Ein Verfahren zur Aufzählung binärer Audrucksbäume

Rüdiger Plantiko*

5. Januar 2012

Zusammenfassung

Ein arithmetischer Ausdruck kann durch einen binären Ausdrucksbaum repräsentiert werden. Es ist wohlbekannt, dass es genau C_n arithmetische Ausdrücke mit n Operationen gibt, wobei C_n die n. CATALANschen Zahl ist. In diesem Artikel stelle ich einen Algorithmus zur Konstruktion aller binären Ausdrucksbäume einer gegebenen Grösse vor. Die CATALANschen Zahlen werden durch diesen Algorithmus als *Kombinationen* begreiflich.

Inhaltsverzeichnis

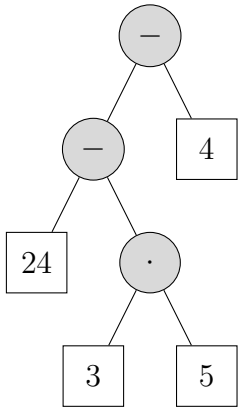
*e-Mail: ruediger.plantiko@astrotexte.ch

1 Binäre Ausdrucksbäume

In diesem Artikel stelle ich ein Verfahren zur Aufzählung *binärer Ausdrucksbäume* vor. Unter einem binären Ausdrucksbaum der Stufe n verstehen ich dabei einen vollen binären Baum mit $n > 0$ inneren Knoten und $n + 1$ Blättern. Er repräsentiert die Festlegung einer bestimmten Klammerung für einen arithmetischen Ausdruck mit n Rechenoperationen und $n+1$ Operanden (z.B. Zahlen). Beispielsweise wird der Term

$$24 - 3 \cdot 5 - 4$$

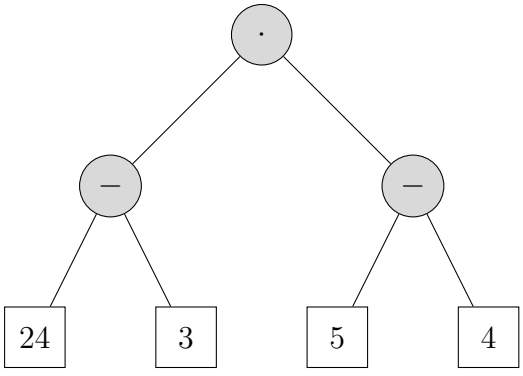
durch folgenden Ausdrucksbaum dargestellt:



Man kann denselben Ausdruck – die Reihenfolge der Zahlen und Operationen beibehaltend – anders klammern, beispielsweise

$$(24 - 3) \cdot (5 - 4)$$

Dann ergibt sich ein anderer Ausdrucksbaum – in diesem Fall der folgende:



Es ist wohlbekannt, dass es C_n Ausdrucksbäume n-ter Stufe gibt, wobei

$$C_n = \binom{2n}{n-1} \cdot \frac{1}{n} \quad (1)$$

$$= \binom{2n}{n} \cdot \frac{1}{n+1} \quad (2)$$

die n. CATALANsche Zahl ist. Diese Folge, die CATALAN (1814-1894) ausführlich beschrieb, die aber auch schon EULER (1707-1783) bekannt war, beginnt mit den Zahlen

$$1, 2, 5, 14, 42, 132, 429, 1430, \ldots \qquad \text{für } n = 1, 2, \ldots$$

In der Form (1) wird sich diese Zahl als Konsequenz aus dem hier vorgestellten Algorithmus ergeben. Vereinfacht gesagt, steht der Binomialkoeffizient $\binom{2n}{n-1}$ für die Möglichkeiten, $n-1$ der insgesamt n inneren Knoten (nämlich alle ausser dem Wurzelknoten) zum rechten oder linken Kind eines anderen inneren Knotens zu erklären: Da jeder innere Knoten zwei Kinder hat, gibt es $2n$ "Enden", in die die $n-1$ inneren Knoten eingehängt werden können.

Allerdings ergibt nicht jede Kombination von $n-1$ Enden einen Ausdrucksbaum. Ich werde aber zeigen, dass unter den n zyklischen Vertauschungen einer beliebigen Kombination genau eine Kombination existiert, die einen gültigen Ausdrucksbaum ergibt. Daraus resultiert die Beziehung (1).

