieren. Neben den Formeln und den Bäumen mit Typen enthält jedes Beispiel auch noch eine Tabelle mit den Regeln für die Typbestimmung. Sie besteht aus den Tokens, dem Typ des Knotens sowie den Typen des linken und rechten Unterbaumes.

#### Beispiel 1

$$2+3\cdot 4 = 14$$

```
2 + 3 \cdot 4 = 14
```

| | | | |
|---|---|---|---|
| 1 | NUM | NONE | NONE |
| 2 | NUM | NONE | NONE |
| 3 | NUM | NONE | NONE |
| 4 | NUM | NONE | NONE |
| + | NUM | NUM | NUM |
| \cdot | NUM | NUM | NUM |
| #link | NUM | NUM | NUM |
| = | STATE | NUM | NUM |

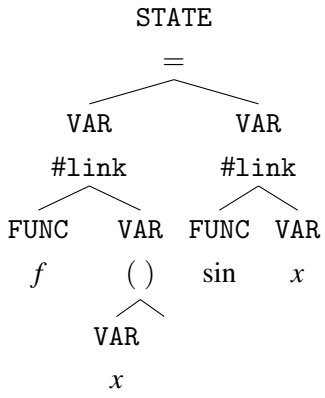


### Beispiel 2

$$f(x) = \sin x$$

```
f(x) = \sin x
```

| | | | |
|---|---|---|---|
| x | VAR | NONE | NONE |
| f | FUNC | NONE | NONE |
| \sin | FUNC | NONE | NONE |
| ( ) | VAR | VAR | NONE |
| #link | VAR | FUNC | VAR |
| = | STATE | VAR | VAR |

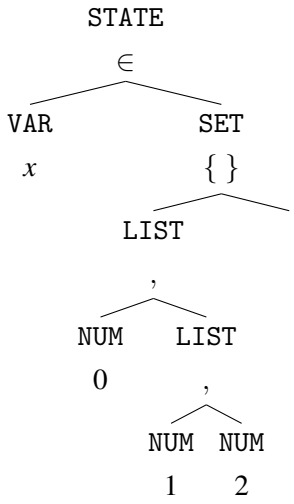



### Beispiel 3

$$x \in \{0,1,2\}$$

```
x \in \{ 0, 1, 2 \}
```

| | | | |
|---|---|---|---|
| 0 | NUM | NONE | NONE |
| 1 | NUM | NONE | NONE |
| 2 | NUM | NONE | NONE |
| \{ \} | SET | LIST | NONE |
| , | LIST | NUM | NUM |
| , | LIST | NUM | LIST |

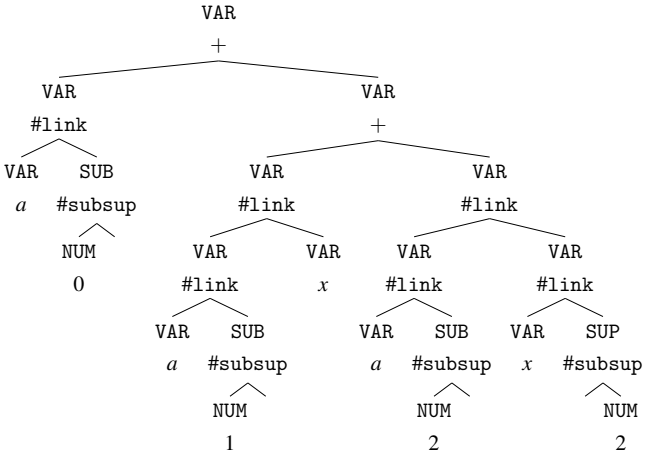


**Beispiel 4**

$$a_0 + a_1 x + a_2 x^2$$

```
a_0 + a_1 x + a_0 + a_1 x
```

| | | | |
|---|---|---|---|
| 0 | NUM | NONE | NONE |
| 1 | NUM | NONE | NONE |
| 2 | NUM | NONE | NONE |
| a | VAR | NONE | NONE |
| x | VAR | NONE | NONE |
| #subsup | SUB | NUM | NONE |
| #subsup | SUP | NONE | NUM |
| #link | VAR | VAR | SUB |
| #link | VAR | VAR | SUP |
| #link | VAR | VAR | VAR |
| + | VAR | VAR | VAR |

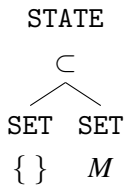



**Beispiel 5**

$$\{\} \subset M$$

```
\{ \} \subset M
```

| | | | |
|---|---|---|---|
| M | SET | NONE | NONE |
| \{ \} | SET | NONE | NONE |
| \subset | STATE | SET | SET |

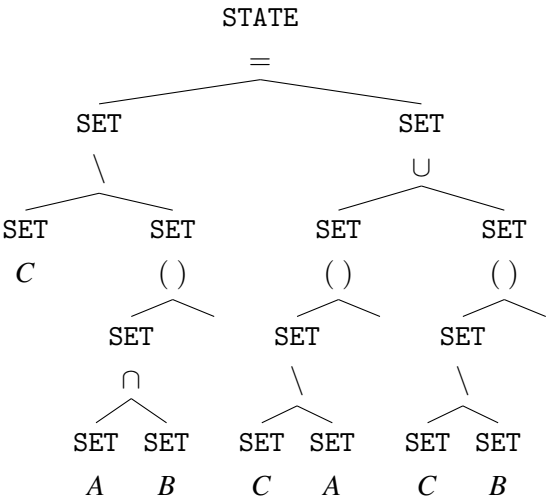


**Beispiel 6**

$$C \setminus (A \cap B) = (C \setminus A) \cup (C \setminus B)$$

```
C \setminus ( A \cap B ) =
  ( C \setminus A ) \cup
    ( C \setminus B )
```

| | | | |
|---|---|---|---|
| A | SET | NONE | NONE |
| B | SET | NONE | NONE |
| C | SET | NONE | NONE |
| ( ) | SET | SET | NONE |
| \cap | SET | SET | SET |
| \cup | SET | SET | SET |
| \setminus | SET | SET | SET |
| = | STATE | SET | SET |

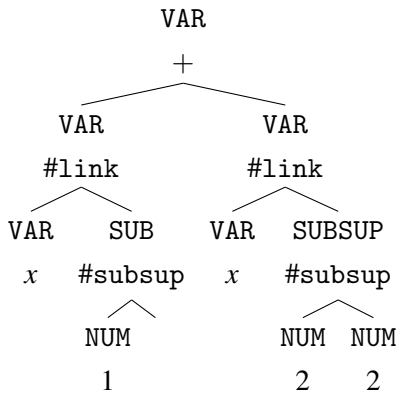



**Beispiel 7**

$$x_1 + x_2^2$$

```
x_1 + x_2^2
```

| | | | |
|---|---|---|---|
| 1 | NUM | NONE | NONE |
| 2 | NUM | NONE | NONE |
| x | VAR | NONE | NONE |
| + | VAR | VAR | VAR |
| #subsup | SUB | NUM | NONE |
| #subsup | SUBSUP | NUM | NUM |
| #link | VAR | VAR | SUB |
| #link | VAR | VAR | SUBSUP |

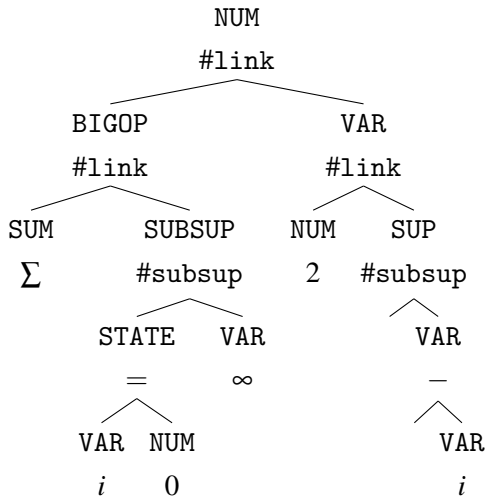


**Beispiel 8**

$$\sum_{i=0}^{\infty} 2^{-i}$$

```
\sum_{i=0}^\infty 2^{-i}
```

| 0 | NUM | NONE | NONE |
|---|---|---|---|
| 2 | NUM | NONE | NONE |
| i | VAR | NONE | NONE |
| \infty | VAR | NONE | NONE |
| \sum | SUM | NONE | NONE |
| = | STATE | VAR | NUM |
| - | VAR | NONE | VAR |
| #subsup | SUP | NONE | VAR |
| #subsup | SUBSUP | STATE | VAR |
| #link | VAR | NUM | SUP |
| #link | BIGOP | SUM | SUBSUP |
| #link | NUM | BIGOP | NUM |

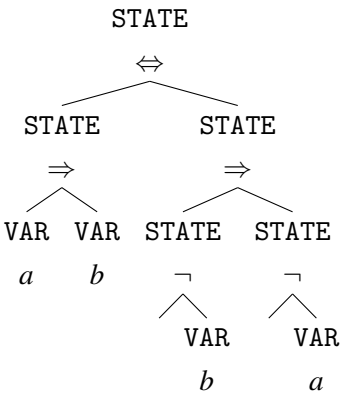



**Beispiel 9**

$$a \Rightarrow b \Longleftrightarrow \neg b \Rightarrow \neg a$$

```
a \Rightarrow b \iff
\neg b \Rightarrow \neg a
```

| a | VAR | NONE | NONE |
|---|---|---|---|
| b | VAR | NONE | NONE |
| \neq | STATE | NONE | VAR |
| \Rightarrow | STATE | STATE | STATE |
| \Rightarrow | STATE | VAR | VAR |
| \iff | STATE | STATE | STATE |




## 4.2 Nachbearbeitung von Syntax-Bäumen

### 4.2.1 Falsche Assoziativität

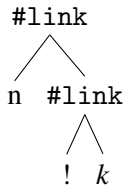
Ein Problem kann bei der Assoziativität des von uns eingeführten `#link`-Operators auftreten: Wir haben angenommen, dass die unsichtbare Verbindung zweier Tokens immer rechtsassoziativ ist. In

den meisten Fällen ist das auch sinnvoll und liefert die richtigen Ergebnisse, wie z. B. bei diesen Formeln:

$$\sin xy \qquad \int f(x)\,dx \qquad 128$$

Da für LaTeX jede Ziffer ein eigenes Token ist, sind Zahlen einfach Bäume, in denen die Ziffern durch `#link`-Operatoren verbunden sind. Die Rechtsassoziativität ist deshalb sinnvoll, weil Zahlen eigentlich von rechts nach links geschrieben werden (die Einerstelle steht ganz rechts). Probleme, die dabei trotzdem auftreten können, werden in 4.2.2 behandelt.

Leider gibt es aber auch Beispiele, wo das nicht funktioniert, wie etwa bei Postfix-Operatoren (die wir der Einfachheit halber nicht als eigene syntaktische Kategorie behandeln, siehe 3.1). Die Formel $n!k$ wird durch die Verarbeitung, die im letzten Kapitel beschrieben wurde, in die Baumdarstelllung




gebracht. In Wirklichkeit gehört der Operator ! aber immer zu dem Operand, der vor ihm steht, und der korrekte Syntaxbaum sollte daher so aussehen:


Wir müssen daher manchmal die Assoziativität von `#link` nachträglich ändern. Das ist genau dann der Fall, wenn der Baum folgende Struktur hat: Im Knoten des Baumes sowie seines rechten Unterbaumes steht das Token `#link` und der linke Unterbaum des rechten Unterbaumes hat hat den entsprechenden Typ. Bevor der Baum umgewandelt werden kann, muss also die Typbestimmung durchgeführt werden. Anschließend kann bei allen Typen, bei denen `#link` linksassoziativ sein muss, die Struktur des Baumes geändert werden. Wenn wir in einem Binärbaum $n$ die Assoziativität von `#link` bei Knoten mit dem Typ $s$ ändern wollen, setzen wir $l \leftarrow \texttt{left}(n), r \leftarrow \texttt{right}(n)$ und wenden das Verfahren zuerst auf die beiden Unterbäume an. Wenn nicht $\texttt{getName}(n,0) = \text{''}\texttt{\#link}\text{''}, r \neq \texttt{NULL}$ und $\texttt{getName}(r,0) = \text{''}\texttt{\#link}\text{''}$ ist, sind wir fertig, weil der Baum dann keine für uns relevante Struktur hat. Sonst betrachten wir den rechten Unterbaum und setzen $\texttt{t1} \leftarrow \texttt{left}(r), \texttt{t2} \leftarrow \texttt{right}(r)$. Wir untersuchen jetzt die Struktur von $r$: es muss $\texttt{t1} \neq \texttt{NULL}, \texttt{t2} \neq \texttt{NULL}$ und $\texttt{getType}(\texttt{t1},0) = s$ gelten. Ist dies nicht der Fall, sind wir fertig. Ansonsten setzen wir den Baum neu zusammen: die neuen Unterbäume werden $L$ und $R$. Wir setzen $R \leftarrow \texttt{t2}$ und erzeugen einen neuen Baum $L$ mit $\text{''}\texttt{\#link}\text{''}$ im Knoten und $l$ und `t1` als linken bzw. rechten Unterbaum. Dann können wir $\texttt{left}(n) \leftarrow L, \texttt{right}(n) \leftarrow R$ setzen und haben den neuen Baum. Für dieses Verfahren definieren wir die Funktion `change_ass`, die das Verfahren auf einen Baum und einen Typen (der als String vorliegt) anwendet.

**Algorithmus 4.2.1** Änderung der Assoziativität des Operators `#link`

Eingabe: im Baum $n$ soll die Assoziativität von `#link` beim Typ $s$ geändert werden.

Ausgabe: Baum $n$

Sonstige Variable: Bäume $\texttt{t1}, \texttt{t2}, l, r, L, R$

**0** Setze $l \leftarrow \texttt{left}(n), r \leftarrow \texttt{right}(n)$.

**1** Setze $l \leftarrow \texttt{change\_ass}(n,s), r \leftarrow \texttt{change\_ass}(n,s)$.

**2** Falls $\texttt{getName}(n,0) \neq ''\texttt{\#link}'' \vee r = \texttt{NULL} \vee \texttt{getName}(r,0) \neq ''\texttt{\#link}''$: der Algoritmus ist fertig.

**3** Setze $\texttt{t1} \leftarrow \texttt{left}(r), \texttt{t2} \leftarrow \texttt{right}(r)$.

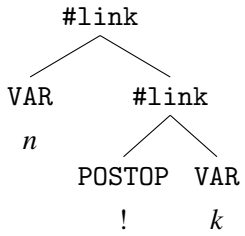
**4** Falls $\texttt{t1} = 0 \vee \texttt{t2} = \texttt{NULL} \vee \texttt{getType}(\texttt{t1},0) \neq s$: der Algoritmus ist fertig.

**5** Setzte $R \leftarrow \texttt{t2}$, $\texttt{setName}(L, ''\texttt{\#link}'', i)$, setze $\texttt{left}(L) \leftarrow l, \texttt{right}(L) \leftarrow \texttt{t1}$.
Setze $\texttt{left}(n) \leftarrow L, \texttt{right}(n) \leftarrow R$

**Beispiel** Betrachten wir noch einmal die Formel $n!k$ – wie ihr Syntaxbaum aussieht, den unser Algorithmus erzeugt, haben wir schon oben gesehen. Wir definieren folgende semantische Kategorien:

$$\texttt{VAR} \rightarrow n$$
$$\texttt{VAR} \rightarrow k$$
$$\texttt{POSTOP} \rightarrow !$$

Nach der Typbestimmung sieht der Baum dann so aus:




Wenn wir jetzt den Algorithmus auf den Baum und den Typen `POSTOP` anwenden, erhalten wir den gewünschten Baum mit der korrekten Darstellung.

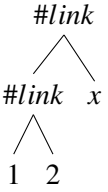
### 4.2.2 Zahlen

Wir haben auch für Ziffern, mit denen Zahlen dargestellt werden, keinen eigenen syntaktischen Typen definiert, sondern sie als Namen behandelt. Dadurch können wir nachträglich beliebige Zeichen als Ziffern deklarieren, was für die Zifferndarstellung zu anderen Basen als 10 sehr nützlich ist:

$$(255)_{10} = (\mathrm{FF})_{16}$$

Die Zifferndarstellung einer Zahl ist also einfach eine Folge von Tokens des Typs `NAME` und unterscheidet sich syntaktisch nicht von einer Multiplikation, bei der das Multiplikationszeichen nicht angeschrieben wird. Das kann zu Problemen führen, wenn mehrstellige Zahlen mit einer Variablen multipliziert werden. Die Formel $12x$ hat etwa folgende Baumdarstellung:


In der richtigen Darstellung sollte jedoch die Rangordnung von `#link` zwischen Ziffern höher sein als sonst:




Wir brauchen also einen Algorithmus, der in einem `#link`-Baum die Zifferngruppen zusammenfasst und so nachträglich die Rangordnung von `#link` an bestimmten Stellen ändert. Wir betrachten den Baum $n$, der die spezielle Struktur hat, dass `#link` der einzige Operator ist. Nach der Typbestimmung haben alle Knoten, die eine Ziffer enthalten, den Typ $t$. Mit `p1` bezeichnen wir den obersten `#link`-Knoten im Baum, dessen linker Unterbaum nicht von Typ $t$ ist. Weiters ist `p2` der Eltern-Knoten von `p2` und `p3` der von `p2`. Wenn in dem Baum $n$ keine einzige Ziffer vorkommt, ist $\texttt{p1} = \texttt{NULL}$. Besteht der Baum jedoch nur aus Ziffern (die durch `#link` verbunden sind), so ist $\texttt{type}(p1) = t$. In beiden Fällen muss der Baum nicht verändert werden. Sonst erzeugen wir einen neuen Baum, in dessen Knoten der Operator `#link` steht. Wenn $\texttt{p3} \neq \texttt{NULL}$ ist, wird der linke Unterbaum der ursprüngliche Baum, in dem $\texttt{right}(\texttt{p3})$ durch $\texttt{left}(\texttt{p2})$ ersetzt worden ist. Den rechten Unterbaum erhalten wir, indem wir das Verfahren auf $r$ anwenden. Ist $\texttt{p3} = \texttt{NULL}$ und $\texttt{p2} \neq \texttt{NULL}$, dann wird der linke Unterbaum $\texttt{left}(\texttt{p2})$ und der rechte Unterbaum ist wie im vorherigen Fall. Falls $\texttt{p3} = \texttt{NULL}$ und $\texttt{p2} = \texttt{NULL}$, dann ist der linke Unterbaum $\texttt{left}(\texttt{p1})$ und der rechte Unterbaum entsteht durch Anwenden des Verfahrens auf $\texttt{right}(\texttt{p1})$. Die drei typischen Fälle, die hierbei auftreten können, sind in Abbildung 4.1 dargestellt. Das Verfahren ist in Algorithmus 4.2.2 beschrieben und wird von der Funktion `change_prec` auf einen `#link`-Baum und einen Typ `t` angewendet.

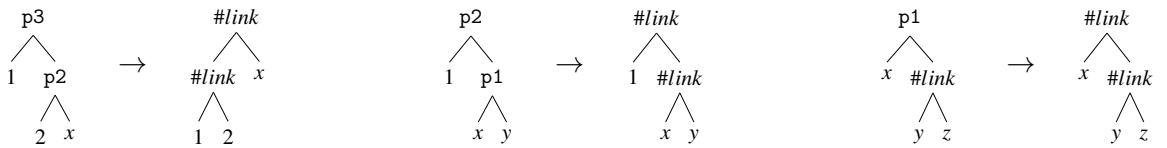



Abbildung 4.1: Umgruppierung von Ziffern

**Algorithmus 4.2.2** Gruppierung von Ziffern

Eingabe: im Baum $n$ sollen alle Knoten vom Typ $t$ als Ziffern gruppiert werden.

Ausgabe: im Baum `num` sind die Ziffern richtig gruppiert.

Sonstige Variable: Bäume $l, r, \texttt{p1}, \texttt{p2}, \texttt{p3}$

**0** Setze $r \leftarrow n$, $\texttt{p1} \leftarrow \texttt{NULL}$, $\texttt{p2} \leftarrow \texttt{NULL}$, $\texttt{p3} \leftarrow \texttt{NULL}$

**1** Setze $\texttt{p3} \leftarrow \texttt{p2}$, $\texttt{p2} \leftarrow \texttt{p1}$, $\texttt{p1} \leftarrow r$, $l \leftarrow \texttt{left}(r)$

**2** Falls $r \neq \texttt{NULL} \wedge l \neq \texttt{NULL} \wedge \texttt{type}(l) = t$, setze $r \leftarrow \texttt{right}(r)$ und wiederhole Schritt 1.

**3** Falls $r = \texttt{NULL} \vee \texttt{type}(r) = t$, setze $\texttt{num} \leftarrow n$ und der Algorithmus ist fertig.

**4** Falls $\texttt{p3} \neq \texttt{NULL}$, setze $\texttt{right}(\texttt{p3}) \leftarrow \texttt{left}(\texttt{p2})$; falls $\texttt{p3} = \texttt{NULL} \wedge \texttt{p2} \neq \texttt{NULL}$, setze $n \leftarrow \texttt{left}(\texttt{p2})$;
falls $\texttt{p3} = \texttt{NULL} \wedge \texttt{p2} = \texttt{NULL}$, setze $n \leftarrow \texttt{left}(\texttt{p1}), r \leftarrow \texttt{right}(\texttt{p1})$.

**5** Setze $\texttt{setname}(num, "\texttt{\#link}", 0), \texttt{left}(\texttt{num}) \leftarrow n, \texttt{right}(\texttt{num}) \leftarrow \texttt{change\_prec}(r,t)$.

Die Darstellung von Zahlen kann außer Ziffern auch noch ein Dezimaltrennzeichen enthalten (meist ein Punkt oder Komma). Dieses verhält sich syntaktisch so wie ein Operator. Seine Rangordnung ist jedoch für Ziffern höher als die von `#link`. Wir müssen deshalb zuerst das oben vorgestellte Verfahren auf alle `#link`-Bäume anwenden und dann die Ziffernbäume zu den Unterbäumen des Knotens mit dem Dezimaltrennzeichen machen.

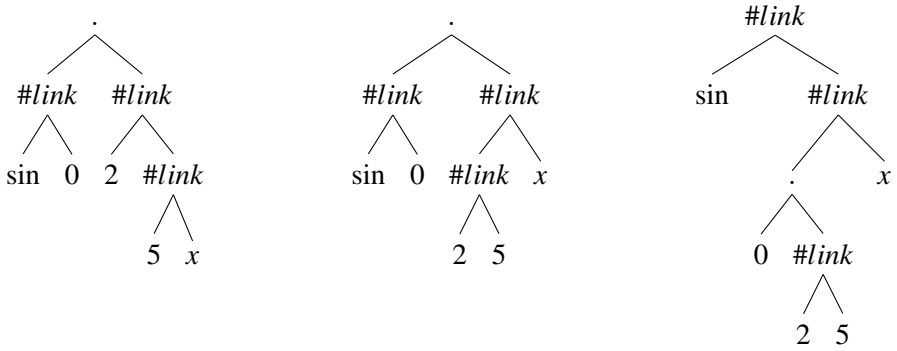



Abbildung 4.2: Umgruppierung von Ziffern mit Dezimaltrennzeichen in der Formel $\sin 0.25x$.

Wir betrachten wieder einen Knoten $n$ mit den Unterbäumen $l$ und $r$. Ist das Token in $n$ der Operator `#link`, dann wenden wir den oben beschriebenen Algorithmus auf $n$ an und sind fertig. Enthält $n$ kein Dezimaltrennzeichen, so wenden wir das Verfahren rekursiv auf $l$ und $r$ an. Wenn $n$ jedoch ein Dezimaltrennzeichen enhält, müssen wir die Unterbäume untersuchen. Wir wollen insgesamt die vier Bäume $\texttt{lnum}, \texttt{rnum}, \texttt{llink}, \texttt{rlink}$ erhalten, die wir dann neu zusammensetzen. Dabei sind `lnum` und `rnum` der ganzzahlige und rationale Teil der Zahl (also die Zifferngruppen, die

links bzw. rechts vom Dezimaltrennzeichen stehen) und `llink` und `rlink` die Bäume, die durch einen `#link`-Operator links bzw. rechts mit der Zahl verbunden sind. Ist $l$ ein `#link`-Baum, dann

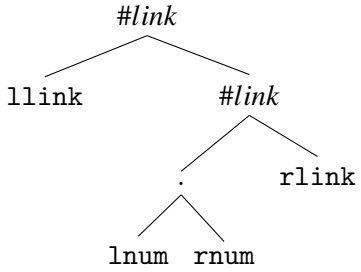



Abbildung 4.3: Struktur eines Baumes, in dem eine Zahl mit Dezimaltrennzeichen links und rechts durch `#link` mit anderen Tokens verbunden ist.

bezeichnen wir mit `p1` den untersten rechten Knoten in $l$, dessen linker Unterbaum keine Ziffer enthält, mit `p2` den Eltern-Knoten von `p1` und mit `p3` den von `p2`. Ist $\texttt{p1} \neq \texttt{NULL}$ und $\texttt{p2} \neq \texttt{NULL}$, dann setzen wir $\texttt{right}(\texttt{p2}) \leftarrow \texttt{left}(\texttt{p1})$ und $\texttt{llink} \leftarrow \texttt{left}(\texttt{l})$. In beiden Fällen setzen wir dann $\texttt{lnum} \leftarrow \texttt{right}(\texttt{p1})$. Das Verfahren wird in Algorithmus 4.2.3 beschrieben und von der Funktion `make_num` auf einen Baum, einen Typen und ein Dezimaltrennzeichen angewendet.

**Algorithmus 4.2.3** Umgruppierung von Ziffern mit Dezimaltrennzeichen

Eingabe: im Baum $n$ werden Knoten mit dem Typ $t$ als Ziffern gruppiert und $d$ wird als Dezimaltrennzeichen verwendet.

Ausgabe: im Baum $n$ sind alle Ziffern richtig gruppiert.

Sonstige Variable: String `name`, Bäume $l, r, \texttt{llink}, \texttt{rlink}, \texttt{p1}, \texttt{p2}, \texttt{p3}, \texttt{P1}, \texttt{P2}, \texttt{P3}$

**0** Setze $\texttt{name} \leftarrow \texttt{getName}(n,0)$. Falls $\texttt{name} = \text{"}\texttt{\#link}\text{"}$, setze $n \leftarrow \texttt{change\_prec}(n,t)$ und der Algorithmus ist fertig.

**1** Setze $l \leftarrow \texttt{left}(n), r \leftarrow \texttt{right}(n), \texttt{llink} \leftarrow 0, \texttt{rlink} \leftarrow \texttt{NULL}$.

**2** Falls $\texttt{name} \neq d$, setze $l \leftarrow \texttt{make\_num}(l,t,d)$, $r \leftarrow \texttt{make\_num}(r,t,d)$ und der Algorithmus ist fertig.

**3** Falls $\texttt{getName}(l,0) \neq \text{"}\texttt{\#link}\text{"}$, weiter bei Schritt 8.

**4** Setze $\texttt{p1} \leftarrow l, \texttt{p2} \leftarrow \texttt{NULL}, \texttt{p3} \leftarrow \texttt{NULL}, \texttt{P1} \leftarrow \texttt{NULL}, \texttt{P2} \leftarrow \texttt{NULL}, \texttt{P3} \leftarrow \texttt{NULL}$.

**5** Falls $\texttt{right}(\texttt{p1}) = \texttt{NULL}$, weiter bei Schritt 7.
Falls $\texttt{left}(\texttt{p1}) \neq \texttt{NULL} \wedge \texttt{getType}(\texttt{left}(\texttt{p1}),0) \neq t$, setze $\texttt{P1} \leftarrow \texttt{p1}, \texttt{P2} \leftarrow \texttt{p2}, \texttt{P3} \leftarrow \texttt{p3}$.

Setze $\texttt{p3} \leftarrow \texttt{p2}, \texttt{p2} \leftarrow \texttt{p1}, \texttt{p1} \leftarrow \texttt{right}(\texttt{p1})$.

**6** Wiederhole Schritt 5, solange $\texttt{p1} \neq \texttt{NULL}$ ist.

**7** Falls $\texttt{P1} = \texttt{NULL}$ weiter bei Schritt 8.

Falls $\texttt{P2} \neq \texttt{NULL}$ setze $\texttt{right}(\texttt{P2}) \leftarrow \texttt{left}(\texttt{P1})$ und $\texttt{llink} \leftarrow l$.

Falls $\texttt{P2} = \texttt{NULL}$ setze $\texttt{llink} \leftarrow \texttt{left}(l)$.

Setze $l \leftarrow \texttt{right}(\texttt{P1})$.

**8** Falls $\texttt{getName}(r,0) \neq \texttt{"\#link"}$, setze $r \leftarrow \texttt{left}(\texttt{p2})$ und weiter bei Schritt 13.

Erzeuge neue Knoten $\texttt{p1},\texttt{p2},\texttt{p3}$ und setze $\texttt{p1} \leftarrow r, \texttt{p2} \leftarrow 0, \texttt{p3} \leftarrow 0$.

**9** Falls $\texttt{p1} = \texttt{NULL} \vee \texttt{right}(\texttt{p1}) = \texttt{NULL} \vee \texttt{getType}(\texttt{left}(\texttt{p1}),0) \neq t$ weiter bei Schritt 12.

**10** Setze $\texttt{p3} \leftarrow \texttt{p2}, \texttt{p2} \leftarrow \texttt{p1}, \texttt{p1} \leftarrow \texttt{right}(\texttt{p1})$.

**11** Zurück zu Schritt 9, falls $\texttt{p1} \neq \texttt{NULL}$ ist.

**12** Falls $\texttt{p1} \neq \texttt{NULL} \wedge \texttt{getType}(\texttt{p1},0) = t$ weiter bei Schritt 13.

Setze $\texttt{rlink} \leftarrow \texttt{p1}$. Falls $\texttt{p3} \neq \texttt{NULL}$, setze $\texttt{right}(\texttt{p3}) \leftarrow \texttt{left}(\texttt{p2})$, sonst setze $r \leftarrow \texttt{left}(\texttt{p2})$.

**13** Setze $\texttt{setName}(\texttt{link},\texttt{"\#link"},0)$, $\texttt{setName}(\texttt{num},d,0)$, $\texttt{left}(\texttt{num}) \leftarrow l$, $\texttt{right}(\texttt{num}) \leftarrow r$

**14** Erzeuge einen neuen Knoten $n$.

Falls $\texttt{rlink} = \texttt{NULL}$, weiter bei Schritt 15.

Setze $\texttt{left}(\mathit{link}) \leftarrow \texttt{num}$ und $\texttt{right}(\mathit{link}) \leftarrow \texttt{rlink}$.

Falls $\texttt{llink} \neq \texttt{NULL}$, setze $\texttt{setName}(n,\texttt{"\#link"},0)$, $\texttt{left}(n) \leftarrow \texttt{llink}$, $\texttt{right}(n) \leftarrow \texttt{link}$;

sonst setze $n \leftarrow \texttt{link}$. Der Algorithmus ist fertig

**15** Falls $\texttt{llink} = \texttt{NULL}$, weiter bei Schritt 16. Setze $\texttt{setName}(n,\texttt{"\#link"},0), \texttt{left}(n) \leftarrow \texttt{llink}, \texttt{right}(n) \leftarrow \texttt{num}$ und der Algorithmus ist fertig.

**16** Setze $n \leftarrow \texttt{num}$ und der Algorithmus ist fertig.

### 4.2.3 Operatoren aus mehreren Tokens

Bis jetzt sind wir immer von zwei Annahmen ausgegangen:

- Jedem semantischen Token entspricht genau ein LaTeX-Token.
- Operatoren und Klammern werden immer als solche erkannt.

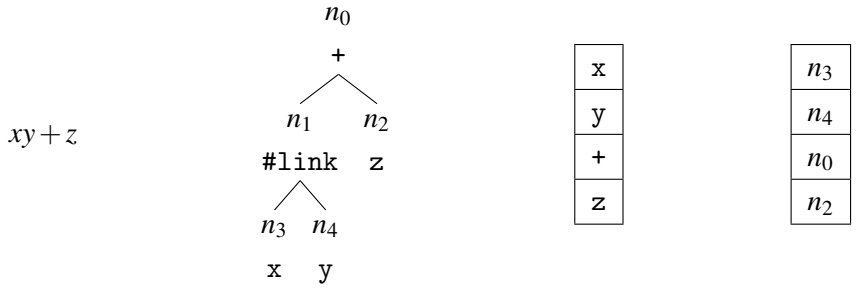
In der Praxis stimmt das aber leider sehr häufig nicht. Es kann etwa vorkommen, dass ein semantisches Token in LaTeX durch mehrere Tokens dargestellt wird, wie man an diesen Formeln sehen kann:

$$a +^* b \quad \texttt{a +\^{}* b}$$

$$U \underset{\text{offen}}{\subseteq} \mathbb{R} \quad \texttt{U \underset\{\text\{offen\}\}\{\subseteq\} \mathbb R}$$

Dafür können wir den umgekehrten Fall, dass nämlich mehrere semantische Tokens durch ein einziges LaTeX-Token dargestellt werden, ausschließen. In den beiden Beispielen kommen Operatoren vor, die aus mehreren Tokens bestehen. Sie können daher nicht als solche erkannt werden. Streng genommen könnte das auch mit Klammern passieren, da das in der Praxis allerdings nie der Fall zu sein scheint, werden wir uns damit nicht weiter beschäftigen. Wir brauchen jedoch Algorithmen, um mehrere Tokens zu einem einzigen zusammenzufassen und bestimmte Tokens zu Operatoren zu machen (unter Berücksichtigung von Rangordnung und Assoziativität).

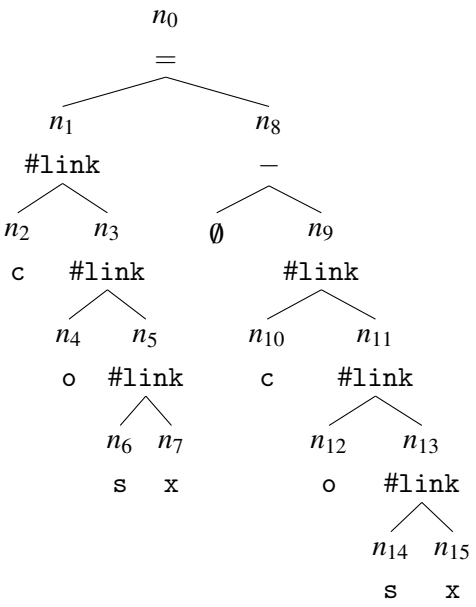
Wenn wir mehrere Tokens zusammenfassen, müssen wir eine Zeichenfolge in einem Binärbaum suchen und ersetzen. Es gibt eine Reihe von Algorithmen zur Suche von Zeichenfolgen in Bäumen[1], die für unsere Zwecke jedoch alle nicht wirklich gut geeignet sind, weil in unseren Bäumen z. B. auch der unsichtbare `#link`-Operator vorkommt. Außerdem dienen sie nur der Suche und können nicht zum Ersetzen verwendet werden. In dem Verfahren, das wir hier verwenden, wird zuerst der Baum in einen Vektor von Tokens zurückverwandelt. Gleichzeitig wird noch ein zweiter Vektor erzeugt, der die Adressen der zugehörigen Knoten enthält. Betrachten wir etwa eine Formel, die aus den Tokens $t_0,\ldots,t_k$ besteht und durch einen Baum mit den Knoten $n_0,\ldots,n_k$ dargestellt wird. Dann erzeugen wir einen Vektor $v$ mit $v_i = t_i$, der die Tokens in der Reihenfolge enthält, in der sie in der Formel vorkommen, und einen zweiten Vektor $w$ mit $w_i = n_j$, wobei $n_j$ die Adresse des Knotens ist, der das Token $t_j$ enthält.




[1] Die klassische Referenz zu dem Thema ist [Hoffmann, O'Donnell 1982], wo die wichtigsten Konzepte und Überlegungen vorgestellt werden.

Der nächste Schritt besteht darin, dass in dem Vektor mit den Tokens alle Vorkommen der zu ersetzenden Sequenz gesucht werden. Die Nummern der Einträge, die die Anfänge der gefundenen Sequenzen sind, werden in einem eigenen Vektor $f$ gespeichert.

**Beispiel** Wir wollen in der Formel `cos x = -cos x` jedes Vorkommen der Sequenz `cos` finden. Die Formel wird durch einen Baum dargestellt. Zuerst speichern wir im Vektor $v$ die Tokens in der richtigen Reihenfolge ab und im Vektor $t$ die Adressen der zugehörigen Knoten. Dann suchen wir in $v$ alle Vorkommen der Sequenz `cos` und speichern die Nummern des jeweils ersten Tokens im Vektor $f$.




`cos x = -cos x`

| c | o | s | x | = | - | c | o | s | x |
|---|---|---|---|---|---|---|---|---|---|
| $v_0$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ | $v_8$ | $v_9$ |
| $n_2$ | $n_4$ | $n_6$ | $n_7$ | $n_0$ | $n_8$ | $n_{10}$ | $n_{12}$ | $n_{14}$ | $n_{15}$ |

| $f_0 = 0$ | $f_1 = 6$ |
|---|---|

In dem Token-Vektor können dann Zeichenfolgen gesucht werden.[2] Durch den zweiten Vektor mit den Adressen ist bekannt, wo sich diese Tokens in dem Baum befinden und können dann zusammengefasst werden, indem die jeweiligen Bäume durch einen einzigen ersetzt werden. Dabei beschränken wir uns jedoch auf zwei Fälle: alle Tokens sind entweder Operatoren oder Namen (andere Möglichkeiten treten ohnehin nie auf).

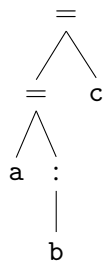
Die Tokens des Baumes sind also im Vektor $v$ abgespeichert und die zugehörigen Knoten im Vektor $n$. Wir wollen jetzt die Sequenz $s_0, \ldots, s_{k-1}$ zu einem Token zusammenfassen. Wenn wir ein Vorkommen davon in $v$ gefunden haben, gibt es einen Index $i$, sodass $v_i = s_0, v_{i+1} = s_1, \ldots, v_{i+k-1} = s_{k-1}$ gilt. Besonders interessant sind der erste und der letzte Knoten der Sequenz sowie der erste Knoten nach der Sequenz in der Formel. Wir verwenden für sie zur besseren Lesbarkeit die Bezeichnungen $\texttt{first} = n_i, \texttt{last} = n_{i+k-1}, \texttt{next} = n_{i+k}$ (falls $i + k < \texttt{size}(n)$).

Im ersten Fall sind $s_0, \ldots, s_{k-1}$ Operatoren. Wir betrachten die Unterbäume $L = \texttt{left}(\texttt{first})$ und $R = \texttt{right}(\texttt{last})$. Dann erzeugen wir einen neuen Baum $T$, in dessen Knoten das Token $s_0 \cdots s_{k-1}$ steht und setzen $\texttt{left}(T) \leftarrow L, \texttt{right}(T) \leftarrow R$. Anschließend ersetzen wir den Baum `first` durch

[2] Eine Übersicht über die wichtigsten geeigneten Suchverfahren für Zeichenketten befindet sich in [Cormen, Leiserson, Rivest, Stein 2007].

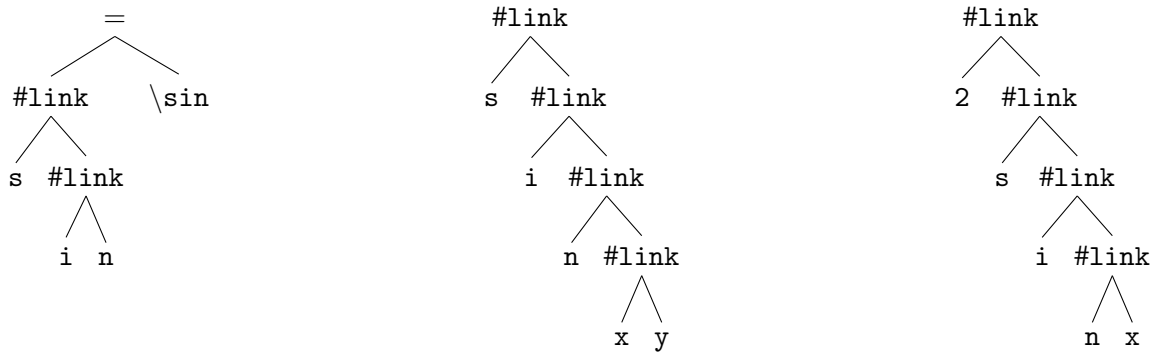
$T$. Als Beispiel können wir so in der Formel $x + -y$ die Sequenz $+-$ zu einem einzigen Operator machen:


Diese Vorgehensweise hat den Nachteil, dass sie nicht immer funktioniert. Haben die Operatoren, die zusammengefasst werden sollen, unterschiedliche Rangordnung, dann hat der Baum nämlich eine andere Struktur, wie man am Beispiel `a = b := c` sieht:




Wir müssen also immer überprüfen, ob das Token in $\texttt{left}(\texttt{next})$ mit $v_{i+k-1}$, dem Token in `last`, übereinstimmt. Ist dies nicht der Fall (so wie im letzten Beispiel), dann nehmen wir den „linkesten“ Knoten im Baum `next`, d. h. wir setzen zuerst $l \leftarrow \texttt{left}(\texttt{next})$ und anschließend so lange wie möglich $l \leftarrow \texttt{left}(l)$. Dann ist $l$ der neue rechte Unterbaum von $L$.

Im zweiten Fall werden Tokens vom Typ `NAME` zusammengefasst; wir können davon ausgehen, dass sie im Baum durch `#link`-Operatoren verbunden sind. Das heißt insbesondere, dass die gesamte Sequenz in einem eigenen Unterbaum aus `#link`-Operatoren enthalten sein muss (weil diese höhere Rangordnung haben als jeder andere Operator). Die Sequenz kann am Anfang, am Ende oder in der Mitte dieses Baumes stehen, es ist auch möglich, dass der ganze Unterbaum nur aus der Sequenz besteht:

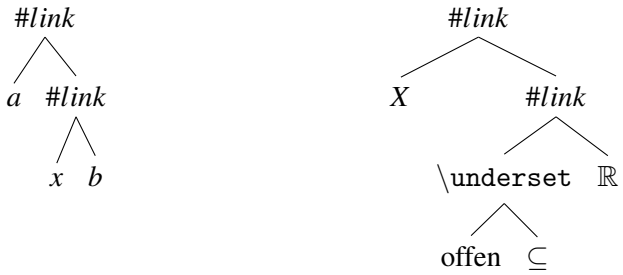



Dabei ist der einzige wichtige Unterschied, ob der letzte und der vorletzte Knoten der Sequenz Unterbäume desselben Baumes sind oder nicht. Der Baum, den wir hier ersetzen müssen, ist der Elternbaum des ersten Knotens der Sequenz; wir setzen daher $T \leftarrow \texttt{parent}(t, \texttt{first})$. Falls die

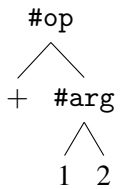
letzten beiden Knoten der Sequenz denselben Elternbaum haben, können wir $T$ einfach durch einen Baum ohne Unterbäume ersetzen, in dessen Knoten das Token $s_0 \cdots s_{k-1}$ steht. Haben sie keinen gemeinsamen Elternbaum, dann erhält $T$ im Knoten das Token `#link` und als linken Unterbaum den Baum mit $s_0 \cdots s_{k-1}$ im Knoten. Kommt nach der Sequenz in der Formel nur noch ein einzelnes Token, dann haben `last` und `next` denselben Unterbaum und wir können $\texttt{right}(T) \leftarrow \texttt{next}$ setzen. Es kann jedoch auch sein, dass nach der Sequenz noch mehrere Tokens stehen, die dann in dem Baum alle durch `#link` verbunden sind. In diesem Fall müssen wir als rechten Unterbaum von $T$ den Elternbaum von `next` nehmen und setzen $\texttt{right}(T) \leftarrow \texttt{parent}(N, \texttt{next})$.

### 4.2.4 Operatoren, die nicht erkannt werden

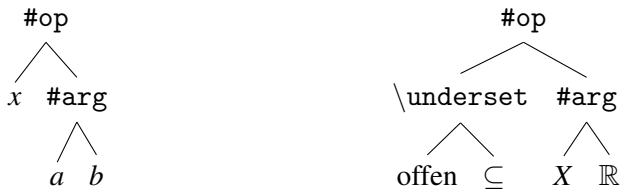
Wenn ein Operator nicht als solcher erkannt wird, kann das im Prinzip zwei Gründe haben: entweder der Operator wurde nicht deklariert und wird wie ein Token vom Typ `NAME` behandelt, oder der Operator besteht aus mehreren Tokens. In beiden Fällen wird der Operator dann durch einen eigenen Baum dargestellt und steht nicht nur im Knoten eines Baumes, dessen Unterbäume dann die Operanden sind. Die Formeln *axb* (wobei *x* hier ein Operator ist) und $X \underset{\text{offen}}{\subseteq} \mathbb{R}$ sind typische Beispiele dafür:



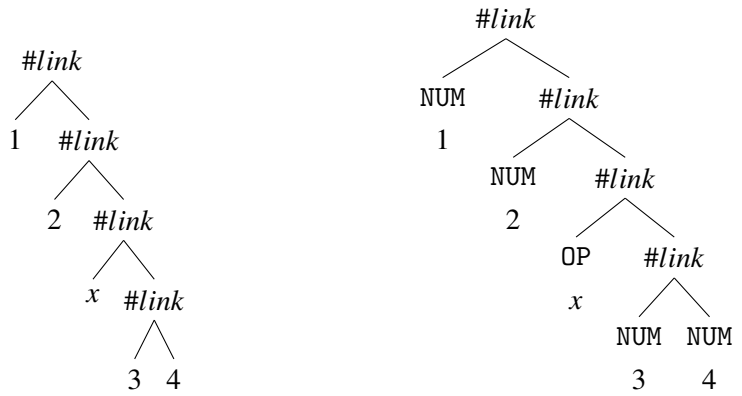

Bevor wir uns weiter mit der Behandlung solcher Operatoren beschäftigen, führen wir eine neue Schreibweise ein. Bis jetzt haben wir (vereinfacht) zwischen Operatoren und Operanden unterschieden. Stattdessen können wir jedoch auch alle Tokens als Argumente einer Anwendungs-Funktion auffassen, die die Operatoren auf ihre Operanden anwendet. Statt $1+2$ würden wir dann $\text{appl}(+,1,2)$ schreiben. Diese dreistellige Funktion ist allerdings für unsere Binärbäume nicht gut geeignet. Deshalb führen wir die Operatoren `#arg` (für die Liste der Argumente) und `#op` (zur Verbindung des Operators mit seinen Argumenten) ein. So erhalten wir die besser passende äquivalente Schreibweise $\texttt{\#op}(+, \texttt{arg}(1,2))$ und folgenden Baum:



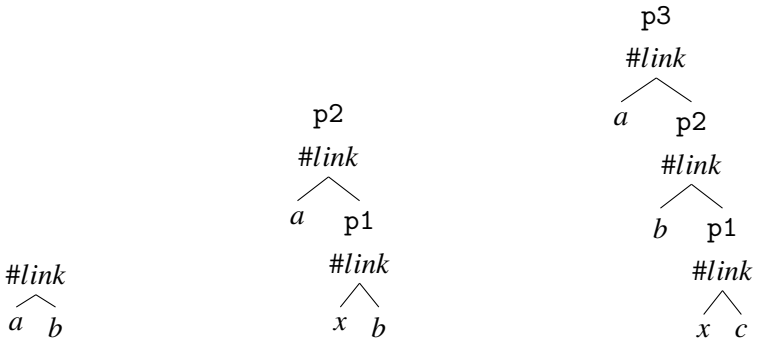

Diese Art der Darstellung hat den Vorteil, dass ein Operator nicht nur aus einem einzelnen Token bestehen muss, sondern auch ein eigener Baum sein kann. Die beiden Bäume von oben hätten dann folgende Darstellung:




Da `#link` bei uns rechtsassoziativ ist, sind rechtsassoziative Operatoren jetzt ein wenig einfacher und wir beginnen deshalb mit ihnen. Im einfachsten Fall haben wir es mit einem Baum zu tun, der nur aus Namen besteht, die durch `#link`-Operatoren verbunden sind. Ein solcher Baum könnte vor und nach der Typbestimmung etwa so aussehen:




Hier ist also der Baum mit dem Typ `OP` eindeutig als Operator gekennzeichnet. Der nächste Schritt besteht dann darin, genau solche Bäume zu finden. Das, was links bzw. rechts davon steht, werden dann die Operanden.




Genauer geht das so: Wir betrachten den Baum $n$, der unsere Formel enthält und machen alle Unterbäume vom Typ $t$ zu Operatoren. Bei einem rechtsassoziativen Operator suchen wir das erste Vorkommen eines solchen Baumes in der Formel. Wir bezeichnen mit `P1` den Eltern-Knoten dieses Baumes, mit `P2` den Eltern-Knoten von `P1` und mit `P3` den von `P2`. Das rechte Argument des Operators erhalten wir, indem wir das Verfahren auf den Baum `right(P1)` anwenden. Für das linke Argument müssen wir überprüfen, ob `P2` überhaupt einen Eltern-Knoten hat. Ist dies nicht der Fall, dann gilt `P3` = `NULL` und das linke Argument ist `left(P2)`. Ansonsten müssen wir den rechten

**Algorithmus 4.2.4** Erzeugung eines rechtsassoziativen Operators in einem `#link`-Baum

Eingabe: im Baum $n$ sollen alle Knoten mit dem Typ $t$ zu rechtsassoziativen Operatoren gemacht werden. Der Baum $n$ muss `#link` im Knoten haben.

Ausgabe: Baum `op`

Sonstige Variable: Bäume $l, r$, `p1`, `p2`, `p3`

**0** Setze $r \leftarrow n$, `p1` $\leftarrow$ `NULL`, `p2` $\leftarrow$ `NULL`, `p3` $\leftarrow$ `NULL`

**1** Setze `p3` $\leftarrow$ `p2`, `p2` $\leftarrow$ `p1`, `p1` $\leftarrow r, l \leftarrow$ `left`$(r), r \leftarrow$ `right`$(r)$

**2** Falls $r \neq$ `NULL` $\wedge$ `type`$(l) \neq t$, wiederhole Schritt 1.

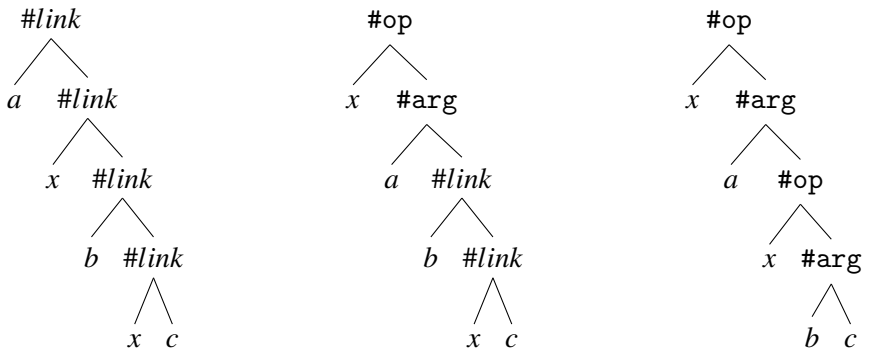
**3** Falls $r =$ `NULL`, setze `op` $\leftarrow n$ und der Algorithmus ist fertig.

**4** Setze `setName`$(arg, "$`#arg`$", 0)$.

Falls `p3` $\neq$ `NULL`, setze `right`$($`p3`$) \leftarrow$ `left`$($`p2`$)$ und `left`$($`arg`$) \leftarrow n$;
sonst setze `left`$($`arg`$) \leftarrow$ `left`$($`p2`$)$.

Setze `right`$($`arg`$) \leftarrow$ `roptree`$(r,t)$,
`setname`$($`op`$, "$`#op`$", 0)$, `left`$($`op`$) \leftarrow l$, `right`$($`op`$) \leftarrow$ `arg`,

Unterbaum von `P3` auf `left`$($`P2`$)$ setzen, und das linke Argument ist $n$. Das Verfahren ist in Algorithmus 4.2.4 beschrieben, und die Funktion `roptree` wendet es auf einen Baum und einen bestimmten Typen an.




Bei einem linksassoziativen Operator suchen wir das erste Vorkommen eines solchen Baumes in der Formel. Die Knoten `P1`, `P2`, `P3` sind wieder wie vorher. Das rechte Argument ist `right`$($`P1`$)$. Für das linke Argument müssen wir wieder prüfen, ob `P3` $=$ `NULL` ist. Wenn ja, dann ist das linke Argument einfach der linke Unterbaum von `P2`. Sonst müssen wir den rechten Unterbaum von `P3` auf `left`$($`P2`$)$ setzen und anschließend das Verfahren auf $n$ anwenden, um das linke Argument zu erhalten. Falls kein Operator gefunden wurde, ist `P1` $=$ `NULL` und der Baum muss nicht verändert werden. Die Funktion `loptree` wendet dieses Verfahren, das in Algorithmus 4.2.5 zusammengefasst ist, auf einen Baum und einen bestimmten Typen an.

**Algorithmus 4.2.5** Erzeugung eines linksassoziativen Operators in einem `#link`-Baum

Eingabe: im Baum $n$ sollen alle Knoten mit dem Typ $t$ zu linksassoziativen Operatoren gemacht werden. Der Baum $n$ muss `#link` im Knoten haben.

Ausgabe: Baum `op`

Sonstige Variable: Bäume $l, r, \texttt{p1}, \texttt{p2}, \texttt{p3}, \texttt{P1}, \texttt{P2}, \texttt{P3}$

**0** Setze $r \leftarrow n$, $\texttt{p1} \leftarrow \texttt{NULL}, \texttt{p2} \leftarrow \texttt{NULL}, \texttt{p3} \leftarrow \texttt{NULL}, \texttt{P1} \leftarrow \texttt{NULL}, \texttt{P2} \leftarrow \texttt{NULL}, \texttt{P3} \leftarrow \texttt{NULL}$.

**1** Setze $\texttt{p3} \leftarrow \texttt{p2}, \texttt{p2} \leftarrow \texttt{p1}, \texttt{p1} \leftarrow r, r \leftarrow \texttt{right}(r)$.

**2** Falls $r = \texttt{NULL}$, weiter bei Schritt 3.

Setze $l \leftarrow \texttt{left}(\texttt{p1})$.

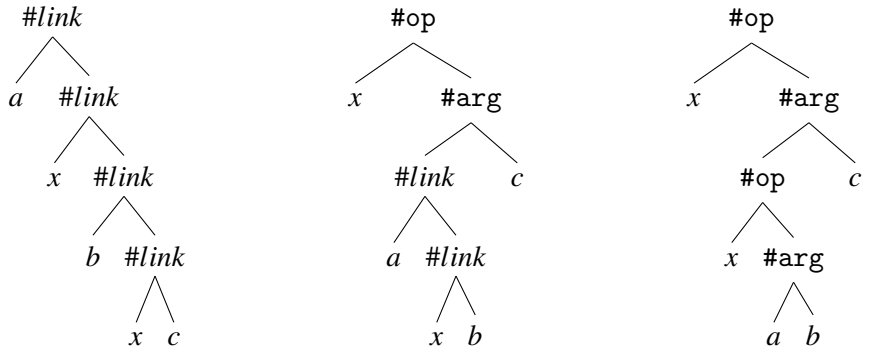
Falls $l \neq \texttt{NULL}$ und $\texttt{type}(l) = t$, setze $\texttt{P1} \leftarrow \texttt{p1}, \texttt{P2} \leftarrow \texttt{p2}, \texttt{P3} \leftarrow \texttt{p3}$ und wiederhole Schritt 1.

**3** Falls $\texttt{P1} = \texttt{NULL}$, setze $\texttt{op} \leftarrow n$ und der Algorithmus ist fertig.

**4** Setze $\texttt{setname}(\texttt{arg}, \texttt{"\#arg"}, 0)$.

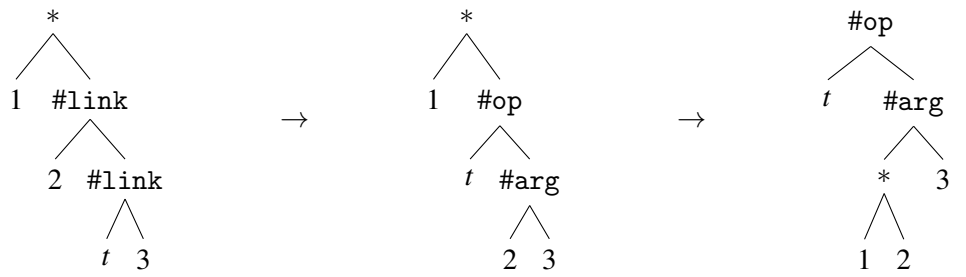
Falls $\texttt{P3} \neq \texttt{NULL}$, setze $\texttt{right}(\texttt{P3}) \leftarrow \texttt{left}(\texttt{P2})$ und $\texttt{left}(\texttt{arg}) \leftarrow \texttt{loptree}(n, t)$;
sonst setze $\texttt{left}(\texttt{arg}) \leftarrow \texttt{left}(\texttt{P2})$.

Setze $\texttt{right}(\texttt{arg}) \leftarrow \texttt{right}(\texttt{P1})$,
$\texttt{setname}(op, \texttt{"\#op"}, 0)$, $\texttt{left}(\texttt{op}) \leftarrow \texttt{left}(\texttt{P1})$, $\texttt{right}(\texttt{op}) \leftarrow \texttt{arg}$.