2 Gerichtete Folgen

Definition 1 (Gerichtete Folge) *Unter einer gerichteten Folge der Länge n verstehe ich eine Folge $(a_{n-1}, a_{n-2}, \ldots, a_0)$ von n Zahlen $a_i \in \{0, 1, 2\}$ mit folgenden Eigenschaften:*

$$\sum_{i=0}^{k} a_i \leq k \qquad \text{für} \quad k = 0, \ldots n-2 \quad (3)$$

$$\sum_{i=0}^{n-1} a_i = n \quad (4)$$

Die erste Ungleichung der Kette (3) erzwingt $a_0 = 0$. Aufgrund von (4) liesse sich eine weitere Zahl durch die anderen substituieren. Um der Symmetrie

der Ungleichungen willen ersetze ich diese Abhängigkeiten aber nicht, sondern notiere die gerichteten Folgen in dieser Form.

Die entscheidende Eigenschaft gerichteter Folgen ergibt der folgende

Satz 1 *Sei $(a_{n-1}, a_{n-2}, \ldots, a_0)$ eine Folge von n Zahlen $a_i \in \{0, 1, 2\}$, die nur die Bedingung (4) erfüllt. Dann gibt es im Orbit ihrer zyklischen Vertauschungen*

$$(a_{k-1}, a_{k-2}, \ldots, a_0, a_{n-1}, \ldots, a_k) \qquad \text{für } k \in \{1, \ldots, n\}$$

genau eine gerichtete Folge*, die also auch die Ungleichungen (3) erfüllt.*

Eindeutigkeit. Angenommen, die Folge $(a_{n-1}, a_{n-2}, \ldots, a_0)$ sei gerichtet. Ich zeige dann, dass keine zyklische Vertauschung dieser Folge gerichtet sein kann. Sei also ein $k \in \{1, \ldots, n-1\}$ gegeben. In der k-fach zyklisch permutierten Folge

$$(a_{k-1}, a_{k-2}, \ldots, a_0, a_{n-1}, \ldots, a_k)$$

betrachten wir die Ungleichung (3) für die Stelle a_{n-1} (also dort, wohin der Umbruch zwischen a_{n-1} und a_0 nach der zyklischen Vertauschung gewandert ist). Für die Summe bis zu diesem Glied gilt nun

$$\begin{aligned} \sum_{i=k}^{n-1} a_i &= n - 1 - \sum_{i=0}^{k-1} a_i \\ &\geq n - k \end{aligned}$$

(die erste Gleichheit wegen (4), die Ungleichung dann wegen (3) für $k-1$). Damit die zyklisch permutierte Folge gerichtet wäre, müsste diese Summe aber $\leq n - k - 1$ sein. Das ist ein Widerspruch. □

Der Beweis der **Existenz** einer gerichteten Folge gelingt mit Induktion. Für $n = 1$ ist die Aussage trivial, denn (0) ist die einzige gerichtete Folge der Länge 1. Auch für $n = 2$ ist der Beweis leicht: Die einzig möglichen Folgen $(0, 1)$ und $(1, 0)$ gehen durch zyklische Vertauschung ineinander über, und nur $(1, 0)$ ist gerichtet.

Die Aussage sei bis zur Länge $n - 1$ bewiesen. Sei nun $(a_{n-1}, a_{n-2}, \ldots, a_0)$ eine Folge der Länge n, die nur die Gleichung (4) erfüllt. Zu zeigen ist, dass es dann eine zyklische Permutation der Folge gibt, die alle Ungleichungen der Kette (3) erfüllt.

Da an allen Stellen der Folge nur die möglichen Werte 0,1 oder 2 stehen und diese sich zu n summieren, und da sich die Behauptung bei zyklischer

Vertauschung der ganzen Folge nicht ändert, können wir ohne Beschränkung der Allgemeinheit voraussetzen, dass die Folge eine der Formen