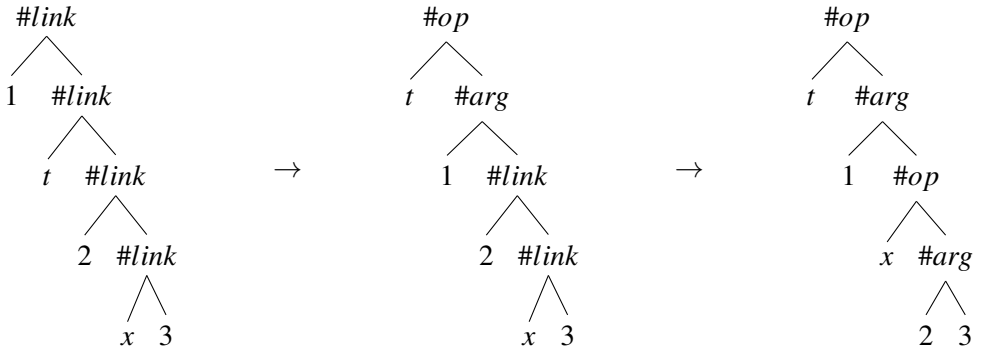



Dabei ist jedoch das Problem der Rangordnung noch nicht berücksichtigt. Müssen mehrere Operatoren nachträglich erkannt werden, dann beginnt man einfach bei dem Operator mit der niedrigsten Rangordnung und wendet das Verfahren so lange hintereinander an, bis alle Operatoren erkannt sind. Die Algorithmen funktionieren allerdings lediglich in Bäumen, die nur `#link`-Operatoren enthalten. In komplexeren Bäumen, in denen auch andere Operatoren auftreten, ist die naive Vorgehensweise, das Verfahren einfach auf alle `#link`-Bäume anzuwenden. Problematisch wird es, wenn die neuen Operatoren eine niedere Rangordnung haben als Operatoren, die von Anfang an erkannt wurden (das Problem kommt daher, dass `#link` die höchstmögliche Rangordnung hat). Ist also eines der Argumente eines gewöhnlichen Operators ein `#op`-Operator, so muss geprüft werden, ob letzterer die höhere Rangordnung hat. Wenn dies nicht der Fall ist, müssen die Operatoren und ihre Argu-

mente vertauscht werden. Die genaue Vorgehensweise ist hier an Hand der Formel $1 * 2t3$ illustriert, wo $t$ zu einem linksassoziativen Operator mit der Rangordnung 1 gemacht wird (und somit niedere Rangordnung als $*$ hat):




Wie rechts- bzw. linksassoziative Operatoren mit einer bestimmten Rangordnung erzeugt werden, wird in Algorithmus 4.2.6 bzw. Algorithmus 4.2.7 beschrieben. Dazu gibt es die Funktionen `make_r_op` und `make_l_op`. Sie wenden die Algorithmen auf einen Baum, einen String, der den Typen enthält, und eine Zahl, die die Rangordung angibt, an.



**Algorithmus 4.2.6** Erzeugung eines rechtsassoziativen Operators

Eingabe: im Baum $n$ sollen alle Knoten vom Typ $t$ zu rechtsassoziativen Operatoren mit der Rangordnung $p$ gemacht werden.

Ausgabe: Baum $n$

Sonstige Variable: String name, Bäume $l, r$, op, arg, la, na, pa

**0** Setze name $\leftarrow$ getName$(n,0)$.

**1** Falls name $=$ "#link", setze $n \leftarrow$ roptree$(n,t)$ und der Algorithmus ist fertig.

**2** Setze $l \leftarrow$ left$(n), r \leftarrow$ right$(n)$ und wende den Algorithmus auf die Unterbäume an: $l \leftarrow$ make_r_op$(l,t,p)$, $r \leftarrow$ make_r_op$(r,t,p)$.

**3** Falls type(name) $\neq$ NAME $\vee\, p \geq$ precedence(name) ist der Algorithmus fertig.

**4** Falls $r =$ NULL $\vee$ getName$(r,0) \neq$ "#op", weiter bei Schritt 5.

Setze op $\leftarrow$ left$(r)$. Falls type(op) $\neq t$, weiter bei Schritt 5.

Setze arg $\leftarrow$ right$(r)$, la $\leftarrow$ left(arg), setName(na, getName$(n,0),0)$, left(na) $\leftarrow l$, right(na) $\leftarrow$ la, left(arg) $\leftarrow$ na.

Erzeuge einen neuen Knoten $n$ und setze setName$(n,$"#op"$,0)$, left$(n) \leftarrow$ op, right$(n) \leftarrow$ arg.

**5** Falls $l =$ NULL $\vee$ getName$(l,0) \neq$ "#op" ist der Algorithmus fertig.

Setze op $\leftarrow$ left$(l)$; falls getType$(op,0) \neq t$, ist der Algorithmus fertig.

Erzeuge neue Knoten arg, ra, na, pa.

Setze arg $\leftarrow$ right$(l)$, ra $\leftarrow$ right(arg).

Setze pa $\leftarrow$ ra und ra $\leftarrow$ right(right(ra)), solange getName(ra, 0) $=$ "#op".

Setze left(na) $\leftarrow$ ra, right(na) $\leftarrow r$, setName(na, getName$(n,0),0)$, right(pa) $\leftarrow$ na. Erzeuge einen neuen Knoten $n$ und setze $n \leftarrow l$.

**Algorithmus 4.2.7** Erzeugung eines linksassoziativen Operators

Eingabe: im Baum $n$ sollen alle Knoten vom Typ $t$ zu linksassoziativen Operatoren mit der Rangordnung $p$ gemacht werden.

Ausgabe: Baum $n$

Sonstige Variable: Bäume $l, r, \texttt{arg}, \texttt{la}, \texttt{ra}, \texttt{na}$

**0** Setze $\texttt{name} \leftarrow \texttt{getName}(n, 0)$.

**1** Falls $\texttt{name} = \text{''}\texttt{\#link}\text{''}$, setze $n \leftarrow \texttt{loptree}(n, t)$ und der Algorithmus ist fertig.

**2** Setze $l \leftarrow \texttt{left}(n), r \leftarrow \texttt{right}(n)$ und wende den Algorithmus auf die Unterbäume an:
$l \leftarrow \texttt{make\_l\_op}(l, t, p)$, $r \leftarrow \texttt{make\_l\_op}(r, t, p)$.

**3** Falls $\texttt{type}(\texttt{name}) \neq \texttt{OP} \vee p \geq \texttt{precedence}(\texttt{name})$ ist der Algorithmus fertig.

**4** Falls $l = \texttt{NULL} \vee \texttt{getName}(l, 0) \neq \text{''}\texttt{\#op}\text{''} \vee \texttt{getType}(\texttt{left}(l), 0) \neq t$: weiter bei Schritt 5.
Setze $\texttt{arg} \leftarrow \texttt{right}(l)$, $\texttt{la} \leftarrow \texttt{left}(\texttt{arg})$, $\texttt{ra} \leftarrow \texttt{right}(\texttt{arg})$,
Setze $\texttt{setName}(\texttt{na}, \texttt{getName}(n, 0), 0)$, $\texttt{left}(\texttt{na}) \leftarrow \texttt{ra}$, $\texttt{right}(\texttt{na}) \leftarrow r$, $\texttt{right}(\texttt{arg}) \leftarrow \texttt{na}$.
Erzeuge einen neuen Knoten $n$ und setze $n \leftarrow l$.

**5** Falls $r = \texttt{NULL} \vee \texttt{getName}(r, 0) \neq \text{''}\texttt{\#op}\text{''} \vee \texttt{getType}(\texttt{left}(r)) \neq t$ weiter bei Schritt 6.
Setze $\texttt{arg} \leftarrow \texttt{right}(r)$.
Solange $\texttt{arg} \neq \texttt{NULL}$: setze $\texttt{la} \leftarrow \texttt{left}(\texttt{arg})$ und falls $\texttt{getName}(\texttt{la}, 0) = \text{''}\texttt{\#op}\text{''}$
setze $\texttt{arg} \leftarrow \texttt{right}(\texttt{la})$, sonst weiter bei Schritt 6.

**6** Setze $\texttt{la} \leftarrow \texttt{left}(\texttt{arg})$, $\texttt{setName}(\texttt{na}, \texttt{getName}(n, 0), 0)$, $\texttt{left}(\texttt{na}) \leftarrow l$,
$\texttt{right}(\texttt{na}) \leftarrow \texttt{left}(\texttt{arg})$.
Setze $\texttt{left}(\texttt{arg}) \leftarrow \texttt{na}$.
Erzeuge einen neuen Knoten $n$ und setze $n \leftarrow r$; der Algorithmus ist jetzt fertig.

# 5 Math-Bridge

## 5.1 Hintergrund

Der Einstieg in das Mathematikstudium an der Universität Wien wurde in den letzten Jahren durch verschiedene Faktoren immer mehr erschwert: Einerseits gab es Verschärfunge durch gesetzliche Maßnahmen ohne erkennbaren Sinn wie die Einführung der Studien-Eingangs- und -Orientierungsphase, in der durch den Wegfall des Rechts der Studierenden auf mehrere Prüfungswiederholungen ein besonderer Leistungsdruck herrscht, und die Kürzung des Anspruchs auf Beihilfen, die den Druck dann nocheinmal verstärkten. Andererseits stieg die Anzahl der Anfängerinnen und Anfänger in den letzten Jahren immer mehr. Dazu kam noch, dass die TU Wien aufhörte, Lehramtsstudien anzubieten, wodurch es dann an der Universität Wien noch mehr Lehramtskandidatinnen und -kandidaten im ersten Semester gab. Um diesen den Übergang von der Schul- zur Hochschulmathematik trotz der erschwerten Rahmenbedingungen möglichst angenehm zu gestalten, wurde das internationale Projekt Math-Bridge (siehe [Math-Bridge Website]) verwendet. Es sollte dabei helfen, den Erstsemestrigen eine „Brücke“ zur Universität zu bauen.

Für die Darstellung mathematischer Formeln (die natürlich im ersten Semester des Mathematikstudiums nicht unwichtig sind) wird von Math-Bridge die Beschreibungssprache OpenMath (siehe [Openmath Website]) verwendet. Diese ist für Menschen jedoch nur schwer les- bzw. schreibbar. Deshalb wurde für die Übersetzung der Formeln der Parser verwendet, der in dieser Arbeit beschrieben ist. In diesem Kapitel beschäftigen wir uns damit, wie die Semantik-Bäume, die wir im vorigen Kapitel erzeugt haben, in eine OpenMath-Darstellung der Formeln gebracht werden können.

## 5.2 OpenMath

Der OpenMath-Standard ist in erster Linie dazu konzipiert, die Semantik von Formeln zu beschreiben. Im Gegensatz zu LaTeX wird also auch der mathematische Inhalt der Formeln dargestellt, d. h. eine Formel ist nicht nur eine Folge von Zeichen, sondern immer ein Objekt mit semantischen Eigenschaften. Im Folgenden werden die Objekte und die Regeln, nach denen sie zusammengesetzt werden, beschrieben.[1]

[1] Die wichtigsten Referenzen dazu sind [Buswell, Caprotti, Carlisle, Dewar, Gaëtano, Kohlhase 2004] sowie die Formelbeispiele auf der Homepage des OpenMath-Projekts.

### 5.2.1 OpenMath-Objekte

#### Einfache Objekte

**Variable** werden mit Hilfe des XML-Elements `OMV` dargestellt. Es muss dann nur noch der Variablenname in Anführungszeichen angegeben werden. Die Darstellung der Variablen $x$ sieht dann etwa so aus:

```
<OMV name="x"/>
```

**Ganze Zahlen** werden mit `OMI` dargestellt, z. B.

```
<OMI> 42 </OMI>
```

**Fließkommazahlen** werden mit `OMF` dargestellt, z. B.

```
<OMF> 3.1415 </OMF>
```

**Symbole** sind die Zeichen, mit denen Operationen und Funktionen dargestellt werden (z. B. $+, \sin$ usw.). Ihre Bedeutung ist in OpenMath in sogenannten Content Dictionaries definiert. Die Darstellung erfolgt durch `OMS`, wobei sowohl der Name des Symbols als auch der Name des zugehörigen Content Dictionary angegeben werden müssen. Die Funktion sin wird etwa so dargestellt:

```
<OMS cd="transc1" name="sin"/>
```

#### Zusammengesetzte Objekte

**Anwendung** In OpenMath gibt es keine Funktionen oder Operatoren im klassischen Sinne, die eines oder mehrere Argumente haben. Stattdessen kann durch die Anwendungs-Funktion ein Objekt auf ein anderes oder mehrere andere angewendet werden (vgl. 4.2.4). Für die Formel $a+b$ wird also die Darstellungsart $\text{appl}(+,a,b)$ verwendet. Dafür gibt es das XML-Element `OMA`. Die Formel $\sin(x)$ hat z. B. diese Darstellung:

```
<OMA>
  <OMS cd="transc1" name="sin"/>
  <OMV name="x"/>
<OMA>
```

Statt `<OMV name="x"/>` könnte hier auch ein beliebiges anderes OpenMath-Objekt stehen.

**Bindung** Eine Variable kann an ein anderes OpenMath-Objekt gebunden werden. Dazu wird folgende Überlegung aus dem Lambda-Kalkül[2] verwendet: Statt $x \mapsto x^2$ kann man auch $\lambda x.x^2$ schreiben. Die Variable muss dann mittels OMBVAR dargestellt werden, für die Bindung gibt es das Objekt OMBIND, und als Symbol wird $\lambda$ verwendet. Die auf den ersten Blick etwas ungewöhnliche Darstellung der Formel $x \mapsto x^2$ sieht dann so aus:

[2] Siehe hierzu [Church 1936] und [Kleene 1935].

```
<OMBIND>
  <OMS cd="fns1" name="lambda" />
  <OMBVAR>
    <OMV name="x" />
  </OMBVAR>
  <OMA>
    <OMS cd="arith1" name="power" />
    <OMV name="x" />
    <OMI>2</OMI>
  </OMA>
</OMBIND>
```

### 5.2.2 Ganze Formeln

Eine ganze Formel ist immer ein `OMOBJ`-Objekt. Die vollständige Darstellung von $x$ muss daher eigentlich so aussehen:

```
<OMOBJ  xmlns="http://www.openmath.org/OpenMath">
  <OMV name="x"/>
</OMOBJ>
```

Zur besseren Lesbarkeit verzichten wir jedoch im Folgenden darauf, die `OMOBJ`-Elemente jedesmal explizit anzuschreiben.

### 5.2.3 Beispiele

#### Operatoren und Funktionen

Operatoren und Funktoren werden wie schon oben beschrieben mittels `OMA` auf ihre Argumente angewendet. Ein typisches Beispiel für einen binären Operator ist $+$, und die Formel $a+b$ hat folgende Darstellung:

```
<OMA>
  <OMS cd="arith1" name="plus"/>
  <OMV name="a"/>
  <OMV name="b"/>
<OMA>
```

Es gibt eine Reihe von Operatoren bzw. Funktionen, deren Darstellung nach demselben Schema erfolgt wie bei $\sin(x)$ oder $a+b$. In der folgenden Tabelle sind die wichtigsten davon aufgelistet:

| Symbol | Beispiel | Content Dictionary | Name |
|---|---|---|---|
| $+$ | $x+y$ | arith1 | plus |

| | | | |
|---|---|---|---|
| $-$ | $x-y$ | arith1 | minus |
| $-$ | $-1$ | arith1 | unary_minus |
| $\cdot$ | $x \cdot y$ | arith1 | times |
| $=$ | $a = b$ | relation1 | eq |
| $<$ | $a < b$ | relation1 | lt |
| $>$ | $a > b$ | relation1 | gt |
| $\geq$ | $a \geq b$ | relation1 | geq |
| $\leq$ | $a \leq b$ | relation1 | leq |
| log | $\log_2 x$ | transc1 | log |
| ! | $n!$ | integer1 | factorial |
| $\in$ | $x \in M$ | set1 | in |
| $\notin$ | $x \notin M$ | set1 | notin |
| $\mid$ | $\mid$ | set1 | suchthat |
| $\cap$ | $A \cap B$ | set1 | intersect |
| $\cup$ | $A \cup B$ | set1 | union |
| $\setminus$ | $A \setminus B$ | set1 | setdiff |
| $\subseteq$ | $A \subseteq B$ | set1 | prsubset |
| $\subset$ | $A \subset B$ | set1 | subset |
| $\times$ | $\mathbb{Z} \times \mathbb{Z}$ | set1 | cartesian_product |
| $\rightarrow$ | $a \rightarrow b$ | logic1 | implies |
| $\wedge$ | $a \wedge b$ | logic1 | and |
| $\vee$ | $a \vee b$ | logic1 | or |
| $\circ$ | $f \circ g$ | fns1 | left_compose |

Weiters gibt es noch einige Objekte, die eigentlich auch Funktionen oder Operatoren sind, obwohl sie in LaTeX anders dargestellt werden. In OpenMath werden sie aber genau wie andere Funktionen oder Operatoren behandelt. Hier sind typische Beispiele:

| **LaTeX** | **Beispiel** | **OpenMath** |
|---|---|---|
| `\frac` | $\frac{1}{2}$ | `<OMA>`<br>`  <OMS cd="arith1" name="divide"/>`<br>`  <OMI> 1 </OMI>`<br>`  <OMI> 2 </OMI>`<br>`</OMA>` |

| | | |
|---|---|---|
| \choose | $\binom{n}{k}$ | <OMA><br>  <OMS cd="combinat1" name="binomial"/><br>  <OMV name="n"/><br>  <OMV name="k"/><br></OMA> |
| \sqrt | $\sqrt{x}$ | <OMA><br>  <OMS cd="arith1" name="root"/><br>  <OMV name="x"/><br></OMA> |
| ^ | $x^2$ | <OMA><br>  <OMS cd="arith1" name="power"/><br>  <OMV name="x"/><br>  <OMI> 2 </OMI><br></OMA> |
| _ | $x_2$ | <OMA><br>  <OMS cd="elementary" name="sequent"/><br>  <OMV name="x"/><br>  <OMI> 2 </OMI><br></OMA> |

### Klammern

Es gibt eine Reihe von Klammern, die alle die Eigenschaft haben, dass sie ihren Inhalt umschließen. In OpenMath wird eine Umklammerung meist wie eine Funktion aufgefasst, die dann auf den Inhalt der Klammer angewendet wird. Wenn es sich dabei um eine Liste handelt, deren Elemente durch Kommata getrennt sind, so hat der Ausdruck in OpenMath mehrere Argumente. Ein typisches Beispiel dafür ist die Darstellung der Menge $\{0,1,2\}$:

```
<OMA>
  <OMS cd="set1" name="set"/>
  <OMI> 0 </OMI>
  <OMI> 1 </OMI>
  <OMI> 2 </OMI>
</OMA>
```

In der folgenden Tabelle sind weitere Klammern, deren Darstellung in OpenMath nach demselben Schema erfolgt, aufgelistet:

| Klammern | | Beispiel | Content dictionary | Name |
|---|---|---|---|---|
| ( | ) | $(x,y)$ | linalg2 | vector |
| [ | ] | $[0,1]$ | interval1 | interval_cc |
| ] | [ | $]0,1[$ | interval1 | interval_oo |
| [ | [ | $[0,1[$ | interval1 | interval_co |
| ] | ] | $]0,1]$ | interval1 | interval_oc |
| \| | \| | $\|x\|$ | arith1 | abs |

Eine Besonderheit ist die Darstellung der Klammern bei Funktionen. Wir haben schon gesehen, dass $\sin(x)$ einfach durch das Element `OMA` dargestellt wird, ohne dass die Klammern explizit angegeben werden müssen. Ganz analog erfolgt auch die Darstellung von Funktionen, die in keinem Content Dictionary definiert sind. Für die Formel $f(x)$ schreibt man etwa einfach

```
<OMA>
  <OMV name="f"/>
  <OMV name="x"/>
</OMA>
```

Dieselbe Schreibweise wird auch für Multiplikationen verwendet, z. B. bei $a(b+c)$ :

```
<OMA>
  <OMV name="a"/>
  <OMA>
    <OMS cd="arith1" name="plus"/>
    <OMV name="b"/>
    <OMV name="c"/>
  </OMA>
</OMA>
```

### Große Operatoren

Die Darstellung von großen Operatoren wie Summen oder Produkten in OpenMath ist ein wenig kompliziert. Dabei kommt wieder eine Überlegung aus dem Lambda-Kalkül ins Spiel, und es wird ein OMBIND-Objekt erzeugt. Betrachten wir etwa folgende Summe:

$$\sum_{i=1}^{n} i$$

Die Ober- und Untergrenzen werden als ganzzahliges Intervall $[1,n]$ aufgefasst. Die Summe ist dann einfach ein Symbol, das auf dieses Intervall und den Summanden (also $i$) angewendet wird. Zusätzlich muss aber noch die Variable $i$ gebunden werden. Die gesamte Darstellung sieht dann so aus:

```
<OMA>
  <OMS cd="arith1" name="sum"/>
    <OMA>
      <OMS cd="interval1" name="integer_interval"/>
      <OMI> 1 </OMI>
      <OMV name="n"/>
    </OMA>
  <OMBIND>
    <OMS cd="fns1" name="lambda"/>
      <OMBVAR>
        <OMV name="i"/>
      </OMBVAR>
      <OMV name="i"/>
  </OMBIND>
</OMA>
</OMA>
```

Die Darstellung von Produkten erfolgt genauso, nur dass der Name dann `product` ist.

### 5.2.4 Umwandeln von Formeln in OpenMath

Die Erzeugung von OpenMath-Formeln aus einem Binärbaum ist der Typbestimmung sehr ähnlich. Wir müssen also zuerst sinnvolle Typen definieren und eine Typbestimmug durchführen. Von jedem OpenMath-Objekt muss bekannt sein, durch welche Tokens es im Baum dargestellt wird und welche Typen die beiden Unterbäume haben. Ein solches Objekt hat einen Objekt-Typ (`OMI`, `OMF`, `OMV`, `OMS`, `OMA`), einen Namen und ein Content Dictionary. Die beiden letzten Attribute können auch fehlen. Wenn wir dann einen beliebigen Knoten in einem Baum betrachten, ist auf Grund seiner Tokens und der Typen der Unterbäume eindeutig bestimmt, in welches OpenMath-Objekt er umgewandelt werden muss.

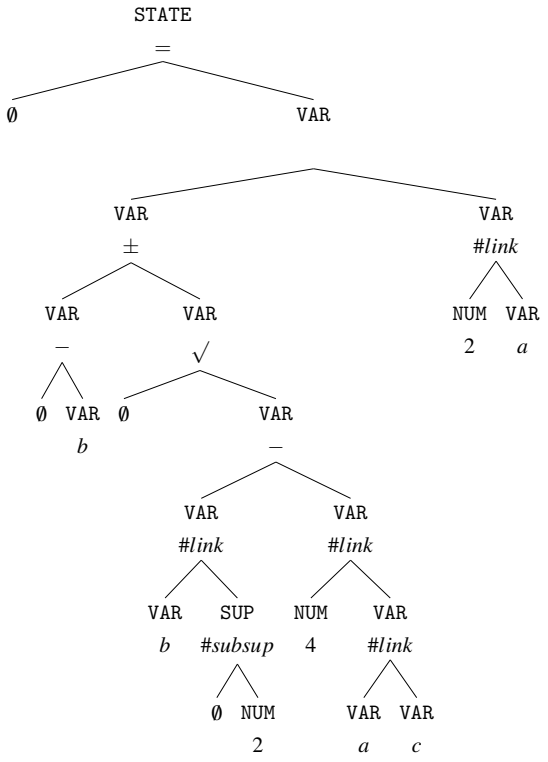
**Beispiel** Das Vorgehen sei hier anhand folgender Formel illustriert:

```
x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Hier sind die Regeln für die Typbestimmung und der Baum, den wir dadurch erhalten:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2: | NUM → NONE | NONE | x: | VAR → NONE | NONE | \sqrt: | VAR → NONE | VAR |
| 4: | NUM → NONE | NONE | =: | VAR → VAR | VAR | \frac: | VAR → VAR | VAR |
| a: | VAR → NONE | NONE | −: | VAR → VAR | VAR | #subsup: | SUP → NONE | NUM |
| b: | VAR → NONE | NONE | −: | VAR → NONE | VAR | #link: | VAR → NUM | VAR |
| c: | VAR → NONE | NONE | \pm: | VAR → VAR | VAR | #link: | VAR → VAR | VAR |
| | | | | | | #link: | VAR → VAR | SUP |




Für die Umwandlung in OpenMath müssen wir noch einige Objekte definieren. Dazu geben wir an, wie die Objekte im Baum dargestellt werden (durch welche Tokens, welche Typen die Unterbäume haben) und wie sie in OpenMath repräsentiert werden:

| Token | Typ | linker Baum | rechter Baum | Content Dictionary | Name |
|---|---|---|---|---|---|
| a | OMV | | | | a |
| b | OMV | | | | b |
| c | OMV | | | | c |
| x | OMV | | | | x |
| = | OMS | VAR | VAR | relation1 | eq |
| - | OMS | VAR | VAR | arith1 | minus |
| - | OMS | | VAR | arith1 | unary_minus |
| \pm | OMS | VAR | VAR | multiops | plusminus |
| \sqrt | OMS | VAR | | arith1 | root |
| \frac | OMS | VAR | VAR | arith1 | divide |
| #subsup | OMS | | NUM | arith1 | power |
| #link | OMS | VAR | VAR | arith1 | times |
| #link | OMS | VAR | SUP | arith1 | power |

Da von Ziffern immer von vornherein klar ist, wie sie dargestellt werden müssen, kann auf Grund dieser Regeln die OpenMath-Ausgabe erzeugt werden. Sie sieht dann so aus:

```
<OMOBJ  xmlns="http://www.openmath.org/OpenMath">
  <OMA cdbase="http://www.openmath.org/cd">
    <OMS cd="relation1" name="eq"/>
    <OMV name="x"/>
    <OMA>
      <OMS cd="arith1" name="divide"/>
      <OMA>
        <OMS cd="multiops" name="plusminus"/>
        <OMA>
          <OMS cd="arith1" name="unary_minus"/>
          <OMV name="b"/>
        </OMA>
        <OMA>
          <OMS cd="arith1" name="root"/>
          <OMA>
            <OMS cd="arith1" name="minus"/>
            <OMA>
              <OMS cd="arith1" name="power"/>
              <OMV name="b"/>
              <OMI>2</OMI>
            </OMA>
            <OMA>
              <OMS cd="arith1" name="times"/>
              <OMI>4</OMI>
              <OMV name="a"/>
              <OMV name="c"/>
            </OMA>
          </OMA>
          <OMI>2</OMI>
        </OMA>
      </OMA>
      <OMA>
        <OMS cd="arith1" name="times"/>
        <OMI>2</OMI>
        <OMV name="a"/>
      </OMA>
    </OMA>
  </OMA>
</OMOBJ>
```

Nach diesem Beispiel wird jetzt hier beschrieben, wie die Umwandlung allgemein funktioniert. Wir gehen davon aus, dass wir einen Knoten *n* bearbeiten, der den Vektor *symb* mit Tokens enthält

und die Unterbäume $l$ und $r$ hat. Ihre Typen sind *ltype* und *rtype*.

Für die Umwandlung brauchen wir jetzt Regeln, die festlegen, welches OpenMath-Objekt aus einem Knoten erzeugt werden soll. Die dazu benötigten Informationen sind die Anzahl der Tokens, der Tokens-Vektor, die Typen des linken und rechten Unterbaumes sowie der Typ, der Namen und der Content Dictionary des Objekts. Außerdem muss noch die Reihenfolge der Unterbäume bekannt sein. Postfix-Operatoren wie ! stehen nämlich hinter ihrem Operanden, und da muss dann die Reihenfolge vertauscht werden.

Eine OpenMath-Regel ist also ein 8-Tupel (`pre`, `num`, `symb`, `ltype`, `rtype`, `object`, `cd`, `name`) bestehend aus einer Booleschen Variable für die Reihenfolge der Unterbäume, der Anzahl der Tokens, dem Vektor der Tokens, den Typen der Unterbäume, dem Objekt, dem Content Dictionary und dem Namen. Wir führen zusätzlich noch `OM#` ein, was ausdrückt, dass aus einem Knoten kein eigenes Objekt erzeugt wird. Das kommt beispielsweise bei runden Klammern oder bei Listen, die durch Kommata getrennt sind, zur Anwendung. (In OpenMath werden die Listenelemente einfach hintereinander geschrieben, siehe oben bei dem Beispiel für eine Menge.) Hier sind einige solcher Regeln:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | `x` | | | OMV | | |
| 1 | 1 | `\frac` | NUM | NUM | OMS | arith1 | divide |
| 1 | 2 | `\| \|` | NUM | NONE | OMS | arith1 | abs |
| 0 | 1 | `#link` | NUM | POSTOP | OMS | integer1 | factorial |
| 1 | 2 | `( )` | NUM | NONE | OM# | | |

Zur Umwandlung in OpenMath verwenden wir die Funktion `openmath`, die aus einem Baum eine OpenMath-Ausgabe erzeugt, und die Funktion `print`, mit der beliebige Datentypen in die Ausgabe geschrieben werden können. Die Ausgabe der einzelnen Objekte funktioniert dann so:

**OMI** Das Objekt ist vom Typ OMI, wenn $symb_0$ eine Ziffer ist oder alle Tokens in $n$ entweder Ziffern oder `#link` sind. Die Ausgabe sieht dann etwa so aus:

```
<OMI> 42 </OMI>
```

Der Baum muss also nur als Ziffernfolge ausgegeben werden.

**OMF** Hier muss $n_0 = ''.''$ sein, und sowohl $l$ als auch $r$ müssen Ziffernfolgen enthalten, also dieselbe Gestalt haben wie ein Baum bei OMI. Die Ausgabe sieht dann etwa so aus:

```
<OMF> 3.1415 </OMF>
```

Bei diesen beiden Zahlen-Objekten ist immer ohne weitere Informationen klar, wie sie in OpenMath dargestellt werden müssen. Für alle anderen Objekte brauchen wir Regeln zur Darstellung, die wir aus den Typen der Unterbäume und den Tokens im Knoten erhalten. Wir suchen also eine Regel $R$ mit $R_{\texttt{symb}} = symb, R_{\texttt{ltype}} = \texttt{type}(l), R_{\texttt{rtype}} = \texttt{type}(r)$. Je nachdem, was $R_{\texttt{object}}$ ist, wird dann ein bestimmtes Objekt erzeugt:

**OMV** Die Unterbäume sind beide leer und es muss nur der Name angegeben werden:

print(″< OMVname = ”″)

print($R_{\texttt{name}}$)

print(″”/ >″)

**OMS** Auch hier sind die Unterbäume leer, es müssen Content Dictionary und Name angegeben werden:

print(″< OMScd = ”″)

print($R_{\texttt{cd}}$)

print(″”name = ”″)

print($R_{\texttt{name}}$)

print(″”/ >″)

**OMA** Bei OMA-Objekten enthält jeder Unterbaum ein eigenes Objekt. Für ihre Reihenfolge in der Ausgabe benötigen wir $R_{\texttt{pre}}$:

Falls $R_{\texttt{pre}} = 1$:

print(″< OMA >″)

openmath($l$)

openmath($r$)

print(″< /OMA >″)

Falls $R_{\texttt{pre}} = 0$:

print(″< OMA >″)

openmath($r$)

openmath($l$)

print(″< /OMA >″)

**OM#** Die Ausgabe ist wie bei OMA, nur dass kein eigenes Objekt erzeugt wird:

Falls $R_{\texttt{pre}} = 1$:

openmath($l$)

openmath($r$)

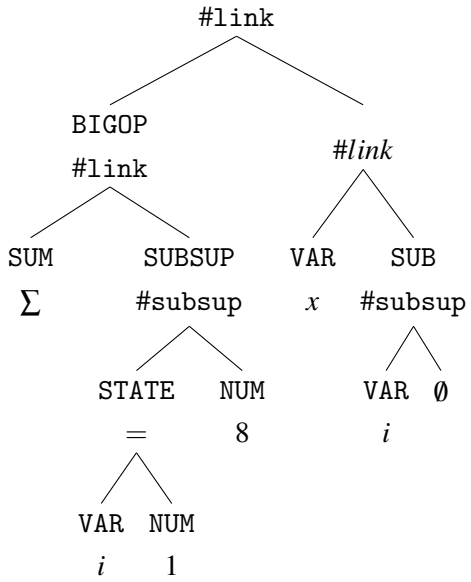
Falls $R_{\texttt{pre}} = 0$:

openmath($r$)

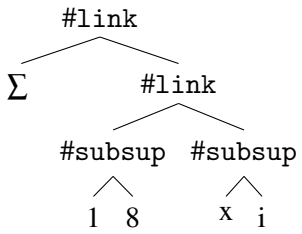
openmath($l$)

### Große Operatoren

Die Darstellung von Summen und Produkten macht etwas Schwierigkeiten. Eine Möglichkeit, diese zu bewältigen, besteht darin, schon bei der Typbestimmung alle großen Operatoren durch den Typ `BIGOP` zu markieren. (Stattdessen kann natürlich auch ein anderer Name gewählt werden.) Der Baum zu der Summe $\sum_{i=1}^{8} x_i$ sieht dann etwa so aus:




Jetzt ist aus der Struktur des Baumes klar, wo die Grenzen der Summe gespeichert sind, nämlich im rechten Unterbaum des `BIGOP`-Baumes. Die Summanden werden durch den rechten Unterbaum des ganzen Baumes dargestellt. Auf Grund dieser Informationen kann jetzt die OpenMath-Ausgabe erzeugt werden. Das ist deshalb immer so einfach möglich, weil es in OpenMath nur Summen und Produkte von dieser Bauart gibt, d. h. $\sum_{n\in\mathbb{N}} \frac{1}{n}$ ist nicht darstellbar, weil es hier nicht eine Ober- und eine Untergrenze gibt. Alternativ könnte man auch die Bäume, die die großen Operatoren darstellen, umformen, bevor sie in OpenMath umgewandelt werden. Der Baum zur Summe $\sum_{i=1}^{8} x_i$ würde dann nach der Umwandlung so aussehen:




Diese Bäume können dann wie gewohnt verarbeitet werden.

## 5.2.5 Fazit

OpenMath ist durchaus interessant für die Darstellung kurzer, einfacher Formeln. Bei komplizierteren oder ungewöhnlichen Formeln macht es jedoch so große Schwierigkeiten, dass es de facto

unbrauchbar ist. Eine Zusammenfassung der wichtigsten technischen Unzulänglichkeiten gibt es in [Kofler, Schodl, Neumaier 2010]. Ich beschränke mich daher hier auf einige praktische Erfahrungen und die daraus entstandenen Probleme. Zum Ersten wurde OpenMath von Informatikern entwickelt, die mit mathematischen Formeln, wie sie wirklich geschrieben werden, scheinbar kaum vertraut sind. Deshalb lässt OpenMath nur sehr restriktive Schreibweisen zu. Schon die einfache Formel $\{n \in \mathbb{N} \mid n \text{ ist gerade}\}$ kann in OpenMath nicht dargestellt werden, weil sie zu informal ist. Ein weiteres Problem sind die Content Dictionaries, in denen alle Objekte definiert sind. Durch die schlechte, unübersichtliche und zum Teil auch unvollständige Dokumentation ist es z. B. unmöglich herauszufinden, wie $x_1$ in OpenMath dargestellt wird. Besonders schwierig wird es, wenn Objekte gar nicht definiert sind. Das Kronecker-Delta $\delta_{ij}$ lässt sich etwa nicht sinnvoll darstellen. Die einzige Lösung besteht darin, es eben nicht als semantisches Objekt, sondern einfach als Zeichenfolge aufzufassen – und damit ist die ganze Formel dann nicht mehr interpretierbar und das eigentliche Ziel von OpenMath verfehlt.

Insgesamt würde ich also von OpenMath abraten und stattdessen eine mächtigere Beschreibungssprache verwenden, die mehr Freiheiten zulässt und besser für „richtige“ mathematische Formeln geeignet ist.

# 6 Concise

An der Fakultät für Mathematik der Universität Wien wird unter Leitung von Prof. Arnold Neumaier die Programmierumgebung Concise entwickelt (siehe [FMathL Website]).Das Ziel des Projekts ist es, eine Sprache für mathematische Modellierung und Dokumentation zu entwickeln, die klar und einfach handzuhaben ist, sich gut auf praktische Probleme anwenden lässt (insbesondere Optimierungsprobleme) und mit der sich die *gesamte* Mathematik beschreiben lässt. Zur Beschreibung der mathematischen Inhalte wird eine Sprache verwendet, die im Moment den Arbeitstitel FMathL (Formal Mathematical Language) trägt. In diesem Kapitel wird beschrieben, wie LaTeX-Formeln in diese Sprache übersetzt werden können. Es geht dabei wirklich nur um die Übersetzung, der Aufbau und die genaue Funktionsweise von Concise und FMathL werden hier nicht erklärt. Für weitere Informationen dazu sei auf die in diesem Kapitel zitierte Literatur verwiesen. Als Referenzen habe ich hauptsächlich [Schodl, Neumaier 2011 a] und [Schodl, Neumaier 2011 b] verwendet.

## 6.1 Die Darstellung von Formeln in Concise

Betrachten wir als einfaches Beispiel folgende Formel:

$$\frac{a}{b}$$

Das ist ein Bruch, der aus einem Zähler und einem Nenner besteht, welche die Formeln $a$ bzw. $b$ enthalten. Diese Überlegung illustriert sehr gut die Weise, wie mathematische Formeln in Concise dargestellt werden. Jede Formel ist ein Ausdruck, der einen bestimmten Typ hat und einen oder mehrere Unterausdrücke enthalten kann. Weiters hat jeder Ausdruck eine Nummer, durch die er eindeutig gekennzeichnet ist.

### 6.1.1 Variable und Konstante

Die einfachsten Ausdrücke sind Variable und Konstanten, die keine Unterausdrücke enthalten. Ihnen muss jedoch am Ende immer ein Wert zugewiesen werden. Die Darstellung der Zahl 1 sieht etwa so aus:

```
$2994.type=Integer
VALUE($2994)=str:1
```

Der Formel entspricht also der Ausdruck mit der Nummer 2994. (Diese Zahl hat keine besondere Bedeutung – wichtig ist nur, dass jeder Ausdruck eine eindeutige Nummer hat.) Der Typ des Aus-

drucks ist `Integer`, und am Ende wird ihm noch der Wert 1 zugewiesen. Variable werden ähnlich dargestellt, sie können optional noch einen Namen haben, der vom Typ `String` ist. Ein Beispiel dafür wäre die Darstellung der Formel $x$:

```
$2990.type=Var
$2990.name=$2992
$2992.type=String
VALUE($2992)=str:x
```

Hier hat der Ausdruck die Nummer 2990 und den Typ `Var`. Er enthält einen Unterausdruck `name`, der hier der Ausdruck mit der Nummer 2992 ist. Dieser ist eine Konstante vom Typ `String`, der zum Schluss der Wert `x` zugewiesen wird. Man könnte den Ausdruck kürzer auch ohne den optionalen Namen so darstellen:

```
$2990.type=Var
VALUE($2990)=str:x
```

### 6.1.2 Operatoren und Funktionen mit einer festen Anzahl von Argumenten

Komplizierte Ausdrücke, die auch Unterausdrücke enthalten, müssen in sogenannten Typesheets definiert werden. Die Definition eines Ausdrucks für Brüche sieht etwa so aus:

```
Fraction:
  allOf> num=Expression
         denom=Expression
```

Der Ausdrück ist also vom Typ `Fraction` und enthält die Unterausdrücke `num` und `denom`, die einfach wieder gewöhnliche Ausdrücke sind. Das Schlüsselwort `allOf` gibt an, dass beide Unterausdrücke immer vorhanden sein müssen. Wenn wir jetzt so den Bruch $\frac{1}{2}$ darstellen, erhalten wir Folgendes:

```
$3134.type=Fraction
$3134.num=3135
$3134.denom=3136
$3135.type=Integer
$3136.type=Integer
VALUE($3135)=str:1
VALUE($3136)=str:2
```

Hier hat der Haupt-Ausdruck den Typ `Fraction` und die Nummer 3134. Seine beiden Unter-Ausdrücke `num` und `denom` sind die Ausdrücke 3135 bzw. 3136. Sie sind die Zahlen 1 und 2, die auf die übliche Weise dargestellt werden. Stattdessen könnten auch beliebige andere Ausdrücke stehen, wodurch sich alle möglichen Formeln darstellen lassen.

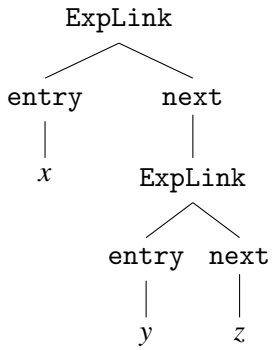
### 6.1.3 Listen

Listen treten bei Aufzählungen auf, in der Zeilenschreibweise von Vektoren oder wenn Funktionen mehrere Argumente haben. Hier sind einige typische Beispiele:

$$\{0,1,2,\dots\} \qquad (x_1,x_2,x_3) \qquad f(x,y)$$

In Concise werden Listen durch Ausdrücke vom Typ `ExpLink` dargestellt. Jeder solche Ausdruck hat die Unter-Ausdrücke `entry`, das aktuelle Listenelement, und `next`, was den Rest der Liste enthält. Die Darstellung der Liste $x,y,z$ in Concise und die dazugehörige Baumstruktur sehen so aus:

```
$3134.type=ExpLink
$3134.next=3135
$3134.entry=3136
$3135.type=Var
$3136.type=ExpLink
$3136.next=3137
$3136.entry=3138
$3137.type=Var
$3138.type=Var
VALUE($3135)=str:x
VALUE($3137)=str:y
VALUE($3138)=str:z
```




Es gibt in Concise noch weitere Fälle, wo Formeln durch Listen dargestellt werden, obwohl man es nicht sofort vermuten würde. Sie haben alle die hier vorgestellte Struktur und werden später erklärt.

### 6.1.4 Relationen

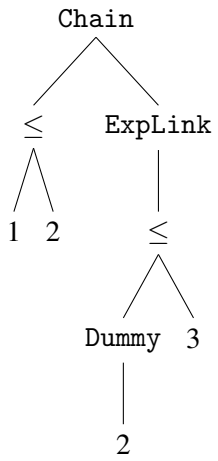
Bei der Syntax haben wir bis jetzt nicht zwischen Operatoren ($+$, $-$ usw.) und Relationen ($=$, $\leq$ usw.) unterschieden. In Concise ist das jedoch nötig, weil es bei der Semantik Unterschiede gibt. Für Relationen gibt es den Ausdruck `Relation`, mit dem eine beliebige Relation zwischen den beiden Ausdrücken `lhs` und `rhs` dargestellt werden kann. Die Formel $1 \leq 2$ ist ein typisches Beispiel:

```
$3134.type=Relation
$3134.relation=LessEq
$3134.lhs=3135
$3134.rhs=3136
$3135.type=Integer
$3136.type=Integer
VALUE($3135)=str:1
VALUE($3136)=str:2
```

Um welche Art von Relation es sich dabei handelt, wird durch `relation` angegeben. Das ist eine Art von Ausdruck, die wir bis jetzt noch nicht kennen, weil es sich dabei weder um die Nummer eines anderen Ausdrucks noch um einen Typen handelt. Wir können das aber auch so auffassen, dass jeder Ausdruck mehrere Typen und deren Namen enthält, wobei der erste Typ dann immer `type` ist. In diesem Beispiel wären die Typen des Ausdrucks `type` und `relation`, und ihre Namen `Relation` und `LessEq`.

Oft werden Schreibweisen wie $1 \leq 2 \leq 3$ verwendet. Da $\leq$ ein Relationssymbol und kein Operator ist, ist das eine Abkürzung für $1 \leq 2 \wedge 2 \leq 3$. Dementsprechend ist die Darstellung in Concise auch ein wenig komplizierter. Die Relationen werden durch den Ausdruck `Chain` verbunden, wobei die zweite Relation (also $2 \leq 3$) in einem Ausdruck `ExpLink` enthalten sein muss. Dadurch, dass die Zahl 2 nur einmal in der Formel vorkommt, muss ihr zweites Vorkommen gewissermaßen „dazugeschummelt“ werden und ist daher in einem `Dummy`-Ausdruck enthalten. Der ganze Concise-Ausdruck und die zugehörige Baumstruktur sehen dann so aus:

```
$3134.type=Chain
$3134.firstrel=3135
$3134.LinkedList=3138
$3135.type=Relation
$3135.relation=LessEq
$3135.lhs=3136
$3135.rhs=3137
$3136.type=Integer
$3137.type=Integer
$3138.type=ExpLink
$3138.entry=3139
$3139.type=Relation
$3139.relation=LessEq
$3139.lhs=3140
$3139.rhs=3142
$3140.type=Dummy
$3140.entry=3137
$3142.type=Integer
VALUE($3136)=str:1
VALUE($3137)=str:2
VALUE($3142)=str:3
```




### 6.1.5 Summen

Die Darstellung von Summen in Concise ist am Anfang etwas ungewöhnlich, hat aber durchaus ihre Berechtigung. Zunächst führen wir eine neue Schreibweise ein und fassen so Summen als Funktio-

nen auf Mengen auf. Sei $M$ eine Menge mit einer binären Operation $+\colon M \times M \to M$, d. h. für alle $x, y \in M$ gilt auch $x + y \in M$. Dann ist eine Summe eine Abbildung

$$\sum\colon \mathcal{P}(M) \to M, \quad \sum X := \sum_{x \in X} x$$

In dieser Schreibweise lassen sich dann auch alle Summen von Zahlen darstellen, z. B.

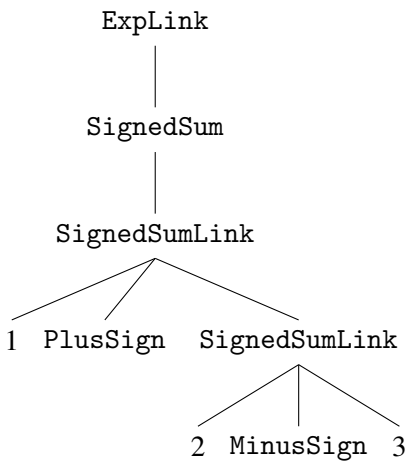
$$1 + 2 + 3 = \sum\{1, 2, 3\}$$

Natürlich müssen nicht alle Summanden positiv sein. Die Minus-Schreibweise $a - b$ ist ja eigentlich nur eine Abkürzung für $a + (-b)$. Dementsprechend sieht die Summen-Darstellung dann so aus:

$$a - b = \sum\{a, -b\}$$

Auf genau dieser Überlegung beruht die Weise, wie Summen (oder allgemeiner Formeln mit $+$ und $-$) in Concise dargestellt werden: Eine Summe ist die Liste ihrer Summanden. Die Summanden sind Ausdrücke vom Typ `SignedSumLink`, die als `sign` das Vorzeichen (also Plus oder Minus) und als `entry` den eigentlichen Summanden enthalten. Der Ausdruck `next` ist wie üblich das nächste Listenelement. Diese `SignedSumLink`-Liste ist Unterausdruck eines Ausdrucks vom Typ `SignedSum`, der wiederum zu einem `ExpLink` gehört. Ein vollständiges Beispiel ist die Darstellung von $1 + 2 - 3$, die so aussieht:

```
$3134.type=ExpLink
$3134.entry=3135
$3135.type=SignedSum
$3135.LinkedList=3136
$3136.type=SignedSumLink
$3136.sign=PlusSign
$3136.entry=3137
$3136.next=3138
$3137.type=Integer
$3138.type=SignedSumLink
$3138.sign=MinusSign
$3138.entry=3139
$3138.next=3140
$3139.type=Integer
$3140.type=Integer
VALUE($3137)=str:1
VALUE($3139)=str:2
VALUE($3140)=str:3
```



### 6.1.6 Mehrfaches Vorkommen von Ausdrücken

Treten Variable oder Konstante mehrfach in einer Formel auf, so wird trotzdem nur ein einziger Ausdruck erzeugt. Bei dem Bruch $\frac{x}{x}$ sind Zähler und Nenner etwa derselbe Ausdruck, der zweimal vorkommt:

```
$3134.type=Fraction
$3134.num=3135
$3134.denom=3135
$3135.type=Var
VALUE($3135)=str:x
```

## 6.2 Die Umwandlung von Semantik-Bäumen in Concise-Ausdrücke

### 6.2.1 Grundsätzliche Überlegungen

#### Concise-Ausdrücke

Ein Concise-Ausdruck ist ein Quintupel $(num, type, name, attr, subex)$, bestehend aus seiner Nummer, den Vektoren, die seine Typen und ihre Namen enthalten sowie den Vektoren, die die Namen und Nummern der Unterausdrücke enthalten.

**Beispiel** Wir hatten oben die Darstellung von $1 \leq 2$. Den Ausdruck mit der Nummer 3134 bezeichnen wir hier mit $e$:

```
$3134.type=Relation
$3134.relation=LessEq
$3134.lhs=3135
$3134.rhs=3136
```

$$e_{num} = 3134$$
$$e_{type} = (\texttt{type}, \texttt{relation})$$
$$e_{name} = (\texttt{Relation}, \texttt{LessEq})$$
$$e_{attr} = (\texttt{lhs}, \texttt{rhs})$$
$$e_{subex} = (3135, 3136)$$

Um die Concise-Ausdrücke darstellen zu können, brauchen wir einerseits einen Vektor, der die Ausdrücke enthält und andererseits einen Vektor für die Werte von Zahlen, Variablen und Konstanten.

#### Regeln für die Umwandlung

Um aus den Semantik-Bäumen die Ausdrücke erzeugen können, müssen wir bei der Typbestimmung die Typen verwenden, die dann auch von Concise verwendet werden. Für die eigentliche Umwandlung brauchen wir Regeln, welcher Ausdruck aus welchem Baum erzeugt wird. Zunächst müssen wir die Tokens im Knoten des Baumes sowie den Typ des Baumes und seiner beiden Unterbäume

kennen, um eindeutig bestimmen zu können, zu welcher semantische Kategorie der Baum gehört. Jeder Kategorie entspricht dann ein bestimmter Ausdruck. Betrachten wir etwa den Baum, der die Formel $\frac{1}{2}$ darstellt:

$$[\texttt{\textbackslash frac}[1]_{\texttt{Integer}}[2]_{\texttt{Integer}}]_{\texttt{Float}}$$

Hier ist klar, dass der Baum einen Bruch darstellt, denn im Knoten steht das Token `\frac` und der Baum hat den Typ `Float` und seine Unterbäume haben beide den Typ `Integer`. Auf Grund dieser Informationen können wir jetzt einen Concise-Ausdruck erzeugen: Wir wissen, dass Brüche als Typ `Fraction` und die Unter-Ausdrücke `num` und `denom` haben. So erhalten wir das schon bekannte Ergebnis.

Es kann jedoch auch vorkommen, dass ein Ausdruck mehr Typen als nur `type` hat. Ein Beispiel dafür war die Formel $1 \leq 2$. Hier wurde zusätzlich noch die Information benötigt, um welche Art von Relation es sich handelt. Es muss also auch noch bekannt sein, dass eine Relation mit dem Relationssymbol $\leq$ einen Typ `relation` mit dem Namen `LessEq` hat.

Insgesamt können wir eine Concise-Regel auffassen als eine Zuordnung

$$(symb, type, htype, ltype, rtype) \rightarrow (lexp, rexp, hexp),$$

die den Tokens des Knotens im Vektor *symb*, dem Typ des Knotens *type*, dem zusätzlichen Typ *htype* und den Typen der Unterbäume *ltype* und *rtype* die Namen der Concise-Ausdrücke *lexp*, *rexp*, *hexp* zuordnet. In der Formel $1 \leq 2$ gilt für die Relation $\leq$ die Regel $c$, die in der folgende Tabelle dargestellt ist:

| $c_{symb}$ | $c_{type}$ | $c_{ltype}$ | $c_{rtype}$ | $c_{lexp}$ | $c_{rexp}$ | $c_{hexp}$ |
|---|---|---|---|---|---|---|
| (″\leq″) | ″Relation″ | ″Integer″ | ″Integer″ | ″lhs″ | ″rhs″ | ″relation″ |

Wenn der linke oder rechte Unterbaum fehlt, wird für den entsprechenden Typ ″NONE″ gesetzt und für den entsprechenden Ausdruck ″.″. Bei Variablen und Konstanten, deren Wert am Ende immer angegeben werden muss, setzen wir *hexp* ← ″VALUE″ und *htype* enthält den Wert, der dann ausgegeben wird.

**Beispiel** Für die Darstellung von $\sqrt{x}$ muss der Baum $[\texttt{\textbackslash sqrt}[x]_{\texttt{Var}}]_{\texttt{Float}}$ in den Ausdruck

```
$3134.type=Sqrt
$3134.radicand=3135
$3135.type=Var
VALUE($3135)=x
```

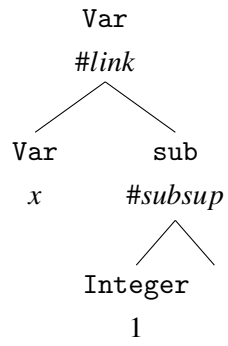
umgewandelt werden. Die dazu benötigten Regeln sind in dieser Tabelle dargestellt:

| *symb* | *type* | *ltype* | *rtype* | *htype* | *lexp* | *rexp* | *hexp* |
|---|---|---|---|---|---|---|---|
| (″x″) | ″Var″ | ″NONE″ | ″NONE″ | ″x″ | ″NONE″ | ″NONE″ | ″VALUE″ |
| (″\sqrt″) | ″Sqrt″ | ″NONE″ | ″Var″ | ″.″ | ″.″ | ″radicand″ | ″.″ |

### Überspringen von Knoten

Vergleichen wir die Formel $x_1$ in Concise mit ihrer Darstellung als Binärbaum:

```
$3134.type=Script
$3134.formula=3135
$3134.sub=3137
$3135.type=Var
$3137.type=Integer
VALUE($3135)=str:x
VALUE($3137)=str:1
```




Die Unterausdrücke des Ausdrucks mit dem Typ `Script` sind in dem Baum nicht in demselben Knoten enthalten. Bei der Umwandlung muss also ein Knoten gewissermaßen übersprungen werden. In den Regeln setzen wir dafür bei einem Knoten, der übersprungen werden soll, *htype* auf ″#″. Wenn der linke bzw. rechte Unterbaum eines Knotens übersprungen werden soll, setzen wir *lexp* bzw. *rexp* auf ″#″. Für die Formel $x_1$ brauchen wir also diese zwei Regeln:

| *symb* | *type* | *ltype* | *rtype* | *htype* | *lexp* | *rexp* | *hexp* |
|---|---|---|---|---|---|---|---|
| (″#link″) | ″Script″ | ″Var″ | ″sub″ | ″.″ | ″formula″ | ″#″ | ″.″ |
| (″#subsup″) | ″sub″ | ″Integer″ | ″NONE″ | ″#″ | ″sub″ | ″NONE″ | ″.″ |

### Vorarbeiten

Bevor unsere Bäume in Concise-Ausdrücke verwandelt werden können, müssen alle in Abschnitt 4.2 beschriebenen Nachbearbeitungen (Umgruppierung von Ziffern, nachträgliche Erzeugung von Operatoren usw.) durchgeführt werden. Weiters müssen die Bäume noch so umgewandelt werden, dass die Besonderheiten der Darstellung in Concise berücksichtigt sind, d. h. Summen und mehrfache Relationen müssen in die oben vorgestellte Form gebracht werden. Erst dann können die Regeln für die Erzeugung der Ausdrücke richtig angewendet werden.

## 6.2.2 Die Algorithmen zur Umwandlung

### Summen

Wenn wir Summen in die richtige Form bringen, müssen wir zuerst die Summationsoperatoren (also im klassischen Fall $+$ und $-$) zu rechtsassoziativen Operatoren machen. Sind sie linksassoziativ, lässt sich die gewünschte Darstellung als Liste nämlich nicht erzeugen. Andererseits dürfen wir die Operatoren nicht einfach als rechtsassoziativ annehmen, denn wenn wir mit $\dot{-}$ ein rechtsassioziatives Minus bezeichnen, dann gilt etwa

$$3\dot{-}2\dot{-}1 = 3\dot{-}(2\dot{-}1) = 3-1 = 2 \neq 3-2-1 = (3-2)-1 = 1-1 = 0,$$

d. h. Subtraktionsoperatoren sind rein linksassoziativ. Die Änderung der Assoziativität geht so: Wir betrachten den Knoten $n$ mit den Unterbäumen $l$ und $r$. Enthält $n$ oder der linke Unterbaum keinen Summations-Operator, wird das Verfahren einfach auf beide Unterbäume angewendet. Sonst wird ein neuer Knoten $n$ erzeugt, der dieselben Tokens wie $l$ enthält. Sein linker Unterbaum wird $\texttt{left}(l)$ und sein rechter Unterbaum hat dieselben Tokens wie der alte Knoten $n$ und die Unterbäume $\texttt{right}(l)$ und $r$. Zum Schluss muss das ganze Verfahren auch noch auf den neuen Knoten $n$ angewendet werden. Es ist in Algorithmus 6.2.1 beschrieben und wird von der Funktion `make_right_ass` auf einen Knoten und einen Vektor mit den Summationssymbolen angewendet.[1]

**Algorithmus 6.2.1** Änderung der Assoziativität eines linksassoziativen Operators

Eingabe: im Knoten $n$ sollen alle Operatoren, die im Vektor $s$ enthalten sind, rechtsassoziativ gemacht werden.

Ausgabe: Knoten n

Sonstige Variable: Bäume $L, R$, Zahlen $i, j$

**0** Setze $l \leftarrow \texttt{left}(n), r \leftarrow \texttt{right}(n)$ und $i \leftarrow 0, j \leftarrow 0$.

**1** Falls $\texttt{getName}(n,0) \neq s_i$, setze $i \leftarrow i+1$ und wiederhole den Schritt.

**2** Falls $l = \texttt{NULL} \vee i \geq \texttt{size}(s)$, weiter bei Schritt 5.

**3** Falls $\texttt{getName}(l,0) \neq s_i$, setze $j \leftarrow j+1$ und wiederhole den Schritt.

**4** Falls $j < \texttt{size}(l)$, weiter bei Schritt 6.

**5** Setze $l \leftarrow \texttt{make\_right\_ass}(l,s)$, $r \leftarrow \texttt{make\_right\_ass}(r,s)$ und der Algorithmus ist fertig.

**6** Erzeuge neue Knoten $L, R$.
Setze $L \leftarrow \texttt{left}(l), \texttt{setName}(R, \texttt{getName}(n,0),0), \texttt{left}(R) \leftarrow \texttt{right}(l), \texttt{right}(R) \leftarrow r$.
Erzeuge einen neuen Knoten $n$ mit $\texttt{setName}(n, \texttt{getName}(l,0),0)$,
$\texttt{left}(n) \leftarrow L, \texttt{right}(n) \leftarrow R$.

**7** Setze $n \leftarrow \texttt{make\_right\_ass}(n,s)$ und der Algorithmus ist fertig.

Danach müssen wir vor jedem Unterbaum, der eine Summe darstellt, einen `SignedSum`- und einen `ExpLink`-Knoten einfügen. Die Hauptschwierigkeit dabei besteht darin zu erkennen, wo der Summenbaum anfängt. Wenn ein Knoten ein Summationszeichen (z. B. $+$ oder $-$) enthält und sein Eltern-Knoten kein Summationszeichen enthält, dann ist er der Anfang eines Summen-Baumes. In diesem Fall müssen darüber noch die zwei erwähnten Knoten eingefügt werden.

Die genaue Vorgehensweise sieht so aus: In einem Vektor $s$ sind alle Summationszeichen gespeichert. Wir bearbeiten den Knoten $n$ und verwenden die boolesche Variable $t$, die wir am Beginn auf

[1] Es gibt natürlich auch ein analoges Verfahren, mit dem rechtsassoziative Operatoren linksassoziativ gemacht werden können. Da rechtsassoziative Operatoren jedoch eher selten sind, wird es in der Praxis nie benötigt und ist daher hier auch nicht beschrieben.

1 setzen, um herauszufinden, ob der Knoten der Anfang der Summe ist. Wenn der Knoten Teil einer Summe ist, dann gibt es einen Index $i$ mit $\texttt{getName}(n,0) = s_i$. Ist außerdem $t = 1$, dann müssen wir vor $n$ die beiden Knoten einfügen. Falls $n$ ein Summen-Knoten ist, setzen wir $t \leftarrow 0$, weil seine Unterbäume dann nicht mehr der Anfang einer Summe sein können. Dann wird das Verfahren auf $\texttt{left}(n)$ und $\texttt{right}(n)$ angewendet. Das Vorgehen ist in Algorithmus 6.2.2 zusammengefasst und verwendet die Funktion `make_sum`, die das Verfahren auf einen Baum, einen Vektor und eine boolesche Variable anwendet.

**Algorithmus 6.2.2** Summen

Eingabe: der Baum $n$ enthält die Formel, im Vektor $s$ sind alle Summationszeichen gespeichert und die boolesche Variable $t$ gibt an, ob $n$ der Anfang einer Summe ist.

Ausgabe: im Baum $n$ sind vor jeder Summe zwei zusätzliche Knoten eingefügt.

Sonstige Variable: Bäume $l, r, \texttt{el}, \texttt{ssl}$, boolesche Variable $f$, Zahl $i$

**0** Setze $f \leftarrow 0, l \leftarrow \texttt{left}(n), r \leftarrow \texttt{right}(n), i \leftarrow 0$.

**1** Falls $s_i = \texttt{getName}(n,0)$, setze $f \leftarrow 1$, sonst $i \leftarrow i+1$ und wiederhole den Schritt.

**2** Setze $l \leftarrow \texttt{make\_sum}(l, s, \neq f)$ und $r \leftarrow \texttt{make\_sum}(r, s, \neq f)$.

**3** Falls $f = 0 \vee t = 0$, ist der Algorithmus fertig.

Erzeuge neue Knoten `el` und `ssl`.

Setze $\texttt{setName}(\texttt{ssl}, ''\texttt{\#link}'')$, $\texttt{setType}(\texttt{ssl}, ''\texttt{SignedSum}'')$, $\texttt{right}(\texttt{ssl}) \leftarrow n$ und $\texttt{setName}(\texttt{el}, ''\texttt{\#link}'')$, $\texttt{setType}(\texttt{el}, ''\texttt{ExpLink}'')$, $\texttt{right}(\texttt{el}) \leftarrow \texttt{ssl}$.

Erzeuge einen neuen Knoten $n$, setze $n \leftarrow \texttt{el}$ und der Algorithmus ist fertig.

### Mehrfache Relationen

Das „Aufspalten“ von mehrfachen Relationen funktioniert ähnlich wie die Umwandlung von Summen, ist aber etwas komplizierter. Hier reicht es nämlich nicht, dass (nach Ändern der Assoziativität) einfach nur neue Knoten eingefügt werden, sondern es müssen auch neue Bäume aus Teilen von schon vorhandenen Bäumen erzeugt werden. In Abbildung 6.1 ist zu sehen, was alles geändert werden muss. Die Relationssymbole müssen in der Syntax auch als rechtsassoziative Operatoren behandelt werden.

Wir betrachten einen Knoten $n$ mit den Unterbäumen $l$ und $r$. Die boolesche Variable $t$ gibt uns an, ob $n$ der erste Knoten in einem Relationen-Baum ist, am Anfang setzen wir daher $t \leftarrow 1$. In dem Vektor $s$ sind alle Relationssymbole gespeichert. Damit $n$ Teil einer mehrfachen Relation sein kann, muss es Indices $i$ und $j$ geben, sodass $\texttt{getName}(n,0) = s_i$ und $\texttt{getName}(r,0) = s_j$ gilt. Ist dies nicht der Fall, dann wenden wir das Verfahren rekursiv auf $l$ und $r$ an. Sonst prüfen wir, ob $n$ der oberste Knoten eines Relationen-Baumes ist. Das ist dann der Fall, wenn $t$ den Wert 1 hat. Wir erzeugen

dann einen neuen Knoten, dessen Unterbäume $l$ und `left`$(r)$ sind und der dasselbe Relationssymbol wie $n$ enthält. Der Knoten $n$ erhält dann diesen Knoten als neuen linken Unterbaum, als Token `#link` und als Typ `Chain`. Das Verfahren wird dann rekursiv auf $r$ angewendet, wobei $t \leftarrow 0$ gesetzt werden muss. Der neue rechte Unterbaum von $n$ wird dann ein Knoten, der das Token `#link`, den Typ `ExpLink` und den rechten Unterbaum $r$ hat. Ist jedoch $t = 0$, so befindet sich $n$ weiter unten in einem Relationen-Baum. Der neue Typ von $n$ wird dann `ExpLink` und das neue Token `#link`. Der neue rechte Unterbaum von $n$ wird genau so erzeugt wie im ersten Fall. Für den linken Unterbaum erzeugen wir einen neuen Knoten, der dasselbe Relationssymbol und denselben Typ wie $n$ hat. Sein rechter Unterbaum ist `left`$(r)$. Der linke Unterbaum ist ein neuer Knoten mit dem Token `#link` und dem Typ `Dummy`, der $l$ als linken Unterbaum hat. Dieses Verfahren ist in Algorithmus 6.2.3 zusammengefasst. Die Funktion `split` wendet es auf einen Knoten $n$, einen Vektor $s$ und eine boolesche Variable $t$ an.

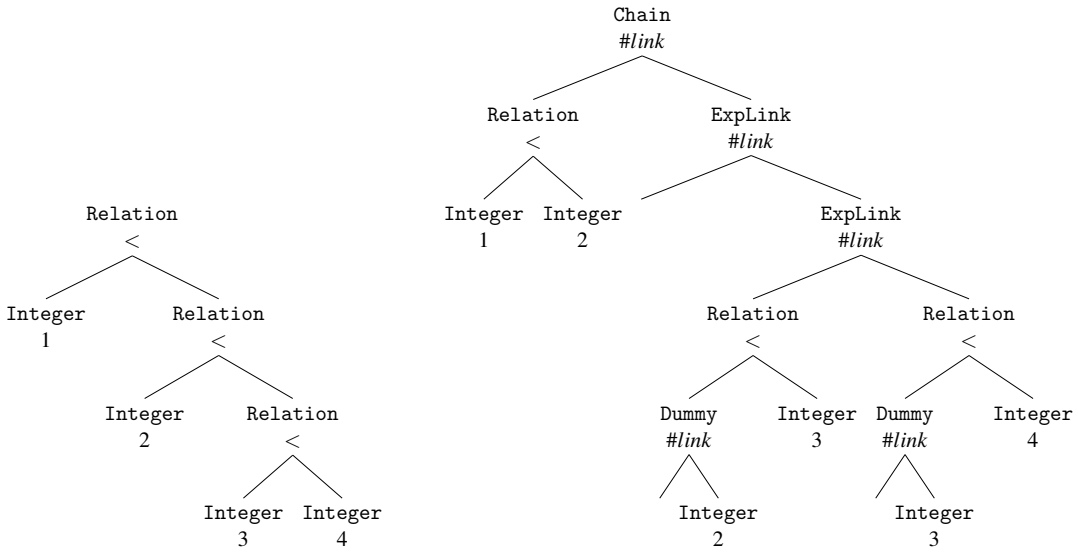



Abbildung 6.1: Der Baum zur Formel $1 < 2 < 3 < 4$ vor und nach der Umwandlung.

### Erzeugung von Concise-Ausdrücken

Wir speichern die Ausdrücke in einem Vektor *exp* ab. Für die Werte verwenden wir eine Symboltabelle *val*, deren Einträge die Werte als Strings und ihre Nummern als Zahlen enthalten. Der Eintrag für `VALUE($5)=str:x` besteht also aus dem String $''$`x`$''$ und der Zahl 5. Die Regeln für die Umwandlung sind in dem Vektor *exprules* enthalten. Die Zahl `index` ist die Nummer des Ausdrucks, der gerade erzeugt wird. Da diese Nummer sich jedoch ändern kann, wenn das Verfahren auf Unterbäume angewendet wird, speichern wir den Wert, den *index* am Beginn der Verarbeitung eines Knotens hat, immer in der Variablen *num* ab. Wir betrachten einen Knoten $n$. Seine Unterbäume sind $l$ und $r$ mit den Typen $lt$ und $rt$. (Falls ein Unterbaum fehlt, ist der zugehörige Typ wie üblich $''$`NONE`$''$.) Im Vektor $s$ speichern wir die Tokens von $n$ ab. Jetzt müssen wir eine Regel $e$ mit $e_{symb} = s, e_{ltype} = lt, e_{rtype} = rt$

**Algorithmus 6.2.3** Aufspalten von mehrfachen Relationen

Eingabe: der Baum $n$ enthält die Formel, der Vektor $s$ enthält alle Relationssymbole und der boolesche Wert $t$ gibt an, ob $n$ der oberste Knoten in einem Relationen-Baum ist.

Ausgabe: im Baum $n$ sind alle Relationen aufgespalten.

sonstige Variable: Bäume $l, r, L, LL$, boolesche Variable $f, rf$, Zahl $i$

**0** Setze $l \leftarrow \texttt{left}(n), r \leftarrow \texttt{right}(n), f \leftarrow 0, rf \leftarrow 0, i \leftarrow 0$.

**1** Falls $s_i = \texttt{getName}(n,0)$, setze $f \leftarrow 1$, sonst $i \leftarrow i+1$ und wiederhole den Schritt.

**2** Falls $r = 0$, weiter bei Schritt 3.

Falls $s_i = \texttt{getName}(r,0)$, setze $rf \leftarrow 1$, sonst $i \leftarrow i+1$ und wiederhole den Schritt.

**3** Falls $f = 1 \wedge (t = 0 \vee rf = 1)$ weiter bei Schritt 4.

Setze $l \leftarrow \texttt{split}(l,s,t)$, $r \leftarrow \texttt{split}(r,s,t)$ und der Algorithmus ist fertig.

**4** Falls $t = 1 \vee rf = 1$ weiter bei Schritt 5.

Erzeuge einen neuen Knoten $LL$ und setze $\texttt{setType}(LL, ''\texttt{Dummy}'')$, $\texttt{setName}(LL, ''\texttt{\#link}'')$, $\texttt{right}(LL) \leftarrow l$, $\texttt{left}(n) \leftarrow LL$ und der Algorithmus ist fertig.

**5** Erzeuge einen neuen Knoten $L$ und setze $\texttt{setName}(L, \texttt{getName}(n)), \texttt{setType}(L, \texttt{getType}(n))$, $\texttt{left}(L) \leftarrow l, \texttt{right}(L) \leftarrow \texttt{left}(r)$ und wende das Verfahren auf $r$ an: $r \leftarrow \texttt{split}(r,s,\neg f)$.

Setze $\texttt{setName}(n,0,''\texttt{\#link}''), \texttt{setType}(n,0,''\texttt{ExpLink}''), \texttt{left}(n) \leftarrow L, \texttt{right}(n) \leftarrow r$.

Der Algorithmus ist jetzt fertig.

---

suchen. Falls $e_{hexp} = ''\texttt{VALUE}''$ ist, überprüfen wir, ob $n$ eine Zahl darstellt. Wenn das der Fall ist, machen wir $v$ zu dem String, der diese Zahl darstellt. Ansonsten setzen wir $v \leftarrow e_{htype}$. Wenn $v$ im Vektor *val* noch nicht vorkommt, fügen wir einen Eintrag mit dem String *val* und der Nummer *num* ein. Gibt es jedoch schon einen Eintrag mit dem String $v$, dann müssen wir alle Ausdrücke suchen, die Unterausdrücke mit der Nummer *num* enthalten. Die Nummern dieser Unterausdrücke müssen wir dann durch die Nummer des älteren Eintrags von *val* ersetzen. So ist es unmöglich, dass Ausdrücke mit demselben Wert mehrfach vorkommen. Der Knoten $n$ ist damit fertig verarbeitet.

Jetzt beschäftigen wir uns mit den Fällen, in denen $e_{hexp} \neq ''\texttt{VALUE}''$ ist: Ist $e_{htype} \neq ''\#''$, kann der Knoten ganz normal verarbeitet werden, d. h. wir hängen an den Vektor *exp* einen Ausdruck mit der Nummer *num* und dem Typ $e_{type}$ an. Wenn wir $exidx \leftarrow \texttt{size}(ex) - 1$ setzen, dann ist dieser Ausdruck $exp_{exidx}$. Ist $e_{hexp} \neq ''.''$, müssen wir noch $e_{htype}$ und $e_{hexp}$ an $(exp_{exidx})_{type}$ bzw. $(exp_{exidx})_{symb}$ anhängen. Wenn jedoch $e_{htype} = ''\#''$ ist, müssen wir *exidx* so wählen, dass $(ex_{exidx})_{num} = index - 2$ ist, weil wir dann sozusagen um einen Ausdruck zurückgehen.

Jetzt muss das Verfahren auf die Unterbäume von $n$ angewendet werden, sofern diese vorhanden sind: Ist $e_{lexp} \neq ''\#''$, wird an *ex* ein Ausdruck mit der Nummer $index - 1$ und dem Typ $e_{lexp}$ angehängt. Andernfalls wird $index \leftarrow index - 1$ gesetzt, wenn $e_{rexp} \neq ''\#''$ ist. Dann wird das Verfahren

rekursiv auf $l, index, ex$ und $v$ angewendet. Für $r$ ist die Vorgehensweise analog. Dieses Verfahren ist in Algorithmus 6.2.4 zusammengefasst und wird von der Funktion `concise` auf einen Baum, einen Vektor mit den Regeln, die Adresse einer Zahl, einen Vektor für die Ausdrücke und eine Symboltabelle für die Werte angewendet.

## 6.3 Record Sheets

Diese Art der Darstellung von Ausdrücken ist aber noch ein wenig zu unpraktisch. Viel wichtiger sind die sogenannten *Record Sheets*, in denen mathematische Inhalte aufgezeichnet werden können. Zum Vergleich ist hier die Formel $a = b$ in der Darstellung, die wir oben kennengelernt haben, und als Record Sheet:

```
$3134.type=Binary
$3134.relation=Equal
$3134.lhs=3135
$3134.rhs=3136
$3135.type=Var
$3136.type=Var
VALUE($3135)=str:a
VALUE($3136)=str:b
```

```
testUser.Example_aeqb=Example_aeqb::English,Expressions
Example_aeqb(Binary)>
  relation(Equal)>
  <
  lhs(Var)>
    name(String)> [EXT]a[/EXT]
  < <
  rhs(Var)>
    name(String)> [EXT]b[/EXT]
  < <
<
```

Ein Record Sheet enthält noch zusätzliche Informationen wie die Sprache oder den Namen des Users. Wir beschäftigen uns aber im Weiteren nur mit der Darstellung der Formeln. (Ob diese einem englischen oder chinesischen Text entnommen sind, ist dabei egal.) Hier ist die Baumstruktur besser erkennbar, jede Formel ist in Wirklichkeit ein zu einem Baum äquivalenter Klammerausdruck, wobei die Klammersymbole > und < sind (wie einfache französische Anführungszeichen). Die Knoten der Baumstruktur haben immer einen Namen und einen Typ. Für den obersten Knoten muss ein Name vorgegeben werden (hier `Example_aeqb`), sonst sind die Namen immer die Namen der Unterausdrücke. Die Typen stehen nach den Namen in runden Klammern, es sind immer die Typen der Unterausdrücke. Bei Zahlen, Variablen und Konstanten (die wir vorhin extra abspeichern müssen) gibt es `name`-Knoten mit dem Typ `String`, und der eigentliche Wert wird dann zwischen `[EXT]` und `[/EXT]` angegeben. So müssen diese Werte nicht außerhalb gespeichert werden sondern sind im Gegensatz zu vorher direkt in das Record Sheet integriert. Die Umwandlung der Ausdrücke als Record Sheets ist nicht sehr kompliziert, weil es sich nur um eine andere Darstellungsart handelt. Es muss also eigentlich gar nichts umgewandelt werden. Die einzige zusätzliche information ist der Name der Formel, der aber frei gewählt werden kann (hier `Example_aeqb`).

Wir gehen davon aus, dass die Ausdrücke alle in dem Vektor *exp* und die Werte von Konstanten und Variablen in dem Vektor *val* gespeichert sind. Wir behandeln jetzt einen Ausdruck $e$. Zuerst

müssen seine Typen und ihre Namen ausgegeben werden, d. h. es werden Zeilen der Form

$$t \quad ''(\,'' \quad n \quad '')>''$$

in die Ausgabe geschrieben. Dabei ist am Anfang der String $t$ der Typ des Ausdrucks, der vorgegeben werden muss, und $n = (e_{name})_0$. Für $i = 1, \ldots, \texttt{size}(e_{type}) - 1$ ist dann $t = (e_{type})_i$ und $n = (e_{name})_i$. Danach wird noch eine schließende Klammer

$$''<''$$

in die Ausgabe geschrieben. Jetzt werden die Unterausdrücke ausgegeben. Dazu suchen wir für $i = 0, \ldots, \texttt{size}(e_{subex})$ immer den Index $j$ mit $exp_j = (e_{subex})_i$. Dann wird geprüft, ob es einen Index $k$ gibt, sodass $(exp_j)_{num}$ die Nummer von $val_k$ ist. Ist dies der Fall, müssen die Zeilen

$$(e_{attr})_i \quad ''(\,'' \quad (exp_j)_{name_0} \quad '')>''$$

$$''\texttt{name(String)} > \ [\texttt{EXT}]'' \quad (val_{index})_{name} \quad ''[/\texttt{EXT}]''$$

$$''<<''$$

ausgegeben werden. Ansonsten wird das Verfahren auf den Ausdruck $exp_j$ angewendet, wobei dann als Typ für den Ausdrück $(e_{attr})_i$ gewählt wird. Das Verfahren ist in Algorithmus 6.3.1 zusammengefasst und wird von der Funktion `recordsheet` auf einen Ausdruck, einen Vektor von Ausdrücken, eine Symboltabelle und einen String angewendet.

**Algorithmus 6.2.4** Erzeugung von Concise-Ausdrücken

Eingabe: aus dem Baum $n$ werden Concise-Ausdrücke erzeugt, der Vektor *expru* enthält die Regeln, die dazu benötigt werden, die Adresse der Zahl *index* ist die Nummer des aktuellen Ausdrucks, die Ausdrücke werden dann im Vektor *ex* gespeichert und die Werte von Variablen und Konstanten im Vektor *val*.

Sonstige Variable: Bäume $l, r$, Zahlen $i, num, exidx$, Strings $lt, rt, s$

**0** Setze $l \leftarrow \texttt{left}(n), r \leftarrow \texttt{right}(n)$, $index \leftarrow index + 1, num \leftarrow index$, $lt \leftarrow ''\texttt{NONE}'', rt \leftarrow ''\texttt{NONE}''$.

Falls $l \neq \texttt{NULL}$, setze $lt \leftarrow \texttt{type}(l)$, falls $r \neq \texttt{NULL}$, setze $rt \leftarrow \texttt{type}(r)$. Setze $s \leftarrow \texttt{getname}(n)$, $i \leftarrow 0$.

**1** Falls $(expru_i)_{symb} = s \wedge (expru_i)_{ltype} = lt \wedge (expru_i)_{rtype} = rt$ setze $e \leftarrow expru_i$, sonst $i \leftarrow i + 1$ und wiederhole den Schritt.

**2** Falls $e_{hexp} \neq ''\texttt{VALUE}''$, weiter bei Schritt 5.

Wenn $s_0$ eine Ziffer oder ein Dezimaltrennzeichen ist oder $n$ ein `#link`-Baum ist, dessen Einträge nur Ziffern sind, wird *val* der String, der die Formel zu $n$ enthält. Sonst wird $val \leftarrow e_{htype}$ gesetzt.

**3** Gibt es in der Symboltabelle $v$ schon einen Eintrag für *val*, setze $i \leftarrow 0$ und weiter bei Schritt 4. Sonst wird an den Vektor *expressions* ein Ausdruck mit der Nummer *num* und dem Typ $e_{type}$ angehängt.

**4** Ersetze in $ex_i$ alle Nummern von Unterausdrücken, die den Wert *num* haben, durch die Nummer des Eintrags von *val* in $v$. Setze $i \leftarrow i + 1$ und wiederhole den Schritt, solange $i < \texttt{size}(ex)$ ist. Der Algorithmus ist dann fertig.

**5** Ist $e_{htype} = ''\#''$, setze $exidx \leftarrow 0$ und weiter bei Schritt 6.

Hänge an *ex* einen Ausdruck mit der Nummer *num* und dem Typ $e_{type}$ an. Setze $exidx \leftarrow \texttt{size}(ex) - 1$. Ist $e_{hexp} \neq ''.''$, $\texttt{push}(ex_{exidx})_{num}, e_{num})$ und $\texttt{push}(ex_{exidx})_{type}, e_{htype})$.

**6** Ist $(ex_{exidx})_{num} \neq index - 2$, setze $exidx \leftarrow exidx + 1$ und wiederhole den Schritt.

**7** Falls $l = 0$, weiter bei Schritt 8. Ist $e_{rexp} \neq ''\#''$, setze $\texttt{push}((ex_{exidx})_{num}, index + 1)$ und $\texttt{push}((ex_{exidx})_{type}, e_{lexp})$. Ist ansonsten $e_{rexp} = ''\#''$, setze $index \leftarrow index - 1$.

Wende das Verfahren auf den linken Unterbaum an: $\texttt{concise}(l, index, ex, v)$.

**8** Falls $r = 0$, ist der Algorithmus fertig. Ist $e_{lexp} \neq ''\#''$, setze $\texttt{push}((ex_{exidx})_{num}, index + 1)$ und $\texttt{push}((ex_{exidx})_{type}, e_{rexp})$. Ist ansonsten $e_{lexp} = ''\#''$, setze $index \leftarrow index - 1$.

Wende das Verfahren auf den rechten Unterbaum an: $\texttt{concise}(r, index, ex, v)$; der Algorithmus ist jetzt fertig.

**Algorithmus 6.3.1** Ausgabe von Record Sheets

Eingabe: Der Ausdruck $e$ soll ausgegeben werden, in dem Vektor $exp$ sind die anderen Ausdrücke abgespeichert, die Symboltabelle $val$ enthält die Werte von Variablen und Konstanten und der String $type$ ist der Typ des Ausdrucks.

Sonstige Variable: Zahlen $i, j, k, index$

**1** $\texttt{print}(type)\ \texttt{print}(''('')\ \texttt{print}((e_{name})_0)\ \texttt{print}()\ >)$

**2** Setze $i \leftarrow 1$.

**3** $\texttt{print}((e_{type})_i)\ \texttt{print}(''('')\ \texttt{print}((e_{name})_0)\ \texttt{print}()\ >)$
Wenn $i < \texttt{size}(e_{type})$, setze $i \leftarrow i+1$ und wiederhole den Schritt.

**4** $\texttt{print}(''<'')$ und setze $i \leftarrow 0, j \leftarrow 0$.

**5** Setze $j \leftarrow j+1$, solange $(exp_j)_{num} \neq (e_{subex})_i$.

**6** Setze $index \leftarrow -1$ und $k \leftarrow 0$.

**7** Setze $k \leftarrow k+1$, solange $k < \texttt{size}(val)$ gilt.
Ist $(exp_j)_{num} = (val_k)_{type}$, setze $index \leftarrow k$ und weiter bei Schritt 8.

**8** Ist $index \geq 0$: $\texttt{print}((e_{attr})_i)\ \texttt{print}(''('')\ \texttt{print}((exp_j)_{name_0})\ \texttt{print}('')>'')$
$\texttt{print}(''\texttt{name}(\texttt{String}) > \ [\texttt{EXT}]'')\ \texttt{print}((val_{name})_{index})\ \texttt{print}(''[/\texttt{EXT}]'')\ \texttt{print}(''<<'')$

**9** Ist $index < 0$, wende das Verfahren auf den nächsten Ausdruck an:
$\texttt{recordsheet}(exp_j, exp, val, (e_{attr})_i)$

**10** Falls $i < \texttt{size}(exp)$, setze $i \leftarrow i+1$ und zurück zu Schritt 5.

**11** $\texttt{print}(''<'')$ und der Algorithmus ist fertig.

# Conclusio

In dieser Arbeit wurde die Funktionsweise eines Parsers beschrieben, der alle für Menschen *eindeutig* lesbaren Formeln lesen kann (sofern keine Arrays und Matrizen darin vorkommen). Folgende Problemfälle können noch nicht behandelt werden:

- Ineinander verschachtelte Betragsstriche u. Ä.

  Es gibt keine Möglichkeit, rein syntaktisch immer zu entscheiden, ob ein Betragsstrich eine öffnende oder schließende Klammer ist, Beispiel $|a|b|c|$.

- Mehrdeutige Rangordnung von Operatoren

  In manchen Formeln ist die Rangordnung der Operatoren nur klar, wenn die Bedeutung bekannt ist, Beispiel $a > b, c > d$.

- Mehrdeutige semantische Kategorien

  Oft kann eine Formel oder ein Teil einer Formel in mehrere semantische Kategorien fallen und die richtige Kategorie ergibt sich nur aus dem Kontext, Beispiel $(0, 1)$.

Es gibt für diese Probleme verschiedene Lösungsansätze. Bei Formeln mit verschachtelten Betragsstrichen kann es sehr schwer bis unmöglich sein, *alle* Lesearten zu behandeln. Eine Möglichkeit besteht darin, die Formel von links nach rechts so zu lesen, dass immer zwei Betragsstriche, zwischen denen eine vollständige Formel steht, als zusammengehörig aufgefasst werden:

$$\underbrace{|a|}\ b\ \underbrace{|c|} \quad \text{statt} \quad \underbrace{|a\ \underbrace{|b|}\ c|}$$

Dazu werden allerdings semantische Informationen benötigt, damit entschieden werden kann, ob die Formelteile zwischen den Strichen vollständige Formeln sind.

Ist die Rangordnung von Operatoren nicht eindeutig, kann es notwendig sein, dass sie für jede einzelne Formel eigens festgelegt werden muss. Wenn das automatisch erfolgen soll, werden auch hier wieder Informationen über die Semantik gebraucht. Besonders kompliziert wird es, wenn sich die Rangordnung eines Operators innerhalb einer Formel ändert.

Mehrdeutige semantische Kategorien werden meist schon durch einen minimalen Kontext eindeutig, wie man an den Beispielen $x \in (0, 1)$ und $(0, 1) \in \mathbb{R}^2$ sieht. Damit sie richtig gelesen werden können, muss bei der Typbestimmung auch der Elternknoten berücksichtigt werden, d. h. die Bäume werden anders traversiert.

Alle diese Mehrdeutigkeiten haben jedoch die gute Eigenschaft, dass sie in der Praxis nur extrem selten auftreten. Daher können sie in einer ersten Version des Parsers problemlos vernachlässigt werden.[2] Für den praktischen Einsatz ist es natürlich trotzdem unerlässlich, dass der Parser auch damit umgehen kann. Weiters wird es von Nöten sein, dass auch nicht ganz korrekte Formeln gelesen werden können. Ein ganz typisches (und weit verbreitetes) Beispiel dafür ist die falsche Verwendung des vertikalen Strichs:

```
\{ x \in \mathbb R | x >0 \}
```
$\{x \in \mathbb{R} | x > 0\}$

All das würde jedoch über den Rahmen einer Diplomarbeit hinausgehen und konnte daher hier noch nicht behandelt werden.

[2] Im praktischen Einsatz bei OpenMath und Concise ist kein einziges der Probleme jemals aufgetreten. Ich habe weiters die Formeln aus dem Vorlesungsskript *Analysis und Lineare Algebra* von Arnold Neumaier (`http://www.mat.univie.ac.at/~neum/FMathL/ALA.pdf`) geparst, und auch hier kam es zu keinen Schwierigkeiten.

# Eine Implementation in C++

Hier ist die Implementation des Parsers vorgestellt, die ich beim Schreiben dieser Arbeit entwickelt habe. Einige Programmteile sind allerdings nur angedeutet und sollten für den praktischen Einsatz etwas sorgfältiger programmiert werden. Dazu gehören zum Beispiel die Such-Funktionen oder die Funktionen zum Lesen von Dateien. Auch die Fehlermeldungen sind noch nicht sehr praxistauglich. Zur Veranschaulichung der in dieser Arbeit vorgestellten Verfahren reicht diese einfache Implementation jedoch aus.

## Das Programm

Das Programm heißt einfach `interpreter` und kann mit folgenden Parametern aufgerufen werden:

`-i` *file* liest die Eingabe aus der Datei *file*; ohne diesen Parameter erfolgt die Eingabe vom Standard-Input.

`-f` *format* bestimmt das Ausgabe-Format *format*; Möglichkeiten: `concise`, `expression`, `openmath`.

`-t` *texfile* liest die LaTeX-Informationen aus der Datei *texfile*.

`-s` *synfile* liest die Syntax-Informationen aus der Datei *synfile*.

`-d` *semfile* liest die Sematik-Informationen aus der Datei *semfile*.

`-r` ersetzt alle LaTeX-Formeln der Eingabe durch Formeln im Ausgabe-Format; wenn als Eingabe eine Datei verwendet wird, wird der Inhalt der Datei, der keine Formeln darstellt, unverändert ausgegeben.

Es werden also insgesamt drei Konfigurationsdateien gebraucht, in denen die für die Verarbeitung nötigen Informationen enthalten sind. In diese Dateien können mit %-Zeichen zeilenweise Kommentare geschrieben werden, d. h. nach % wird dann bis zum Zeilenende alles ignoriert. In der Datei mit den LaTeX-Informationen wird einfach nach jedem Token mit spezieller Bedeutung der Typ angegeben, z. B.

`texfile.txt`

```
_               SUB
^               SUP
{               LCUR
}               RCUR
```

```
\big            SIZE
\mathbb         FONT
\displaystyle   STYLE
\choose         TEXOP
\text           TEXT
\hat            ACC
\sqrt           SQRT
\frac           FRAC
```

In der Datei mit den Syntax-Informationen muss zu jedem Token, das kein Name ist, der Typ, die Rangordnung und die Assoziativitär (`l` oder `r`) angegeben werden. Die letzten beiden Informationen sind jedoch nur bei Operatoren relevant. Beispiel:

symfiile.txt

```
(           LBRACK
)           RBRACK
[           NBRACK
]           NBRACK
|           VERT
\sqrt       CMD
\frac       CMD
#subsup     CMD
%
% Operatoren:
%
+           OP 1      l
-           OP 1      l
*           OP 2      l
/           OP 2      l
:           OP 0      l
.           OP 0      l
=           OP 0      r
<           OP 0      r
>           OP 0      r
#link       OP 1024   r
```

Die Dateien mit den semantischen Informationen haben eine etwas kompliziertere Syntax. Sie enthalten nämlich nicht nur Typen von Tokens, sondern es muss mit Hilfe von speziellen Befehlen noch bekanntgegeben werden, um welche Informationen es sich handelt:

`#num` $n$ Der String $n$ ist der Typ, den Ziffern haben.

`#dec` $d$ Der String $d$ wird als Dezimaltrennzeichen verwendet.

`#seq` $n\ s_0 \ldots s_{n-1}$ Die Zeichenfolge $s_0 \ldots s_{n-1}$ wird durch einen einzigen Knoten ersetzt.

`#post` *p* Der String *p* ist ein Postfix-Operator.

`#lop` *t p* Alle Knoten, deren Typ der String `t` ist, werden zu linksassoziativen Operatoren mit Rangordnung `p` gemacht.

`#rop` *t p* Alle Knoten, deren Typ der String `t` ist, werden zu rechtsassoziativen Operatoren mit Rangordnung `p` gemacht.

`#rule` *n* $s_0 \ldots s_{n-1}$ $t_h$ $t_l$ $t_r$ Es gibt eine semantische Kategorie $((s_0,\ldots,s_{n-1}),t_h,t_l,t_r)$

`#sign` *s* Der String *s* wird von Concise als Summationsoperator verwendet.

`#rel` *s* Der String *s* wird von Concise als Relationssymbol verwendet.

`#exp` *t n* $s_0 \ldots s_{n-1}$ *type ltype rtype htype lexp rexp hexp* Eingabe einer Regel für die Erzeugung von Concise ausdrücken.

`#cd` *n* $s_0 \ldots s_{n-1}$ *ltype rtype object cd name* Eingabe einer OpenMath-Regel.

Eine Datei mit semantischen Informationen für OpenMath könnte etwa so aussehen:

`omfile.txt`

```
#num NUM
#dec  .

#seq  2 : =

#post POSTOP

#lop  OP1 1
#lop  OP2 2

% Semantische Kategorien:

#rule 1 1     NUM   NONE NONE
#rule 1 x     VAR   NONE NONE
#rule 1 y     VAR   NONE NONE
#rule 1 =     STATE VAR  VAR
#rule 1 \sqrt VAR   NONE VAR
#rule 1 \frac VAR   NUM  VAR

% OpenMath-Regeln:

#cd 1 x       NONE  NONE OMV .          x
#cd 1 y       NONE  NONE OMV .          y
#cd 1 =       VAR   VAR  OMS relation1 eq
#cd 1 \sqrt   NONE  VAR OMS arith1     root
#cd 1 \frac   NUM   VAR OMS arith1     divide
```

Eine Datei für Concise könnte so aussehen:

confile.txt

```
#num Integer
#dec  .

#seq  2 : =

#lop  OP1 1
#lop  OP2 2

#sign +
#sign -

#rel <
#rel >

% Semantische Kategorien:

#rule 1 1 Integer  NONE NONE
#rule 1 x Var  NONE NONE
#rule 1 y Var  NONE NONE
#rule 1 =               Relation Var            Var
#rule 1 \sqrt           Var      NONE           Var
#rule 1 \frac           Var      Integer        Var

% Concise-Regeln:

#exp  1 x Var NONE NONE x NONE NONE VALUE
#exp  1 y Var NONE NONE y NONE NONE VALUE
#exp  1 = Binary Var Var relation lhs rhs Equal
#exp  1 \sqrt Sqrt NONE Var . . radicand .
#exp  1 \frac Fraction Integer Var . num denom .
```

## Beispiele

### Beispiel 1

Wir betrachten eine XML-Datei, die LaTeX-Formeln enthält:

file.omdoc

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE omdoc SYSTEM "../dtd/activemath.dtd">

<omdoc>

    <section>

        <content>
            Wir setzen $y = \frac{1}{\sqrt x}$, um die Gleichung zu l&ouml;sen.
        <content>

    </section>

</omdoc>
```

In dieser Datei sollen alle LaTeX-Formeln durch OpenMath-Formeln ersetzt werden. Dazu rufen wir das Programm wie folgt auf:

```
./interpreter -r -t texfile.txt -s symfile.txt -d omfile.txt -f openmath -i file.omdoc
```

Wir erhalten dann folgende Ausgabe:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE omdoc SYSTEM "../dtd/activemath.dtd">

<omdoc>

<section>

<content>
                Wir setzen
<OMOBJ  xmlns="http://www.openmath.org/OpenMath">
<OMA cdbase="http://www.openmath.org/cd">
<OMA>
<OMS cd="relation1" name="eq" />
<OMV name="y"/>
<OMA>
<OMS cd="arith1" name="divide" />
<OMI>1</OMI>
<OMA>
<OMS cd="arith1" name="root" />
```

```
<OMV name="x"/>
</OMA>
</OMA>
</OMA>
</OMA>
</OMOBJ>
, um die Gleichung zu l\"osen.
<content>

</section>

</omdoc>
```

### Beispiel 2

Wir wollen alle Formeln aus einer TEX-Datei in Record Sheets verwandeln. Die Datei hat folgenden Inhalt:

file.tex

```
\documentclass{article}

\begin{document}

Wir setzen $y = \frac{1}{\sqrt x}$, um die Gleichung zu l\"osen.

\end{document}
```

Das Programm wird wie folgt aufgerufen:

```
./interpreter -s symfile.txt -d confile.txt -f concise -t texfile.txt -i file.tex
```

Wir erhalten dann folgende Ausgabe:

```
Example_aeqb(Binary)>
relation(Equal)>
<
lhs(Var)>
name(String)> [EXT]y[/EXT]
< <
rhs(Fraction)>
<
num(Integer)>
name(String)> [EXT]1[/EXT]
< <
denom(Sqrt)>
<
radicand(Var)>
name(String)> [EXT]x[/EXT]
```

```
< <
<
<
<
```

## Der Quelltext

In der Haupt-Datei `interpreter.cpp` sind die anderen Dateien eingefügt und in der `main`-Funktion ist die Benutzer-Schnittstelle definiert. Außerdem werden hier die Inhalte aus den Konfigurationsdateien eingelesen.

In der Datei `table.h` ist eine Klasse für Symboltabellen definiert. Für die Einträge der Tabelle gibt es die Klasse `item`, die aus einem String für das Symbol und einem beliebigen Datentypen für seinen Wert besteht. Die Klasse hat folgende Methoden:

`getName()` der Name des Symbols

`getName()` der Wert des Symbols

Die eigentliche Klasse `symtable` für die Symboltabelle besteht nur aus einem Vektor von Einträgen und hat folgende Methoden:

`insert( s )` fügt den Eintrag `s` ein; bestehende Einträge werden überschrieben.

`insert( s, t )` fügt einen Eintrag mit dem Namen `s` und dem Wert `t` ein; bestehende Einträge werden überschrieben.

`search( s )` die Nummer des Eintrags mit dem Namen `s`; gibt es keinen solchen Eintrag, wird $-1$ zurückgegeben.

`getName( i )` der Name des `i`-ten Eintrags

`getType( i )` der Wert des `i`-ten Eintrags

`size()` die Anzahl der Einträge

In der Datei `tree.h` ist eine Klasse `node` für Binärbäume definiert. Ein Knoten besteht aus einem String-Vektor für die Formelzeichen, einem String für den Typ und zwei Zeigern auf die Unterbäume. Die Klasse hat folgende Methoden:

`getLeft()` Zeiger auf den linken Unterbaum

`getRight()` Zeiger auf den rechten Unterbaum

`setLeft( l )` setzt den linken Unterbaum auf `l`

`setRight( r )` setzt den rechten Unterbaum auf `r`

`getType()` der Typ des Baumes

`setType( t )` setzt den Typ des Baumes auf `t`.

`getName()` der Vektor mit den Formelzeichen

`getName( i )` das `i`-te Formelzeichen

`setName( s )` setzt das nullte Formelzeichen aus `s`, wenn `s` ein String ist, und wenn `s` ein Vektor ist, wird der ganze Vektor mit den Formelzeichen auf `s` gesetzt.

`addName( s )` hängt `s` an den Vektor mit den Formelzeichen an.

In der Datei `rule.h` ist eine Klasse `rule` für die semantischen Kategorien definiert. Eine Kategorie besteht aus Strings für den linken und rechten Typen sowie für den Haupttypen und aus einem Vektor für die Formelzeichen. Die Klasse hat folgende Methoden:

`rule( h, l, r )` Konstruktor: der Haupttyp wird `h`, der linke Typ `l` und der rechte Typ `r`.

`rule( n, h, l, r )` Konstruktor: der Vektor mit den Formelzeichen wird `n`, der Haupttyp `h`, der linke Typ `l` und der rechte Typ `r`.

`getHead()` der Haupttyp

`getLeft()` der linke Typ

`getRight()` der rechte Typ

`getName()` der Vektor mit den Formelzeichen

`getName( i )` das `i`-te Formelzeichen

`setHead( h )` setzt den Haupttyp auf `h`.

`setLeft( l )` setzt den linken Typ auf `l`.

`setLeft( r )` setzt den rechten Typ auf `r`.

`setName( v )` setzt den Vektor mit den Formelzeichen auf `v`.

`addName( n )` hängt `n` an den Vektor mit den Formelzeichen an.

`equals( r )` vergleicht die Kategorie mit der Kategorie `r`. Falls der Vektor mit den Formelzeichen sowie der linke und rechte Typ übereinstimmen, wird 1 zurückgegeben, sonst 0.

In der Datei `openmath.h` ist die Klasse `entry` für die Einträge der Content Dictionarys definiert. Ein Eintrag besteht aus einem Vektor mit Formelzeichen, einem linken und rechten Typ, den Namen des Objekts, des Content Dictionary und des Eintrags sowie einer booleschen Variablen, die angibt, ob die Reihenfolge der Unterbäume bei der Ausgabe vertauscht werden muss. Die Klasse hat folgende Methoden:

`entry( sy, lt, rt, obj, c, n )` Konstruktor: der Vektor mit den Formelzeichen wird `sy`, der linke Typ `lt`, der rechte Typ `rt`, der Name des Objekts `obj`, der Name des Content Dictionary `c` und der Name des Eintrags `n`. Die boolesche Variable für die Reihenfolge der Unterbäume wird auf 1 gesetzt.

`equals( s, l, r )` vergleicht, ob der Vektor der Formelzeichen mit `s`, der linke Typ mit `l` und der rechte Typ mit `r` übereinstimmen. Wenn ja, wird 1 zurückgegeben, sonst 0.

`getObject()` der Name des Objekts

`getCd()` der Name des Content Dictionary

`getName()` der Name des Eintrags

`invOrder()` ändert den Wert der booleschen Variable für die Reihenfolge.

`prefix()` der Wert der booleschen Variable für die Reihenfolge

In der Datei `concise.h` sind eine Klasse `exprule` für die Regeln zur Erzeugung von Concise-Ausdrücken und eine Klasse `expression` für die Ausdrücke selbst definiert. Die Regeln bestehen aus einem Vektor mit Formelzeichen, einem Typ, einem linken Typ, einem rechten Typ, einem Haupttyp, einem linken Ausdruck, einem rechten Ausdruck, einem Hauptausdruck und einer booleschen Variable, die angibt, ob die Unterbäume bei der Umwandlung vertauscht werden müssen. Die Klasse hat folgende Methoden:

`exprule( s, t, lt, rt, ht, le, re, he, p )` Konstruktor: der Vektor mit den Formelzeichen wird `s`, der Typ `t`, der linke Typ `lt`, der rechte Typ *rt*, der Haupttyp `ht`, der linke Ausdruck `le`, der rechte Ausdruck `re`, der Hauptausdruck `he` und die boolesche Variable für die Reihenfolge `p`.

`getSymb()` der Vektor mit den Formelzeichen

`getType()` der Typ

`getLtype()` der linke Typ

`getRtype()` der rechte Typ

`getHtype()` der Haupttyp

`getLexp()` der linke Ausdruck

`getRexp()` der rechte Ausdruck

`getHexp()` der Hauptausdruck

`prefix()` gibt an, ob die Unterbäume bei der Umwandlung vertauscht werden müssen.

`equals( s, lt, rt )` vergleicht, ob der Vektor der Formelzeichen mit `s`, der linke Typ mit `lt` und der rechte Typ mit `rt` übereinstimmen. Wenn ja, wird 1 zurückgegeben, sonst 0.

Die Ausdrücke bestehen aus einer Nummer, dem Vektor der Typen, dem Vektor der Namen, dem Vektor der Nummern der Unterausdrücke und dem Vektor ihrer Namen. Die Klasse hat folgende Methoden:

`expression( n, t )` Konstruktor: die Nummer wird `n`, der erste Typ `"type"` und der erste Name `t`.

`add_type( t, n )` hängt an den Vektor mit den Typen `t` und an den Vektor mit den Namen `n` an.

`add_exp( s, a )` hängt `s` an den Vektor mit den Nummern der Unterausdrücke und `a` an den Vektor mit ihren Namen an.

`type_num()` die Anzahl der Typen

`sub_num()` die Anzahl der Unterausdrücke

`getNum()` die Nummer des Ausdrucks

`getType( i )` der `i`-te Typ

`getName( i )` der Name des `i`-ten Typs

`getAttr( i )` der Name des `i`-ten Unterausdrucks

`getSub( i )` die Nummer des `i`-ten Unterausdrucks

`setSub( i, j )` setzt den `i`-ten Unterausdruck auf `j`.

In den anderen Dateien sind verschiedenste Funktionen definiert. In der Datei `str2type.h`:

`str2type( s )` verwandelt den String `s` in den entsprechenden Typ.

In der Datei `token.h` sind die Funktionen zur Zerlegung von Strings in Tokens:

`isletter( c )` 1, wenn `c` ein Buchstabe ist, und sonst 0.

`isblank( c )` 1, wenn `c` ein Leerzeichen, Tabulator oder Zeilenumbruch ist, und sonst 0.

`token ( s, &i )` liest das nächste Token ab der `i`-ten Stelle aus dem String `s` aus.

`tokenize ( s )` verwandelt den String `s` in einen Vektor von Tokens.

In der Datei `vec.h` sind Funktionen zur Bearbeitung von Vektoren:

`append( &a, b )` hängt den Vektor `b` an `a` an.

`vec2string( v )` verwandelt den Vektor `v` in einen String, in dem die Einträge des Vektors hintereinandergereiht sind.

`search( v, x)` sucht den Eintrag `x` im Vektor `v`. Kommt `x` in `v` nicht vor, wird $-1$ zurückgegeben, sonst der Index des ersten Eintrags von `v`, der gleich `x` ist.

In der Datei `next.h` gibt es Funktionen, um die nächste Gruppe von Tokens in einem Vektor zu bestimmen:

`next_within( &v, begin, end, int &i)` die nächste Gruppe von Strings nach der `i`-ten Stelle im Vektor `v`, die von `begin` und `end` umschlossen ist. Wenn `v[i+i]` ungleich `begin` ist, besteht die Gruppe nur aus $v[i+1]$. Die Variable `i` ist nachher der Index des ersten Elements nach der Gruppe.

`next_group( &v, &i)` die nächste Klammergruppe nach dem `i`-ten Element im Vektor `v`; äquivalent zu `next_within(v,"{","}",i)`.

`next_squarebr( &v, &i)` die nächste von eckigen Klammern umschlossene Gruppe nach dem `i`-ten Element im Vektor `v`; äquivalent zu `next_within(v,"[","]",i)`.

Die Datei `tex.h` enthält die Funktionen, die für die in Kapitel 2 beschriebene LaTeX-Verarbeitung benötigt werden:

`tex_type( s )` der Typ von `s`

`setfont( &v, s )` fügt bei jedem Eintrag des Vektors `v`, der aus Buchstaben besteht, die Schriftinformation `s` hinzu. Ist `s` etwa `"\mathbb"`, dann wird aus `"N"` das neue Token `"\mathbb{N}"`.

`stylecheck( v )` verarbeitet Tokens vom Typ `FONT`, `TEXT` und `SIZE` und entfernt Tokens vom Typ `BLANK` oder `STYLE`.

`find_texop( v )` prüft, ob es im Vektor `v` einen TeX-Operator (z. B. `\choose`) gibt, der auf alle Tokens des Vektors wirkt. Wenn ja, wird 1 zurückgegeben, sonst `0`.

Im Vektor zur Formel `n \choose k` wäre das Ergebnis 1, bei `1+{n \choose k}` wäre es jedoch `0`, weil hier `\choose` nur für einen Teil der Formel relevant ist.

`texcheck( v )` macht die gesamte LaTeX-Verarbeitung des Vektors `v` und bringt ihn in die neue Darstellungsart.

Die Datei `symbol.h` enthält die Funktionen zur Bestimmung der syntaktischen Eigenschaften der Tokens:

`type_of( s )` der Typ von `s`

`precedence_of( s )` die Rangordnung des Operators `s`

`left_ass( s )` gibt 1 zurück, wenn `s` ein linksassoziativer Operator ist, sonst 0.

Die Datei `postfix.h` enthält die Funktion zur Erzeugung der Postfix-Darstellung:

`postfix( v )` bringt die Formel im Vektor `v` in Postfix-Darstellung (Algorithmus 3.3.1).

In der Datei `tree_func.h` sind die Funktionen zur Nachbearbeitung der Bäume definiert:

`parse( s, &i )` erzeugt aus dem Vektor `s` einen Binärbaum (Algorithmus 3.4.1).

`change_ass( n, s )` ändert im Baum `n` die Assoziativität von `#link` beim Typ `s` (Algorithmus 4.2.1).

`parent( t, n )` bestimmt den Eltern-Knoten von `n` im Baum `t`.

`tree2vec( t, v, w )` verwandelt den Baum `t` zurück in einen Vektor `v`; der Vektor `w` enthält die Knoten, die zu den entsprechenden Einträgen von `v` gehören.

`tree2string( n )` verwandelt den Baum `n` zurück in einen String.

`search( v, p, &f )` sucht im Vektor `v` alle Vorkommen des Vektors `p`; die Indices der Fundstellen werden in `f` gespeichert.

`replace( t, seq )` ersetzt im Baum `t` die Sequenz `seq` durch einen einzigen Knoten.

`roptree( n, t )` macht im `#link`-Baum `n` Knoten vom Typ `t` zu rechtsassoziativen Operatoren (Algorithmus 4.2.4).

`loptree( n, t )` macht im `#link`-Baum `n` Knoten vom Typ `t` zu linksassoziativen Operatoren (Algorithmus 4.2.5).

`make_r_op( n, t, p )` macht im Baum `n` alle Knoten vom Typ `t` zu rechtsassoziativen Operatoren mit der Rangordnung `p` (Algorithmus 4.2.6).

`make_l_op( n, t, p )` macht im Baum `n` alle Knoten vom Typ `t` zu linksassoziativen Operatoren mit der Rangordnung `p` (Algorithmus 4.2.7).

`is_num( n )` entscheidet, ob der Baum `n` eine Zifferngruppe darstellt; sind alle Operatoren `#link` und enhalten alle Knoten ohne Unterbäume Ziffern, wird 1 zurückgegeben, sonst 0.

`is_num( n, t )` ist die verallgemeinerte Version der vorherigen Funktion. Hier sind alle Knoten vom Typ `t` Ziffern.

`change_prec( n, t )` gruppiert im Baum `n` alle Knoten vom Typ `t` als Ziffern (Algorithmus 4.2.2).

`make_num( n, t, d )` gruppiert im Baum `n` alle Knoten vom Typ `t` als Ziffern, wobei `d` als Dezimaltrennzeichen verwendet wird (Algorithmus 4.2.3).

`rulecheck( n )` führt im Baum `n` die Typbestimmung durch (Algorithmus 4.1.1).

`openmath( n, dict )` erzeugt aus dem Baum `n` die OpenMath-Ausgabe; die Regeln dazu sind im Vektor `dict` enthalten.

`split( n, s, bool t )` spaltet im Baum `n` mehrfache Relationen auf; die Relationssymbole sind im Vektor `s` gespeichert (Algorithmus 6.2.3).

`make_sum( n, s, bool t )` erzeugt im Baum `n` die Summen-Darstellung für Concise, wobei die Summationssymbole im Vektor `s` gespeichert sind (Algorithmus 6.2.2).

`concise( n, expru, &index, &ex, &v )` erzeugt aus dem Baum `n` Concise-Ausdrücke; im Vektor `expru` sind die Regeln dazu gespeichert, `index` ist die Nummer der Ausdrücke, der Vektor `ex` enthält nachher die Ausdrücke und die Symboltabelle `val` die Werte von Variablen und Konstanten (Algorithmus 6.2.4).

`make_right_ass( n, s )` macht in dem Baum `n` alle Operatoren, die im Vektor `s` gespeichert sind, zu rechtsassoziativen Operatoren (Algorithmus 6.2.1).

In der Datei `exp_func.h` sind Funktionen zur Verarbeitung und Ausgabe von Concise-Ausdrücken definiert:

`output( e )` erzeugt die Ausgabe des Ausdrucks `e`.

`recordsheet( e, exp, val, type )` erzeugt aus den Ausdrücken im Vektor `exp` und den Werten im Vektor `val` ein Record Sheet; der erste Ausdruck ist *e* und sein Typ ist `type`.

`print_exp( value, expressions )` erzeugt die Ausgabe der Ausdrücke im Vektor `expressions` und der Werte im Vektor `value`.

In der Datei verb*readfile.h* sind Fuktionen zum Lesen von Dateien definiert:

`getformula( file )` liest alle Formeln, die in der Datei `file` in einer `$`- oder `$$`-Umgebung stehen, aus.

`loadFile( file, &form, &text )` lädt den Inhalt der Datei `file`; die Formeln werden im Vektor `form` gespeichert, die übrigen Textteile im Vektor `text`.

`loadTex( &st, file ))` lädt die TEX-Informationen aus der Datei `file`; die Tokens und ihre Typen werden in der Symboltabelle `st` gespeichert.

`loadSym( &sym, &prec, &right, file )` lädt Informationen zur Syntax aus der Datei `file`; die Typen werden in der Symboltabelle `sym` gespeichert, die Rangordnungen der Operatoren in der Symboltabelle `prec` und die rechtsasoziativen Operatoren im Vektor `right`.

`loadSem( &ru, &nu, &de, &se, &po, &lo, &ro, &expru, &rel, &sign, &dict, file )`
: lädt Informationen zur Syntax aus der Datei `file`; die semantische Kategorien werden im Vektor `ru` gespeichert, der Typ von Ziffern im String `nu`, das Dezimaltrennzeichen in `de`, Zeichenfolge, die ersetzt werden sollen, im Vektor `se`, Postfix-Operatoren im Vektor `po`, Typen von nicht erkannten linksassoziativen Operatoren im Vektor `lo`, Typen von nicht erkannten rechtssassoziativen Operatoren im Vektor `ro`, die regeln für die Erzeugung von Concise-Ausdrücken im Vektor `expru`, Relationssymbole im Vektor `rel`, Summationsoperatoren im Vektor `sign` und OpenMath-Regeln im Vektor `dict`.

### interpreter.cpp

```
#include "required.h"


#include "str2type.h"

#include "token.h"

#include "vec.h"

#include "next.h"

#include "tex.h"

#include "symbol.h"

#include "postfix.h"

#include "tree_func.h"

#include "exp_func.h"

#include "readfile.h"


int main( int argc, char **argv )
  {
  std::string texfile, synfile, semfile, infile;
  bool in, rep, con, exp, om;
  std::vector<std::string> f, T, v, r, sign;
  std::vector<exprule> exprules;
  std::vector<entry> cd;
  int n, index;
  node *t;

  in = rep = con = exp = om = false;

  for( int i=0 ; i<argc ; i++ )
    {
    if ( strcmp(argv[i],"-i") == 0 )
      {
      std::cout << argv[i] << std::endl;
      if( i<argc-1 )
        {
        infile = argv[++i];
        in=true;
        }
      else
```