$$(1, a_{n-2}, \ldots, a_0)$$

oder

$$(2, 0, a_{n-3}, \ldots, a_0)$$

hat: dass also eine der Teilfolgen (1) oder (2,0) am Anfang steht. Wenn wir diese Teilfolge vom Rest abtrennen, so stellt dieser Rest eine Folge dar, auf die die Induktionsvoraussetzung angewendet werden kann. Er lässt sich also durch Drehung um eine bestimmte Stellenzahl, sagen wir p, in eine gerichtete Folge verwandeln. Wenn wir nun den abgespaltenen Anfangsteil an der Stelle wieder in die Restfolge einbauen, an der er ursprünglich stand, also vor a_{n-2} im ersten Fall bzw. vor a_{n-3} im zweiten Fall, so ist jede Zahl der gesamten Folge um p Stellen gedreht worden. Man überzeugt sich leicht, dass die Ungleichungen (3) für diese zyklische Permutation erfüllt sind: Bis zur wieder eingebauten Teilfolge (1) oder (2,0) folgt dies aus der Induktionsannahme. Für die nächsten Glieder (1) bzw. (2,0) sind die Ungleichungen dann ebenfalls erfüllt, da die rechten Seiten je Stufe um 1 wachsen, während die linken Seiten um 1 bzw. erst um 0, dann um 2 wachsen. Ab dem darauffolgenden Glied sind beide Seiten der für die Restfolge gültigen Ungleichung jeweils um 2 erhöht, so dass sie weiterhin erfüllt bleibt. □

3 Schema und Signatur

Die inneren Knoten eines Baumes lassen sich "von links unten beginnend" aufzählen. Hierzu kann man wie folgt vorgehen:

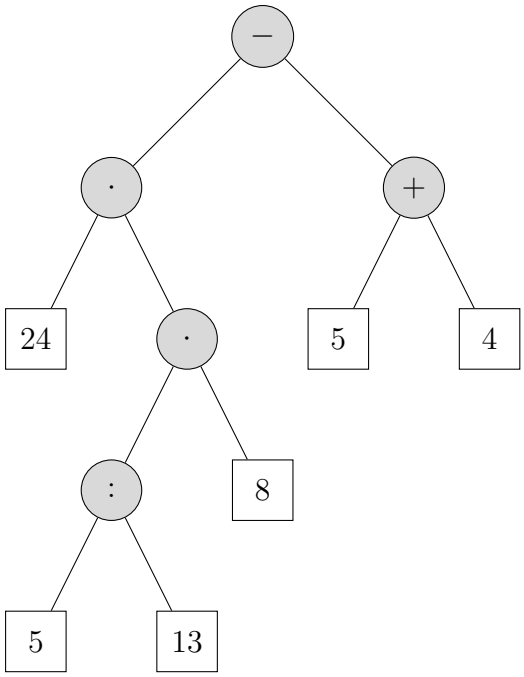
1. Definition des Verfahrens "*Startknoten zu einem Bezugsknoten finden*":
   (a) Beginnend beim Bezugsknoten, geht man solange zum linken Kindknoten über, solange dieser ein innerer Knoten ist.
   (b) Wenn dieser Knoten zwei Blätter als Kinder hat, ist er der Startknoten. Wenn nicht, wiederholt man das Verfahren mit dem rechten Kind dieses Knotens als Bezugsknoten.
2. Man beginnt die Aufzählung mit dem Startknoten zum Wurzelknoten. Dies ist der erste "aktuelle Knoten".

3. Der Vaterknoten des aktuellen Knotens wird der nächste aktuelle Knoten, wenn der andere Kindknoten bereits aufgezählt oder ein Blatt ist.

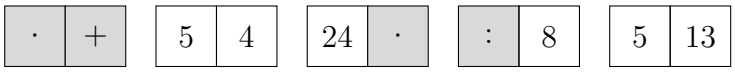
4. Wenn das andere Kind ein innerer Knoten ist, der noch nicht aufgezählt wurde, so finde dessen Startknoten nach dem Verfahren 1 und mache diesen zum aktuellen Knoten.

Nach diesem wohlbestimmten und terminierenden Verfahren lassen sich alle n inneren Knoten eines gegebenen binären Ausdrucksbaums zusammen mit den Vater-Kind-Beziehungen, die den Ausdruck charakterisieren, aufzählen. Die Aufzählung lässt sich in Form eines Schemas aus n Doppelkästchen notieren, wobei jedes Doppelkästchen die beiden Kinder eines inneren Knotens enthält. Die Kästchen können gemäss ihrer Aufzählungsreihenfolge am besten entgegen der Schreibrichtung, also von rechts nach links notiert werden, damit das Schema immer mit dem Wurzelknoten beginnt.

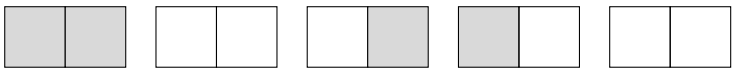
Sei beispielsweise der folgende binäre Ausdrucksbaum mit fünf inneren Knoten gegeben.