```
      std::cerr << "No input file given!" << std::endl;
    }

  else if ( strcmp(argv[i],"-f") == 0 )
    {
    if( i<argc-1 )
      {
      ++i;
      if ( strcmp(argv[i],"concise") == 0 )
        con=true;
      if ( strcmp(argv[i],"expression") == 0 )
        exp=true;
      if ( strcmp(argv[i],"openmath") == 0 )
        om=true;
      }
    else
      std::cerr << "No output format given!" << std::endl;
    }

  else if ( strcmp(argv[i],"-t") == 0 )
    {
    if( i<argc-1 )
      texfile = argv[++i];
    else
      std::cerr << "No TeX file given!" << std::endl;
    }

  else if ( strcmp(argv[i],"-s") == 0 )
    {
    if( i<argc-1 )
      synfile = argv[++i];
    else
      std::cerr << "No syntax file given!" << std::endl;
    }

  else if ( strcmp(argv[i],"-d") == 0 )
    {
    if( i<argc-1 )
      semfile = argv[++i];
    else
      std::cerr << "No semantic file given!" << std::endl;
    }
  else if ( strcmp(argv[i],"-r") == 0 )
    {
    rep=true;
    }
  }
```

```
loadTex( texsymbols, texfile );
loadSym( symbols, precedence, right_ass, synfile );
loadSem( rules, number, dec, seq, post, lop, rop, exprules, r, sign, cd, semfile );

if(in)
  {
  if(rep)
    loadFile( infile, f, T );
  else
    f = getformula( infile );
  }
else
  {
  rep=false;
  f.resize(1);
  getline( std::cin, f.back() );
  }

n = f.size();

if( n<= 0 )
  {
  std::cerr << "No formulas!" << std::endl;
  return 0;
  }

for( int i=0 ; i<n ; i++ )
  {
  if(rep)
    std::cout << T[i] << std::endl;

  v = tokenize( f[i] );
  v = stylecheck(v);
  v = texcheck(v);
  v = postfix(v);

  t = new node();
  index = v.size();
  t = parse(v,index);

  if( seq.size()>0 )
    for( int j=0 ; j<seq.size() ; j++ )
      t = replace( t, seq[j] );

  t = rulecheck(t);

  if( seq.size()>0 )
    {
```

```
  for( int j=0 ; j<post.size() ; j++ )
    t = change_ass(t, post[j]);

  t = rulecheck(t);
  }

if( lop.size()>0 )
  for( int j=0 ; j<lop.size() ; j++ )
    t = make_l_op(t, lop.getName(j), lop.getType(j) );

if( rop.size()>0 )
  for( int j=0 ; j<rop.size() ; j++ )
    t = make_r_op(t, rop.getName(j), rop.getType(j) );

t = make_num(t, number, dec );

if( con || exp )
  {
  t = split( t, r, true );

  if( sign.size()>0 )
    {
    t = make_right_ass( t, sign );
    t = make_sum( t, sign, true );
    }

  symtable<int> value;
  std::vector<expression> expressions;

  concise( t, exprules, index, expressions, value );

  if( expressions.size()>0 || value.size()>0 )
    {
    if( exp )
      print_exp( value, expressions );
    if( con )
      recordsheet( expressions[0], expressions, value, "Example_aeqb" );
    }
  }

if( om )
  {
  std::cout << "<OMOBJ  xmlns=\"http://www.openmath.org/OpenMath\">" << std::endl;
  std::cout << "<OMA cdbase=\"http://www.openmath.org/cd\">" << std::endl;
  openmath( t, cd );
  std::cout << "</OMA>" << std::endl;
  std::cout << "</OMOBJ>" << std::endl;
  }
```

```
    }

    if( rep && T.size()>0 )
      std::cout << T.back() << std::endl;

  }
```

## required.h

```
#include<iostream>
#include<fstream>
#include<string>
#include<string.h>
#include<sstream>
#include<cstdlib>
#include<cmath>
#include<vector>

enum TYPE
{
  NONE, NAME, OP, LBRACK, RBRACK, NBRACK, VERT, CMD,
  LCUR, RCUR, SUB, SUP, ACC, SQRT, FRAC, TEXOP, FONT, TEXT, BLANK, STYLE, SIZE
};

#include "table.h"

#include "tree.h"

#include "rule.h"

#include "openmath.h"

#include "concise.h"

symtable<int> texsymbols, symbols, precedence, lop, rop;
std::vector< std::vector<std::string> > seq;
std::vector<std::string> right_ass, post;
std::string number, dec;
std::vector<rule> rules;
```

## str2type.h

```
int str2type( std::string s )
  {
  if( s=="NONE" )
    return NONE;
  if( s=="NAME" )
```

```
    return NAME;
  if( s=="OP" )
    return OP;
  if( s=="LBRACK" )
    return LBRACK;
  if( s=="RBRACK" )
    return RBRACK;
  if( s=="NBRACK" )
    return NBRACK;
  if( s=="VERT" )
    return VERT;
  if( s=="CMD" )
    return CMD;
  if( s=="LCUR" )
    return LCUR;
  if( s=="RCUR" )
    return RCUR;
  if( s=="SUB" )
    return SUB;
  if( s=="SUP" )
    return SUP;
  if( s=="ACC" )
    return ACC;
  if( s=="SQRT" )
    return SQRT;
  if( s=="FRAC" )
    return FRAC;
  if( s=="TEXOP" )
    return TEXOP;

  if( s=="FONT" )
    return FONT;
  if( s=="TEXT" )
    return TEXT;
  if( s=="BLANK" )
    return BLANK;
  if( s=="STYLE" )
    return STYLE;
  if( s=="SIZE" )
    return SIZE;

  return NONE;
  }
```

### token.h

```
inline bool isletter( char c )
  {
```

```
  if( (c>='a'&&c<='z') || (c>='A'&&c<='Z') )
    return true;
  return false;
  }

inline bool isblank( char c )
  {
  if( c==' ' || c=='\t' || c=='\n' )
    return true;
  return false;
  }


std::string token ( std::string &s, int &i )
  {
  std::string t;
  char c;

  while( isblank(s[i]) )
    ++i;

  c = s[i++];
  t.push_back(c);

  if( c=='\\' )
    {
    c = s[i++];
    t.push_back(c);

    if( isletter(c) )
      while( isletter(s[i]) )
        t.push_back(s[i++]);
    }

  return t;
  }


std::vector<std::string> tokenize ( std::string s )
  {
  int i=0;
  std::vector<std::string> v;

  while( i<s.length() )
    {
    v.push_back( token(s, i) );
    }
```

```
  return v;
  }
```

### vec.h

```
template <class T>
inline void append( std::vector<T> &a, std::vector<T> b )
  {
  for( int i=0 ; i<b.size() ; ++i )
    a.push_back(b[i]);
  }


std::string vec2string( std::vector<std::string> v )
  {
  std::string s;

  for( int i=0 ; i<v.size() ; ++i )
    s += v[i];

  return s;
  }


template <class T>
int search( std::vector<T> v, T x)
  {
  for( int i=0 ; i<v.size() ; i++ )
    if(v[i]==x)
      return i;

  return -1;
  }
```

### next.h

```
std::vector<std::string> next_within( std::vector<std::string> &v,
                                      std::string begin, std::string end, int &i)
  {
  std::vector<std::string> next;
  int n = v.size();

  ++i;

  if(i>=n)
    return next;
```

```
  if(v[i]!=begin)
    {
    next.push_back(v[i++]);
    return next;
    }

  ++i;

  for(int c=1 ; i<n ; ++i)
    {
    if(v[i]==end)
      {
      --c;
      if(c==0)
        {
        ++i;
        break;
        }
      }

    else if(v[i]==begin)
      ++c;

    next.push_back(v[i]);
    }

  return next;
  }


inline std::vector<std::string> next_group( std::vector<std::string> &v, int &i)
  {
  return next_within( v, "{", "}", i );
  }

inline std::vector<std::string>  next_squarebr( std::vector<std::string> &v, int &i)
  {
  return next_within( v, "[", "]", i );
  }
```

## tex.h

```
int tex_type( std::string s )
  {
  if( s=="" || s[0]==' ' || s[0]=='\0' )
    return BLANK;

  int i=texsymbols.search(s);
```

```
  if(i>=0)
    return texsymbols.getType(i);

  return NONE;
  }


void setfont( std::vector<std::string> &v, std::string s )
  {
  int n = v.size();
  std::string t;

  for( int i=0 ; i<n ; i++ )
    {
    t=v[i];

    if( isletter(t[0]) )
      {
      t=s;
      t+="{";
      t+=v[i];
      t+="}";
      v[i]=t;
      }
    }
  }


std::vector<std::string> stylecheck( std::vector<std::string> v )
  {
  std::vector<std::string> w, f;
  int n, type;
  std::string t;

  n = v.size();

  for( int i=0 ; i<n ; ++i )
    {
    t=v[i];
    type=tex_type(t);

    if( type==FONT )
      {
      f=next_group(v,i);
      --i;
      setfont(f,t);
      append(w,f);
```

```
      }
    else if( type==TEXT )
      {
      f=next_group(v,i);
      --i;
      w.push_back(t);
      w.back() += "{";
      w.back() += vec2string(f);
      w.back() += "}";
      }
    else if( type==SIZE )
      {
      t += v[++i];
      w.push_back(t);
      }
    else if( type!=BLANK && type!=STYLE )
      {
      w.push_back(t);
      }
    }

  return w;
  }


bool find_texop( std::vector<std::string> v )
  {
  int n, c;
  std::string t;

  n = v.size();
  c = 0;

  for( int i=0 ; i<n ; i++ )
    {
    t = v[i];
    if( t=="{" )
      ++c;
    else if( t=="}" )
      --c;
    else if ( c==0 && tex_type(t)==TEXOP )
      return true;
    }

  return false;
  }
```

```
std::vector<std::string> texcheck( std::vector<std::string> v )
  {
  std::vector<std::string> w, arg1, arg2;
  int n, type;
  std::string t;

  n = v.size();

  for( int i=0 ; i<n ; i++ )
    {
    t=v[i];
    type=tex_type(t);

    if( type==ACC || type==SQRT || type==FRAC )
      {
      arg1.resize(0);
      arg2.resize(0);

      if(type==ACC)
        {
        arg1 = next_group( v, i );
        }

      else if(type==SQRT)
        {
        if(v[i+1]=="[")
          {
          arg1 = next_squarebr( v, i );
          --i;
          }
        arg2 = next_group( v, i );
        }

      else if(type==FRAC)
        {
        arg1 = next_group( v, i );
        --i;
        arg2 = next_group( v, i );
        }

      --i;

      if( arg1.size()==0 )
        arg1.push_back("{}");
      else
        arg1=texcheck(arg1);

      if( arg2.size()==0 )
```

```
    arg2.push_back("{}");
  else
    arg2=texcheck(arg2);

  w.push_back("{");
  append(w,arg1);
  w.push_back(t);
  append(w,arg2);
  w.push_back("}");
  }

else if( type==LCUR )
  {
  --i;
  arg1 = next_group(v,i);
  arg1 = texcheck(arg1);
  i-=1;

  if( find_texop(arg1) )
    {
    w.push_back("{");
    append(w,arg1);
    w.push_back("}");
    }
  else
    {
    append(w,arg1);
    if( i<n-1 && arg1.size()==0 && (v[i+1]=="_"||v[i+1]=="^") )
      w.push_back("{}");
    }
  }

else if( type==SUB || type==SUP )
  {
  arg1.resize(0);
  arg2.resize(0);

  if(i==0)
    w.push_back("{}");

  if(t=="_")
    arg1 = next_group( v, i );

  if( i<n && v[i]=="_" )
    arg1 = next_group( v, i );

  if( i<n && v[i]=="^" )
    arg2 = next_group( v, i );
```

```
  if( i<n && v[i]=="\'" )
    {
    arg2.push_back("\\prime");
    ++i;
    }

  if( arg1.size()==0 )
    arg1.push_back("{}");
    else
      arg1=texcheck(arg1);

  if( arg2.size()==0 )
    arg2.push_back("{}");
    else
      arg2=texcheck(arg2);

  w.push_back("{");
  append(w,arg1);
  w.push_back("#subsup");
  append(w,arg2);
  w.push_back("}");
  --i;
  }

else if( t=="\\root" )
  {
  arg1.resize(0);
  arg2.resize(0);

  for( int j=i+1 ; j<n ; j++ )
    {
    ++i;
    if(v[j]!="\\of")
      arg1.push_back(v[j]);
    else
      break;
    }

  arg2 = next_group(v,i);
  --i;

  if( arg1.size()==0 )
    arg1.push_back("{}");
  else
    arg1=texcheck(arg1);

  if( arg2.size()==0 )
```

```
        arg2.push_back("{}");
      else
        arg2=texcheck(arg2);

      w.push_back("{");
      append(w,arg1);
      w.push_back("\\sqrt");
      append(w,arg2);
      w.push_back("}");
      }

    else if(t=="\'")
      {
      w.push_back("{");
      w.push_back("{}");
      w.push_back("#subsup");
      w.push_back("\\prime");
      w.push_back("}");
      }

    else
      {
      w.push_back(t);
      }
    }

  return w;
  }
```

## symbol.h

```
int type_of(  std::string s )
  {
  if( s=="" || s[0]==' ' || s[0]=='\0' )
    return NONE;

  int i=symbols.search(s);

  if(i>=0)
    return symbols.getType(i);

  return NAME;
  }


int precedence_of(  std::string s )
  {
  int i = precedence.search(s);
```

```
  if(i>=0)
    return precedence.getType(i);

  return 0;
  }


bool left_ass(  std::string s )
  {
  if( search( right_ass, s ) >= 0 )
    return false;

  return true;
  }
```

## postfix.h

```
std::vector<std::string> postfix( std::vector<std::string> v )
  {
  std::vector<std::string> stack, out, under;
  int n, type, t, prec, p;
  bool l, nbrack, vert;
  std::string s, b;

  nbrack=vert=false;
  n = v.size();

  for( int i=0 ; i<n ; i++ )
    {
    s = v[i];

    if(s=="{")
      stack.push_back(s);

    else if(s=="}")
      {

      while( stack.size() > 0 && stack.back()!="{" )
        {
        out.push_back( stack.back() );
        stack.pop_back();
        }

      stack.pop_back();

      if( out.size()>0 && out.back()=="#subsup" )
        {
        out.push_back( "#link" );
```

```
    if( stack.size()>0 && stack.back()=="#link" )
      stack.pop_back();
    }
  }

else
  {
  type=type_of(s);

  if(type==NAME)
    {
    out.push_back(s);
    }


  else if(type==OP)
    {
    prec = precedence_of(s);
    l=left_ass(s);

    while( stack.size()>0 )
      {
      b=stack.back();

      if( type_of(b)!=OP )
        break;

      p=precedence_of(b);

      if( l && prec>p )
        break;

      if( !l && prec>=p )
        break;

      out.push_back(b);
      stack.pop_back();
      }

    stack.push_back(s);
    }


  else if(type==LBRACK)
    {
    stack.push_back(s);
    }
```

```
else if(type==RBRACK)
  {
  while( stack.size()>0 && type_of(stack.back())!=LBRACK )
    {
    out.push_back( stack.back() );
    stack.pop_back();
    }

  if(stack.size()>0)
    {
    out.push_back( stack.back() );
    out.push_back(s);
    stack.pop_back();
    }
  }


else if(type==NBRACK)
  {
  nbrack = !nbrack;

  if(nbrack)
    stack.push_back(s);

  else
    {
    while( stack.size() > 0 )
      {
      b=stack.back();
      t=type_of(b);
      out.push_back( b );
      stack.pop_back();

      if(t==NBRACK)
        break;
      }
    out.push_back(s);
    }
  }


else if(type==VERT)
  {
  vert = !vert;

  if(vert)
```

```
    stack.push_back(s);
  else
    {
    while( stack.size() > 0 )
      {
      b=stack.back();
      out.push_back( b );
      stack.pop_back();

      if(b==s)
        break;
      }

    out.push_back(s);
    }
  }


else if(type==CMD)
  {
  while( stack.size() > 0 )
    {
    b=stack.back();

    if(b=="{")
      break;

    if( b.size()>4 && b[0]=='\\' && b[1]=='l' && b[2]=='e' && b[3]=='f' && b[4]=='t' )
      break;

    out.push_back( stack.back() );
    stack.pop_back();
    }
  stack.push_back(s);
  }
}

if( type==OP )
  {
  if( i==0 )
    {
    out.push_back("{}");
    }
  else
    {
    b = v[i-1];
    t = type_of(b);
    if( t==CMD || t==LBRACK || (nbrack&&t==NBRACK) || (vert&&t==VERT) || b=="{" )
```

```
          out.push_back("{}");
        }

      if( i>=n-1 )
        {
        if( i>n-1 || v[i] !="}" )
          out.push_back("{}");
        }
      else
        {
        b = v[i+1];
        t = type_of(b);
        if( t==CMD || t==OP || t==RBRACK || (nbrack&&t==NBRACK) || (vert&&t==VERT) || b=="}" )
          out.push_back("{}");
        }
      }

    else if(type==LBRACK)
      {
      if( i<n && type_of(v[i+1])==RBRACK )
        out.push_back("{}");
      }

    else if(type==NBRACK)
      {
      if( i<n && type_of(v[i+1])==NBRACK )
        out.push_back("{}");
      }

    else if(type==VERT)
      {
      if( i<n && type_of(v[i+1])==VERT )
        out.push_back("{}");
      }

  t=NONE;
  if(i<n-1)
    {
    b=v[i+1];
    t=type_of(b);

    if( s!="{" &&
        (type==NAME || type==RBRACK || (type==NBRACK && !nbrack) || (type==VERT && !vert) || s=="}") )
      {
      if( i<n-1 && v[i+1]!="" )
        {
        if( t!=OP && t!=RBRACK && (t!=NBRACK || !nbrack ) && (t!=VERT || !vert) && t!=CMD && b!="}" )
          {
```

```
            stack.push_back("#link");
            }
          }
        }
      }
    }

  while( stack.size()>0 )
    {
    out.push_back( stack.back() );
    stack.pop_back();
    }

  return out;
  }
```

## tree_func.h

```
node *parse( std::vector<std::string> s, int &i )
  {
  if(i<1)
    return 0;

  std::string t = s[--i];

  if(t=="{}")
    return 0;

  node *x = new node;
  int type = type_of(t);
  x->addName(t);

  if( type==OP || type==CMD )
    {
    x->setRight( parse(s,i) );
    x->setLeft( parse(s,i) );
    }

  else if( type==RBRACK || type==NBRACK || type==VERT )
    {
    if(i>0)
      x->addName(s[--i]);
    x->setLeft( parse(s,i) );
    }

  return x;
  }
```

```
node *change_ass( node *n, std::string s )
  {
  if(!n)
    return 0;

  node *l, *r, *t1, *t2, *t3;

  t1=t2=t3=0;

  l = n->getLeft();
  r = n->getRight();

  l = change_ass(l,s);
  r = change_ass(r,s);

  if( n->getName(0) != "#link" )
    return n;

  t1 = l;

  if(r)
    {
    t2 = r->getLeft();
    t3 = r->getRight();
    }

  if( t2 && t3 && t2->getType() == s )
    {
    r = t3;
    l = new node;
    l-> addName("#link");
    l->setLeft(t1);
    l->setRight(t2);

    n->setLeft( l );
    n->setRight( r );
    }

  return n;
  }


node *parent( node *t, node *n )
  {
  if( !t || !n )
    return 0;
```

```
  node *l = new node;
  node *r = new node;
  node *p = new node;

  l = t->getLeft();
  r = t->getRight();

  if( l==n || r==n )
    return t;

    p = parent( l, n );

    if(p)
      return p;

    return parent( r, n );
  }


void tree2vec( node *t, std::vector<std::string> &v, std::vector<node *> &w )
  {
  if(!t)
    return;

  node *l, *r;
  std::string n;
  int type;

  l = t->getLeft();
  r = t->getRight();
  n = t->getName(0);
  type = type_of(n);

  if( type == RBRACK || type== NBRACK || type == VERT )
    {
    v.push_back( t->getName(1) );
    w.push_back(t);

    if(l)
      tree2vec(l, v, w);

    v.push_back(n);
    w.push_back(t);

    return;
    }

  type = tex_type(n);
```

```
if( type == ACC || type== SQRT || type == FRAC )
  {
  v.push_back(n);
  w.push_back(t);

  if(l)
    {
    if(type==SQRT)
      {
      v.push_back("[");
      tree2vec(l, v, w);
      v.push_back("]");
      }
    else
      {
      v.push_back("{");
      tree2vec(l, v, w);
      v.push_back("}");
      }
    }

  if(r)
    {
    v.push_back("{");
    tree2vec(r, v, w);
    v.push_back("}");
    }
  return;
  }

if(n=="#subsup")
  {
  if(l)
    {
    v.push_back("_{");
    w.push_back(t);
    tree2vec(l, v, w);
    v.push_back("}");
    }

  if(r)
    {
    v.push_back("^{");
    w.push_back(t);
    tree2vec(r, v, w);
    v.push_back("}");
    }
```

```
    return;
    }


  if(l)
    tree2vec(l, v, w);

  if(n!="#link")
    {
    v.push_back(n);
    w.push_back(t);
    }

  if(r)
    tree2vec(r, v, w);
  }


std::string tree2string( node *n )
  {
  std::string s = "";

  if(!n)
    return s;

  node *l, *r;
  std::string name;
  int type;

  l = n->getLeft();
  r = n->getRight();
  name = n->getName(0);
  type = tex_type(name);

  if( type == ACC || type== SQRT || type == FRAC )
    {
    s += name;

    if(l)
      {
      if(type==SQRT)
        {
        s += "[";
        s += tree2string(l);
        s += "]";
        }
      else
        {
```

```
      s += "{";
      s += tree2string(l);
      s += "}";
      }
    }

  if(r)
    {
    s += "{";
    s += tree2string(r);
    s += "}";
    }

  return s;
  }

if( name=="#subsup" )
  {
  if(l)
    {
    s += "_{";
    s += tree2string(l);
    s += "}";
    }

  if(r)
    {
    s += "^{";
    s += tree2string(r);
    s += "}";
    }
  }

type = type_of(name);

if( type==RBRACK || type==NBRACK || type==VERT )
  {
  s += n->getName(1);

  if(l)
    s += tree2string(l);

  s += name;

  return s;
  }

if(l)
```

```
    s += tree2string(l);

  if( name=="#link" )
    name="";

  s +=name;

  if(r)
    s += tree2string(r);

  return s;
  }


void search( std::vector<std::string> v, std::vector<std::string> p, std::vector<int> &f )
  {
  int i, j, m, n;

  n = v.size();
  m = p.size();

  for( i=0 ; i<n ; i++ )
    {
    if( v[i]==p[0] && i<=n-m )
      {
      for( j=1 ; j<m ; j++ )
        {
        if( v[i+j] != p[j] )
          break;
        }

      if(j==m)
        {
        f.push_back(i);
        i = i+m-1;
        }
      }
    }
  }


node *replace( node *t, std::vector<std::string> seq )
  {
  if(!t)
    return 0;

  if( seq.size()<2 )
    return t;
```

```
std::vector<std::string> vec;
std::vector<node *> nodes;
std::vector<int> found;
int match, type;
node *L, *R, *T, *LR, *first, *last, *next;

type = type_of(seq[0]);

tree2vec( t, vec, nodes );
search( vec, seq, found );

if( found.size()==0 )
  return t;

for( int i=0 ; i<found.size() ; i++ )
  {
  match = found[i];

  L = new node;
  R = new node;
  T = new node;

  first = new node;
  last  = new node;
  next  = new node;

  first = nodes[match];
  last  = nodes[match+seq.size()-1];
  next  = nodes[match+seq.size()];

  if( type==OP )
    {
    L = first->getLeft();
    T = last;
    R = T->getRight();

    if( T->getLeft() && T->getLeft()->getName(0)!=seq[seq.size()-2] )
      {
      L = T->getLeft();
      LR = new node;
      LR = L->getRight();

      for( int i=0 ; i<seq.size()-1 ; i++ )
        {
        LR = new node;
        LR = LR->getLeft();
        }
```

```
        L ->setRight(LR);
        }

      t = new node;
      t->setLeft(L);
      t->setRight(R);
      t->setName(vec2string(seq));
      }

    else if( type==NAME )
      {
        L = new node;
        R = new node;
        T = new node;
        T = parent( t, first );

      if( parent(t,last) == parent(t,nodes[found[i]+seq.size()-2]) )
        {
        T->setName( vec2string(seq) );
        L=R=0;
        }

      else
        {
        R = parent(t, next );

        if( R == parent(t, last ) )
          R = next;

        L->addName(vec2string(seq));
        T->addName("#link");
        }

      T->setLeft(L);
      T->setRight(R);
      }
    }

return t;
}


node *roptree( node *n, std::string t )
  {
  node *l, *r, *arg, *op, *p1, *p2, *p3;

  l   = new node;
```

```
  r   = new node;
  arg = new node;
  op  = new node;
  p1  = new node;
  p2  = new node;
  p3  = new node;

  r = n;
  p1 = p2 = p3 = 0;

  while(r)
    {
    p3 = p2;
    p2 = p1;
    p1 = r;
    l = r->getLeft();
    r = r->getRight();

    if( l && l->getType()==t )
      break;
    }

  if(!r)
    return n;

  arg->setName("#arg");

  if(p3)
    {
    p3->setRight( p2->getLeft() );
    arg->setLeft( n );
    }
  else
    arg->setLeft( p2->getLeft() );

  arg->setRight( roptree( r, t ) );

  op->setName("#op");
  op->setLeft(l);
  op->setRight(arg);

  return op;
  }


node *loptree( node *n, std::string t )
  {
  node *l, *r, *arg, *op, *p1, *p2, *p3, *P1, *P2, *P3;
```

```
l   = new node;
r   = new node;
arg = new node;
op  = new node;
p1  = new node;
p2  = new node;
p3  = new node;
P1  = new node;
P2  = new node;
P3  = new node;

r = n;
p1 = p2 = p3 = P1 = P2 = P3 = 0;

while(r)
  {
  p3 = p2;
  p2 = p1;
  p1 = r;
  r = r->getRight();

  if(!r)
    break;

  l = p1->getLeft();

  if( l && l->getType()==t )
    {
    P3=p3;
    P2=p2;
    P1=p1;
    }
  }

if(!P1)
  return n;

arg->setName("#arg");

if(P3)
  {
  P3->setRight( P2->getLeft() );
  arg->setLeft( loptree( n, t ) );
  }
else
  arg->setLeft( P2->getLeft() );
```

```
  arg->setRight( P1->getRight() );

  op->setName("#op");
  op->setLeft( P1->getLeft() );
  op->setRight(arg);

  return op;
  }


node *make_r_op( node *n, std::string t, int p )
  {
  if(!n)
    return 0;

  std::string name = n->getName(0);

  if(name=="#link")
    return roptree(n,t);

  node *r = n->getRight();
  node *l = n->getLeft();

  l = make_r_op(l, t, p);
  r = make_r_op(r, t, p);

  n->setLeft(l);
  n->setRight(r);

  if( type_of(name)!=OP || p >= precedence_of(name) )
    return n;

  node *op = new node;
  node *arg = new node;
  node *la = new node;
  node *ra = new node;
  node *na = new node;
  node *pa = new node;

  if( r && r->getName(0)=="#op" )
    {
    op = r->getLeft();

    if( op->getType()==t )
      {
      arg = r->getRight();
      la = arg->getLeft();
      na->setName(n->getName());
```

```
      na->setLeft(l);
      na->setRight(la);
      arg->setLeft(na);
      n = new node;
      n->setName("#op");
      n->setLeft(op);
      n->setRight(arg);
      }
    }

  if( l && l->getName(0)=="#op" )
    {
    op = l->getLeft();

    if( op->getType()==t )
      {
      arg = new node;
      ra = new node;
      na = new node;
      pa = new node;

      arg = l->getRight();
      ra = arg->getRight();
      pa = arg;

      while( ra->getName(0)=="#op" )
        {
        pa = ra->getRight();
        ra = pa->getRight();
        }

      na->setLeft(ra);
      na->setRight(r);
      na->setName( n->getName() );
      pa->setRight(na);
      n = new node;
      n = l;
      }
    }

  return n;
  }


node *make_l_op( node *n, std::string t, int p )
  {
  if(!n)
    return 0;
```

```
std::string name = n->getName(0);

if(name=="#link")
  return loptree(n,t);

node *l = n->getLeft();
node *r = n->getRight();

l = make_l_op( l, t, p );
r = make_l_op( r, t, p );

n->setLeft(l);
n->setRight(r);

if( type_of(name)!=OP || p >= precedence_of(name) )
  return n;

node *op = new node;
node *arg = new node;
node *la = new node;
node *ra = new node;
node *na = new node;

if( l && l->getName(0)=="#op" && l->getLeft()->getType()==t )
  {
  arg = l->getRight();
  la = arg->getLeft();
  ra = arg->getRight();

  na->setName(n->getName());
  na->setLeft(ra);
  na->setRight(r);
  arg->setRight(na);

  n = new node;
  n = l;
  }

if( r && r->getName(0)=="#op" && r->getLeft()->getType()==t )
  {
  arg = r->getRight();

  while( arg )
    {
    la = arg->getLeft();

    if( la->getName(0)=="#op" )
```

```
        arg = la->getRight();
      else
        break;
      }

    la = arg->getLeft();

    na->setName( n->getName() );
    na->setLeft(l);
    na->setRight( arg->getLeft() );

    arg->setLeft(na);
    n = new node;
    n = r;
    }

  return n;
  }


bool is_num( node *n )
  {
  if(!n)
    return false;

  std::string s;
  s = n->getName()[0];

  if( isdigit(s[0]) )
    return true;

  if( s=="#link" )
    if( is_num(n->getLeft()) && is_num(n->getRight()) )
      return true;

  return false;
  }


bool is_num( node *n, std::string t )
  {
  if(!n)
    return false;

  if( n->getType()==t && !n->getLeft() && !n->getRight() )
    return true;

  if( n->getName(0)=="#link" && is_num( n->getLeft(), t ) && is_num( n->getRight(), t ) )
```

```
    return true;

  return false;
  }


node *change_prec( node *n, std::string t )
  {
  if(!n)
    return 0;
  if(n->getName(0)!="#link")
    return n;

  node *l   = new node;
  node *r   = new node;
  node *num = new node;
  node *p1  = new node;
  node *p2  = new node;
  node *p3  = new node;

  r = n;
  p1 = p2 = p3 = 0;

  while(r)
    {
    p3 = p2;
    p2 = p1;
    p1 = r;
    l = r->getLeft();

    if( !l || l->getType()!=t )
      break;

    r = r->getRight();
    }

  if( !r || r->getType()==t )
    return n;

  if( p3 )
    p3->setRight( p2->getLeft() );

  else if( p2 )
    n = p2->getLeft();

  else
    {
    n = p1->getLeft();
```

```
    r = p1->getRight();
    }

  num->setName("#link");
  num->setLeft(n);
  num->setRight( change_prec(r,t) );

  return num;
  }


node *make_num( node *n, std::string t, std::string d )
  {
  if(!n)
    return 0;

  std::string name = n->getName(0);

  if(name=="#link")
    return change_prec(n,t);

  node *l = new node;
  node *r = new node;
  node *p1 = new node;
  node *p2 = new node;
  node *p3 = new node;
  node *P1 = new node;
  node *P2 = new node;
  node *P3 = new node;
  node *llink = new node;
  node *rlink = new node;
  node *num = new node;
  node *link = new node;

  l = n->getLeft();
  r = n->getRight();

  llink = rlink = 0;

  if(name!=d)
    {
    l = make_num(l, t, d);
    r = make_num(r, t, d);
    n->setLeft(l);
    n->setRight(r);

    return n;
    }
```

```
if( l && l->getName(0)=="#link" )
  {
  p2=p3=P1=P2=P3=0;

  p1=l;

  while(p1)
    {
    if( !p1->getRight() )
      break;

    if( p1->getLeft() && p1->getLeft()->getType()!=t )
      {
      P1=p1;
      P2=p2;
      P3=p3;
      }

    p3=p2;
    p2=p1;
    p1=p1->getRight();
    }

  if(P1)
    {
    if(P2)
      {
      P2->setRight( P1->getLeft() );
      llink = l;
      }
    else
      llink = l->getLeft();

    l = P1->getRight();
    }
  }

if( r && r->getName(0)=="#link" )
  {
  p1=new node;
  p2=new node;
  p3=new node;
  p3=p2=0;
  p1=r;

  while(p1)
```

```
    {
    if( ( p1->getLeft() && p1->getLeft()->getType()!=t ) || !p1->getRight() )
      break;

    p3=p2;
    p2=p1;
    p1=p1->getRight();
    }

  if( !p1 || p1->getType()!=t )
    {
    rlink=p1;
    if(p3)
      p3->setRight( p2->getLeft() );
    else if(p2)
      r=p2->getLeft();
    }
  }

n = new node;

link->setName("#link");
num->setName(d);
num->setLeft(l);
num->setRight(r);

if(rlink)
  {
  link->setLeft(num);
  link->setRight(rlink);

  if(llink)
    {
    n->setName("#link");
    n->setLeft(llink);
    n->setRight(link);
    }
  else
    {
    n=link;
    }
  }

else if(llink)
  {
  n->setName("#link");
  n->setLeft(llink);
  n->setRight(num);
```

```
    }

  else
    n=num;

  return n;
  }


node *rulecheck( node *n )
  {
  if(!n)
    return 0;

  node *l, *r;
  std::vector<std::string> name;
  std::string type, ltype, rtype;
  rule ru;

  type = n->getType();
  name = n->getName();

  l = n->getLeft();
  r = n->getRight();

  l = rulecheck(l);
  r = rulecheck(r);

  n->setLeft(l);
  n->setRight(r);
  n->setType("NONE");

  if(l)
    ltype = l->getType();
  else
    ltype = "NONE";

  if(r)
    rtype = r->getType();
  else
    rtype = "NONE";

  ru = rule( name, type, ltype, rtype );

  for( int i = 0 ; i<rules.size() ; i++ )
    {
    if( ru.equals( rules[i] ) )
      {
```

```
      n->setType( rules[i].getHead() );
      break;
      }
    }

  return n;
  }


void openmath( node *n, std::vector<entry> dict )
  {
  if(!n)
    return;

  node *l, *r;
  std::vector<std::string> symb;
  std::string ltype, rtype;
  std::string name, cd, obj;
  bool found = false;

  l = n->getLeft();
  r = n->getRight();

  symb = n->getName();

  ltype=rtype="NONE";

  if(l)
    ltype = l->getType();

  if(r)
    rtype = r->getType();

  if( symb[0]=="#link" && ltype=="BIGOP" )
    {
    std::cout << "<OMA>" << std::endl;
    std::cout << "<OMS cd=\"arith1\" name=\"";

    if( l->getLeft()->getName(0)=="\\sum" )
      std::cout << "sum" ;

    else if( l->getLeft()->getName(0)=="\\prod" )
      std::cout << "product" ;

    std::cout << "\"/>" << std::endl;

    std::cout << "<OMA>" << std::endl;
    std::cout << "<OMS cd=\"interval1\" name=\"integer_interval\"/>" << std::endl;
```

```
  openmath(l->getRight()->getLeft()->getRight(),dict);
  openmath(l->getRight()->getRight(),dict);
  std::cout << "</OMA>" << std::endl;

  std::cout << "<OMBIND>" << std::endl;
  std::cout << "<OMS cd=\"fns1\" name=\"lambda\"/>" << std::endl;
  std::cout << "<OMBVAR>" << std::endl;
  std::cout << "<OMV name=\"" << l->getRight()->getLeft()->getLeft()->getName(0)
            << "\"/>" << std::endl;
  std::cout << "</OMBVAR>" << std::endl;

  openmath(r,dict);

  std::cout << "</OMBIND>" << std::endl;

  std::cout << "</OMA>" << std::endl;

  return;
  }

if( is_num(n) )
  {
  std::cout << "<OMI>" << tree2string(n) << "</OMI>" << std::endl;
  return;
  }

else if ( symb[0]=="." && is_num(l) && is_num(r) )
  {
  std::cout << "<OMF>" << tree2string(l) << "." << tree2string(r) << "</OMF>" << std::endl;
  return;
  }

else
  {
  for( int i=0 ; i<dict.size() ; i++ )
    {
    if( dict[i].equals( symb, ltype, rtype ) )
      {
      name = dict[i].getName();
      cd = dict[i].getCd();
      obj = dict[i].getObject();
      found = true;

      if( obj=="OMS" )
        {
        if(l||r)
        std::cout << "<OMA>" << std::endl;
        std::cout << "<OMS cd=\"" << cd << "\" name=\"" << name << "\" />" << std::endl;
```

```
  if(dict[i].prefix())
    {
    if(l)
      openmath(l,dict);

    if(r)
      openmath(r,dict);
    }
  else
    {
    if(r)
      openmath(r,dict);
    if(l)
      openmath(l,dict);
    }

  if(l||r)
  std::cout << "</OMA>" << std::endl;
  }

if( obj=="OMBIND" )
  {
  if(l||r)
  std::cout << "<OMBIND>" << std::endl;
  std::cout << "<OMS cd=\"" << cd << "\" name=\"" << name << "\" />" << std::endl;

  if(dict[i].prefix())
    {
    if(l)
      {
      std::cout << "<OMBVAR>" << std::endl;
      openmath(l,dict);
      std::cout << "</OMBVAR>" << std::endl;
      }

    if(r)
      openmath(r,dict);
    }
  else
    {
    if(r)
      std::cout << "<OMBVAR>" << std::endl;
      openmath(r,dict);
      std::cout << "</OMBVAR>" << std::endl;

    if(l)
      openmath(l,dict);
```

```
      }

    if(l||r)
    std::cout << "</OMBIND>" << std::endl;
    }

  else if( obj=="OMA" )
    {
    std::cout << "<OMA>" << std::endl;

    if(dict[i].prefix())
      {
      if(l)
        openmath(l,dict);

      if(r)
        openmath(r,dict);
      }
    else
      {
      if(r)
        openmath(r,dict);

      if(l)
        openmath(l,dict);
      }

    std::cout << "</OMA>" << std::endl;
    }

  else if( obj=="OM#" )
    {
    if(dict[i].prefix())
      {
      if(l)
        openmath(l,dict);

      if(r)
        openmath(r,dict);
      }
    else
      {
      if(r)
        openmath(r,dict);

      if(l)
        openmath(l,dict);
      }
```

```
          }

        else if( obj=="OMV" )
          {
            std::cout << "<OMV name=\"" << name << "\"/>" << std::endl;
          }
        break;
        }
      }

    if( !found )
      {
      std::cerr << "OPENMATH ERROR!\t" << n << std::endl;
      return;
      }
    }
  }


node *split( node *n, std::vector<std::string> s, bool t )
  {
  if(!n)
    return 0;

  node *l, *r, *L, *LL;
  bool f, rf;
  f=rf=false;

  l = n->getLeft();
  r = n->getRight();

  if( search( s, n->getName(0) )>=0 )
    f=true;

  if( r && search( s, r->getName(0) )>=0 )
    rf=true;

  if( ( t && !rf) || !f )
    {
    l = split(l,s,t);
    r = split(r,s,t);
    n->setLeft(l);
    n->setRight(r);
    return n;
    }

  if( !t && !rf )
    {
```

```
    LL = new node;
    LL->setType("Dummy");
    LL->setName( "#link" );
    LL->setRight(l);
    n->setLeft(LL);

    return n;
    }

  L = new node;

  L->setName( n->getName() );
  L->setType( n->getType() );
  L->setLeft(l);
  L->setRight(r->getLeft());

  r = split( r, s, !f );

  n->setName("#link");
  n->setType("ExpLink");
  n->setLeft( L );
  n->setRight( r );

  if(!t)
    {
    LL = new node;
    LL->setType("Dummy");
    LL->setName( "#link" );
    LL->setRight(l);
    L->setLeft(LL);
    }
  else
    {
    node *R = new node;
    R->setName("#link");
    R->setType("ExpLink");
    R->setRight(r);

    n->setType("Chain");
    n->setRight(R);
    n->setRight(r);
    }

  return n;
  }


node *make_sum( node *n, std::vector<std::string> s, bool t )
```

```
  {
  if(!n)
    return 0;

  bool found=false;

  if( search( s, n->getName(0) )>=0 )
    found=true;

  node *l = n->getLeft();
  node *r = n->getRight();

  l = make_sum(l,s,!found);
  r = make_sum(r,s,!found);

  n->setLeft(l);
  n->setRight(r);

  if( found && t )
    {
    node *el = new node;
    node *ssl = new node;

    ssl->setName("#link");
    ssl->setType("SignedSum");
    ssl->setRight(n);

    el->setName("#link");
    el->setType("ExpLink");
    el->setRight(ssl);

    n = new node;
    n = el;
    }

  return n;
  }


void concise( node *n, std::vector<exprule> expru, int &index,
              std::vector<expression> &ex, symtable<int> &v )
  {
  if(!n)
    return;

  node *l, *r;
  int num, exidx, i;
  std::string lt, rt, val;
```

```
std::vector<std::string> s;
exprule e;

l = n->getLeft();
r = n->getRight();
++index;
num=index;

lt = rt = "NONE";

if(l)
  lt = l->getType();
if(r)
  rt = r->getType();

s = n->getName();

for (i=0 ; i<expru.size() ; i++ )
  {
  if( expru[i].equals(s,lt,rt) )
    {
    e=expru[i];
    break;
    }
  }

if(i>=expru.size() )
  {
  std::cerr << "No exprule found!" << std::endl;
  return;
  }

if( e.getHexp()=="VALUE" )
  {
  if( is_num(n) || ( s[0]=="." && is_num(l) && is_num(r) ) )
    val = tree2string(n);
  else
    val = e.getHtype();

  if( v.search( val ) < 0 )
    {
    v.insert( val, num );
    ex.push_back( expression( num, e.getType() ) );
    }
  else
    {
    for( int i = 0 ; i < ex.size() ; i++ )
      {
```

```
        for( int j = 0 ; j < ex[i].sub_num() ; j++ )
          {
          if( ex[i].getSub(j) == num )
            ex[i].setSub(j, v.getType( v.search( val ) ) );
          }
        }
      }
    return;
    }

  if( e.getHtype() != "#" )
    {
    ex.push_back( expression( num, e.getType() ) );
    exidx = ex.size()-1;

    if( e.getHexp()!="." )
      {
      ex[exidx].add_type( e.getHtype(), e.getHexp() );
      }
    }
  else
    {
    for( exidx=0 ; exidx<ex.size() ; exidx++ )
      if( ex[exidx].getNum()==index-2 )
        break;
    }

  if( l )
    {
    if( e.getLexp()!="#" )
      ex[exidx].add_exp( index+1, e.getLexp() );
    else if( e.getRexp()=="#" )
      --index;
    concise( l, expru, index, ex, v );
    }

  if( r )
    {
    if( e.getRexp()!="#" )
      ex[exidx].add_exp( index+1, e.getRexp() );
    else if( e.getLexp()=="#" )
      --index;
    concise( r, expru, index, ex, v );
    }
  }


node *make_right_ass( node *n, std::vector<std::string> s)
```

```
  {
  if(!n)
    return 0;

  node *l, *r, *L, *R;

  l = n->getLeft();
  r = n->getRight();

  if( search( s, n->getName(0) ) < 0 || !l || search( s, l->getName(0) ) < 0 )
    {
    l = make_right_ass( l, s );
    r = make_right_ass( r, s );

    n->setLeft(l);
    n->setRight(r);

    return n;
    }

  L = new node;
  R = new node;

  L = l->getLeft();
  R->setName( n->getName() );
  R->setLeft( l->getRight() );
  R->setRight( r );

  n = new node;
  n->setName( l->getName() );
  n->setLeft(L);
  n->setRight(R);

  return make_right_ass(n,s);
  }
```

## exp_func.h

```
void output( expression e )
  {
  int num = e.getNum();

  for( int i=0 ; i<e.type_num() ; i++ )
    std::cout << "$" << num << "." << e.getType(i) << "=" << e.getName(i) << std::endl;

  for( int i=0 ; i<e.sub_num() ; i++ )
    std::cout << "$" << num << "." << e.getAttr(i) << "=" << e.getSub(i) << std::endl;
```

```
  }


void recordsheet( expression e, std::vector<expression> exp, symtable<int> val, std::string type )
  {
  if( exp.size()==1 )
    {
    std::cout << type << "(" << exp[0].getName(0) << ")>" << std::endl;
    std::cout << "name(String)> [EXT]" << val.getName(0) << "[/EXT]" << std::endl;
    std::cout << "< <" << std::endl ;

    return;
    }

  int index;

  std::cout << type << "(" << e.getName(0) << ")>" << std::endl;

  for( int i=1 ; i<e.type_num() ; i++ )
    {
    std::cout << e.getType(i) << "(" << e.getName(i) << ")>" << std::endl;
    }

    std::cout << "<" << std::endl;

  for( int i=0 ; i<e.sub_num() ; i++ )
    {
    for( int j=0 ; j<exp.size() ; j++ )
      {
      if( exp[j].getNum()==e.getSub(i) )
        {
        index = -1;

        for( int k=0 ; k<val.size() ; k++ )
          {
          if( exp[j].getNum() == val.getType(k) )
            {
            index=k;
            break;
            }
          }

        if( index >= 0 )
          {
          std::cout << e.getAttr(i) << "(" << exp[j].getName(0) << ")>" << std::endl;
          std::cout << "name(String)> [EXT]" << val.getName(index) << "[/EXT]" << std::endl;
          std::cout << "< <" << std::endl ;
          }
```

```
        else
          {
          recordsheet( exp[j], exp, val, e.getAttr(i) );
          }
        }
      }
    }
    std::cout << "<" << std::endl;
  }


void print_exp( symtable<int> value, std::vector<expression> expressions )
  {
  for( int i=0 ; i<expressions.size() ; i++ )
    {
    output( expressions[i] );
    }

  for( int i=0 ; i<value.size() ; i++ )
    {
    std::cout << "VALUE($" << value.getType(i) << ")=str:" << value.getName(i) << std::endl;
    }
  }
```

## readfile.h

```
std::vector<std::string>getformula( std::string file)
  {
  std::ifstream infile ( file.c_str() );
  std::vector<std::string> form;
  std::string line;
  char x,y;
  bool math;

  if(!infile.is_open())
    {
    std::cerr << "Error: Unable to open file " << file << std::endl;
    return form;
    }

  math=false;
  x=y=0;

  while ( !infile.eof() )
    {
    y=x;
    infile.get(x);
```

```
    if(x=='%')
      {
      y='\n';
      while( x!='\n' && !infile.eof() )
        infile.get(x);
      }

    else if(x=='$')
      {
      if(y=='$')
        math=!math;

      if(math)
       {
        form.push_back(line);
        line="";
       }
      math=!math;
      }

    else if(math)
      {
        line.push_back(x);
      }
    }

  infile.close();

  return form;
  }


void loadFile( std::string file, std::vector<std::string> &form, std::vector<std::string> &text )
  {
  char x,y;
  bool math;
  std::string line, empty;

  std::ifstream infile ( file.c_str() );

  if(!infile.is_open())
    {
    std::cerr << "Error: Unable to open file " << file << std::endl;
    return;
    }

  math=false;
  x=y=0;
```

```
  while (! infile.eof() )
    {
    y=x;
    infile.get(x);

    if(x=='$')
      {
      if(y=='$')
        {
        math=!math;
        }

      if(math)
        {
        form.push_back(line);
        }
      else
        {
        text.push_back(line);
        }

      math=!math;
      line="";
      }
    else if(x!='$')
      {
      line.push_back(x);
      }
    }

  text.push_back(line);
  infile.close();
  }


std::string next_word ( std::ifstream &infile )
  {
  std::string s;
  char x=' ';

  while ( !infile.eof() )
    {
    if ( isblank(x) )
      {
      while ( !infile.eof() && isblank(x) )
        infile.get(x);
      }
```

```
    else if ( x=='%' )
      {
      while( x!='\n' )
        infile.get(x);
      }
    else
      {
      break;
      }
    }

  while ( !infile.eof() && !isblank(x) )
    {
    s.push_back(x);
    infile.get(x);
    }

  if( s[s.size()-1]=='\\' && x==' ' )
    s.push_back(x);

  return s;
  }


void loadTex( symtable<int> &st, std::string file )
  {
  std::ifstream infile ( file.c_str() );

  if ( !infile.is_open() )
    {
    std::cerr << "Unable to open file " << file << std::endl;
    return;
    }

  std::string s, t;

  while( !infile.eof() )
    {
    s = next_word( infile );
    t = next_word( infile );
    st.insert( s, str2type(t) );
    }
  }


void loadSym( symtable<int> &sym, symtable<int> &prec, std::vector<std::string> &right,
              std::string file )
  {
```

```
  std::ifstream infile ( file.c_str() );

  if ( !infile.is_open() )
    {
    std::cerr << "Unable to open file " << file << std::endl;
    return;
    }

  std::string s, t, p, a;

  while( !infile.eof() )
    {
    s = next_word( infile );
    t = next_word( infile );

    sym.insert( s, str2type(t) );

    if(t=="OP")
      {
      p = next_word( infile );
      a = next_word( infile );

      prec.insert( s, atoi( p.c_str() ) );
      if(a=="r")
        right.push_back(s);
      }
    }

  infile.close();
  }


void loadSem( std::vector<rule> &ru, std::string &nu, std::string &de,
              std::vector< std::vector<std::string> > &se, std::vector<std::string> &po,
              symtable<int> &lo, symtable<int> &ro, std::vector< exprule > &expru,
              std::vector<std::string> &rel, std::vector<std::string> &sign,
              std::vector<entry> &dict, std::string file )
  {
  std::ifstream infile ( file.c_str() );

  if ( !infile.is_open() )
    {
    std::cerr << "Unable to open file " << file << std::endl;
    return;
    }

  std::string s, t, p, type, ltype, rtype, htype, lexp, rexp, hexp, object, cd, name;
  std::vector<std::string> symb, sq;
```

```
exprule e;
entry en;
bool pre;
int n;

while( !infile.eof() )
  {
  s = next_word( infile );

  if(s=="#num")
    {
    nu = next_word( infile );
    }

else if(s=="#dec")
    {
    de = next_word( infile );
    }

else if(s=="#seq")
    {
    t = next_word( infile );
    n = atoi( t.c_str() );
    sq.resize(n);

    for( int i=0 ; i<n ; i++ )
      sq[i] = next_word( infile );

    se.push_back(sq);
    }

else if(s=="#post")
    {
    t = next_word( infile );
    po.push_back(t);
    }

else if(s=="#lop")
    {
    t = next_word( infile );
    p = next_word( infile );
    n = atoi( p.c_str() );
    lo.insert( t, n );
    }

else if(s=="#rop")
    {
    t = next_word( infile );
```

```
    p = next_word( infile );
    n = atoi( p.c_str() );
    ro.insert( t, n );
    }

else if(s=="#rule")
    {
    t = next_word( infile );
    n = atoi( t.c_str() );
    rule r;

    for( int i=0 ; i<n ; i++ )
      r.addName( next_word(infile) );

    r.setHead( next_word(infile) );
    r.setLeft( next_word(infile) );
    r.setRight( next_word(infile) );
    ru.push_back(r);
    }

  else if(s=="#sign")
    {
    sign.push_back( next_word( infile ) );
    }

  else if(s=="#rel")
    {
    rel.push_back( next_word( infile ) );
    }

  else if(s=="#exp")
    {
    t = next_word( infile );
    n = atoi( t.c_str() );
    pre=true;

    if(n<0)
      {
      n=-n;
      pre=false;
      }

    symb.resize(n);
    for( int i=0 ; i<n ; i++ )
      symb[i] = next_word( infile );

    type = next_word( infile );
    ltype = next_word( infile );
```

```
    rtype = next_word( infile );
    htype = next_word( infile );
    lexp = next_word( infile );
    rexp = next_word( infile );
    hexp = next_word( infile );
    e = exprule( symb, type, ltype, rtype, htype, lexp, rexp, hexp, pre );
    expru.push_back(e);
    }

  else if(s=="#cd")
    {
    t = next_word( infile );
    n = atoi( t.c_str() );
    pre=true;

    if(n<0)
      {
      n=-n;
      pre=false;
      }

    symb.resize(n);

    for( int i=0 ; i<n ; i++ )
      symb[i] = next_word( infile );

    ltype = next_word( infile );
    rtype = next_word( infile );
    object = next_word( infile );
    cd = next_word( infile );
    name = next_word( infile );

    en = entry( symb, ltype, rtype, object, cd, name );
    if(!pre)
      en.invOrder();
    dict.push_back(en);
    }

else if( s.size()>0 )
    {
    std::cerr << "Error: unknown token \"" << s << "\"" << std::endl;
    }
  }

infile.close();
}
```

# Literaturverzeichnis


[Chomsky 1956] Chomsky, Noam: *Three models for the description of language.* In: IRE Transactions on Information Theory. Vol.2, 1956, S. 113–124

[Chomsky 1959] Chomsky, Noam: *On certain formal properties of grammars.* In: Information and Control. Vol.2, 1959, S. 137–167

[Vater 1994] Vater, Heinz: *Einführung in die Sprachwissenschaft*. Fink, 1994

[Sedgewick 2002] Sedgewick, Robert: *Algorithms* (3rd ed.). Addison-Wesley, 2002

[Cormen, Leiserson, Rivest, Stein 2007] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford: *Introduction to Algorithms*. 3rd edition, Mit Press 2009

[Standish 1980] Standish, Thomas A.: *Data Structure Techniques*. Addison Wesley Longman Publishing Co, 1980

[Knuth 1969] Knuth, Donald: *The Art of Computer Programming*. Addison-Wesley, 1969–2011

[Knuth 2000] Knuth, Donald: *Computers & Typesetting, Volumes A–E*. Addison-Wesley, 2000

[Dijkstra 1961] Dijkstra, Edsger Wybe: *ALGOL-60 Translation*. Stichting Mathematisch Centrum, ALGOL Bulletin Supplement nr. 10, 1961

[Pratt 1973] Pratt, Vaughan R.: *Top Down Operator Precedence*. Proceedings of the ACM Symposium on Principles of Programming Languages. 1973. pp41–51.

[Samelson, Bauer 1960] Samelson, Klaus, Bauer, Friedrich: *Sequential Formula Translation*. Communications of the ACM CACM Homepage archive, Volume 3 Issue 2, Feb. 1960

[Ginev 2011] Ginev, Deyan: *The Structure of Mathematical Expressions*. 2011

[Ganesalingam 2009] Ganesalingam, M.: *The Language of Mathematics*. 2009

[Hopcroft, Motwani, Ullman 2006] Hopcroft, John E., Motwani, Rajeev, Ullman, Jeffrey D.: *Introduction to Automata Theory, Languages, and Computation*. 3rd edition, Addison-Wesley, 2006

[Hoffmann, O'Donnell 1982] Hoffmann, Christoph M., O'Donnell, Michael J.: *Pattern Matching in Trees*. Journal of the ACM, Volume 29 Issue 1, Jan. 1982, pages 68–95, ACM New York, 1982

[Bachmann 2001] Bachmann, Peter: *Grundlagen der Theoretischen Informatik*. Cottbus 2001, S. 47 (Vorlesungsskript, `http://www.informatik.tu-cottbus.de/~wwwpscb/lehre/GTI/vorl/skript.pdf`

[Kleene 1935] Kleene, Stephen: *A theory of positive integers in formal logic*. In: American Journal of Mathematics. 57, 1935, S. 153–173 und 219–244

[Church 1936] Church, Alonzo: *An unsolvable problem of elementary number theory*. In: American Journal of Mathematics. 58, 1936, S. 345–363 Henk

[Aho, Liam, Sethi 2007] Aho, Alfred V., Liam, Monica S., Sethi, Ravi: *Compilers: Principles, Techniques, & Tools*. 2nd edition, Addison-Wesley, 2007

[Math-Bridge Website] *Math-Bridge Website*. `http://www.math-bridge.org/`

[Openmath Website] *OpenMath Website*. `http://www.openmath.org/`

[Buswell, Caprotti, Carlisle, Dewar, Gaëtano, Kohlhase 2004] Buswell, S., Caprotti, O., Carlisle, D.P., Dewar, M.C., Gaëtano, M., Kohlhase, M.: *The OpenMath Standard*. The OpenMath Society, 2004

[Kofler, Schodl, Neumaier 2010] Kofler, Kevin, Schodl, Peter, Neumaier, Arnold: *Limitations in OpenMath*. Draft, Wien, 2010
`http://www.mat.univie.ac.at/~neum/FMathL/openmath-limitations.pdf`

[FMathL Website] *FMathL Website*. `http://www.mat.univie.ac.at/~neum/FMathL.html`

[Schodl, Neumaier 2011 a] Schodl, Peter, Neumaier, Arnold: *The FMathL type system*. Wien, 2011
`http://www.mat.univie.ac.at/~neum/FMathL/types.pdf`

[Schodl, Neumaier 2011 b] Schodl, Peter, Neumaier, Arnold: *Representing expressions in the semantic memory*. Wien, 2011
`http://www.mat.univie.ac.at/~neum/FMathL/expressions.pdf`

# Lebenslauf

Ich wurde am 10. März 1987 als Sohn von Getrude Groiß und Matthias Langer in Wien geboren. Von 1993 bis 1997 besuchte ich die Volksschule Josefinum und von 1997 bis 2005 das BG XIII Fichtnergasse (Humanistisches Gymnasium), wo ich mit Auszeichnung maturierte. Seit 2005 studierte ich an der Universität Wien Sprachwissenschaft und Mathematik (Schwerpunkt Angewandte Mathematik). Ich war seit 2007 als Studierendenvertreter tätig (davon zwei Mal als gewähltes Mitglied der Studienvertretung Mathematik) und arbeitete seit 2009 als Tutor an der Fakultät für Mathematik.