Daraus resultiert das folgende Schema:



Weil das Schema nur die Referenzen notiert, ist die Operation des Wurzelknotens (−) selbst nicht mehr im Schema enthalten. Das spielt aber für die Frage danach, wieviele verschiedene binäre Ausdrucksbäume es *strukturell* gibt, keine Rolle – ebensowenig wie die konkrete Besetzung des ganzen Schemas mit Zahlen und Operationen. Die Gestalt des Ausdrucksbaums ist bereits vollständig durch folgendes, auf die Besetzung der Knoten reduziertes Schema bestimmt:



Tatsächlich lässt sich der Baum, von rechts nach links vorgehend, wieder aus diesem Schema ableiten. Es ist wichtig, sich das Vorgehen für diese Umkehrung genau klarzumachen, denn es wird Anlass für den Algorithmus zum Aufbau aller binärer Bäume geben:

- Wir arbeiten uns von rechts nach links durch das Schema durch. In jedem Schritt werden die Operanden der nächsten Operation festgelegt. Diese können entweder Blätter oder in früheren Schritten fertiggestellte Operationen sein.
- Neben dem entstehenden Ausdrucksbaum haben wir einen Stapel für noch nicht verbaute Operationen. Dieser Stapel ist am Anfang leer.
- Bei jedem Schritt kommt die aktuelle Operation, da sie in diesem Schritt noch nicht in den Baum eingebaut wird, auf den Stapel.
- Wenn man für einen Knoten auf ein Operationskästchen (im Schema grau gefärbt) trifft, besetzt man diese Plätze durch Operationen, die man der Reihe nach vom Stapel nimmt.
- Es müssen bei jedem Schritt mindestens so viel freie Operationen auf dem Stapel verfügbar sein, wie es im aktuellen Schritt mit Operationen zu besetzende Kindpositionen gibt.
- Am Ende darf nur noch eine Operation auf dem Stapel liegen: Der Wurzelknoten.

Nicht graphisch könnte man den Baum daher auch als eine Folge der Länge n darstellen, wobei jedes Element angibt, wieviele Kindknoten des aktuellen Knotens mit inneren Knoten besetzt sind. Ist nur ein Kindknoten innerer

Knoten, muss man allerdings das linke vom rechten Kind unterscheiden. Der obige binäre Ausdrucksbaum lässt sich also auch durch die Folge

$$(\mathbf{2}, \mathbf{0}, \mathbf{1^R}, \mathbf{1^L}, \mathbf{0}) \tag{5}$$

charakterisieren, die ich die *Signatur* des Ausdrucksbaums nennen möchte.

Kombinatorisch gesehen, haben wir die $n-1$ inneren Knoten (also den Wurzelknoten ausgenommen) auf die insgesamt $2n$ verfügbaren Plätze "linkes bzw. rechtes Kind" der Operationen verteilt, also eine *Kombination* im Sinne der Kombinatorik gebildet.

Satz 2 *Die möglichen binären Ausdrucksbäume n. Stufe entsprechen eindeutig den Orbits der Menge aller Kombinationen "$(n-1)$ aus $2n$" unter der zyklischen Vertauschungsoperation um 2 (der $2n$) Stellen. Insbesondere gibt es daher insgesamt*

$$\binom{2n}{n-1} \cdot \frac{1}{n}$$

binäre Ausdrucksbäume n. Stufe.

Zum **Beweis** muss nur alles Vorangehende zusammengetragen werden: Ein binärer Ausdrucksbaum lässt sich eindeutig einer Signatur (wie oben (5)) zuordnen. Aber nicht alle $(n-1)$-Tupel aus der Menge $\{\mathbf{0}, \mathbf{1^L}, \mathbf{1^R}, \mathbf{2}\}^{n-1}$ bilden gültige Signaturen eines Ausdrucksbaums. Aus dem oben beschriebenen Vorgehen zum Aufbau eines Baumes folgt:

Eine Signatur ist genau dann die Bauanleitung für einen binären Ausdrucksbaum, wenn im k. Schritt die Zahl der bis zum aktuellen Knoten aufgetretenen Operationskästchen (im Schema die grau gefärbten Stellen), kleiner als k ist, im n. Schritt aber gleich $n-1$ ist. Anders ausgedrückt, muss das Schema eine *gerichtete Folge* ergeben (wobei wir die Superscripte L und R in der Signatur ignorieren, so dass wir die Signatur auf eine Folge von Zahlen aus $\{0, 1, 2\}$ reduziert haben.) Dann und nur dann, wenn die Signatur eine gerichtete Folge ergibt, kann aus dem Schema ein binärer Ausdrucksbaum konstruiert werden.

Nun lässt sich *jeder* Kombination "$(n-1)$ aus $2n$" ein Schema und jedem Schema eine Signatur zuordnen. Wie oben bewiesen, gibt es genau eine zyklische Vertauschung, die diese Signatur zu einer gerichteten Folge macht. Bei entsprechender zyklischer Vertauschung des Schemas lässt sich dann ein binärer Ausdrucksbaum aufbauen. □

4 Der Algorithmus in C

Ich habe im letzten Abschnitt gezeigt, dass es eine Bijektion zwischen den binären Ausdrucksbäumen und den Schemata gibt, deren Signatur eine gerichtete Folge ist. Für die Zuordnungen vom Ausdrucksbaum zum Schema und umgekehrt habe ich Vorgehensweisen beschrieben, die zueinander inverse Abbildungen darstellen. Die Vorgehensweise zum Aufbau von Ausdrucksbäumen aus Schemata lässt sich zu einem Algorithmus ausbauen, der alle Ausdrucksbäume einer Stufe aufzählt.

In diesem Abschnitt beschreibe ich die Realisierung dieses Algorithmus in C. Ich benutze das Feature der dynamischen Arraygrössen aus dem C99-Standard,[1] was die Lesbarkeit des Codes erhöht.

Die Idee ist, "von unten" beginnend, den Baum stufenweise aufzubauen. In jeder Stufe werden nacheinander die vier Varianten $\{\mathbf{0}, \mathbf{1^L}, \mathbf{1^R}.\mathbf{2}\}$ rekursiv entfaltet. Beim Eintritt in eine neue Stufe wird geprüft, ob die Signatur noch eine gerichtete Folge ergibt. Für die Zweige, die es bis zum Wurzelknoten schaffen, wird die Referenz auf den Wurzelknoten an eine Besucherfunktion übergeben, hier z.B. `printNode` zum Ausdruck des Baums.

Der Typ eines Knotens kann sehr elementar gehalten werden. Für jeden Knoten müssen wir wissen, ob es sich um ein Blatt oder um einen inneren Knoten handelt. Das ergibt die Boolesche Variable `isLeaf`. Ausserdem müssen Referenzen auf den linken und rechten Kindknoten festgehalten werden. All dies zusammengenommen, ergibt den folgenden neuen Datentyp `node`:

```
typedef struct node {
  bool isLeaf;
  struct node* left;
  struct node* right;
  } node;

// A node initially is a leaf
const node initialNode = {
  .isLeaf = true,
  .left=0,
  .right=0
  };
```

Die Referenz auf die Besucherfunktion, an die die vollständigen Bäume über-

[1] In der GNU Compiler Collection muss man demnach zum Compilieren die Option `-std=C99` angeben.

geben werden, lässt sich folgendermassen beschreiben:

```
typedef void(*visitor)(node*);
```

Für den Algorithmus benötigen wir eine Liste, die Platz für sämtliche $2n+1$ Knoten bietet, und einen Stapel, der sicher nicht mehr als n Knoten enthalten kann (die Gesamtzahl der inneren Knoten). Den hierfür benötigten Speicher können wir am Beginn, noch vor dem Abstieg in die Rekursion, reservieren. Den Stapel realisieren wir als Liste von *Zeigern auf Knoten*, da es unnötig ist, die Knoten in den Stapel zu kopieren. Der Algorithmus beginnt "hinten" – nach dem letzten Listenelement. Damit haben wir auf oberster Ebene folgende Funktion:

```
void getExpressions(int size, visitor f) {

// Allocate the complete tree as an array
  node nodes[2*size+1];

// Maximum # of stacked nodes = # of operations
  node* stack[size];

// Start with pointer ``end'' = after the last element
  node* current = nodes + 2*size+ 1;

// Go up!
  doNode( current, 0, stack, nodes, f );

}
```

Die Rekursion erfolgt über die Funktion `doNode()`. Sie erhält von der ihr übergeordneten Aufrufebene

1. eine Referenz auf den aktuellen Knoten,
2. die aktuelle Grüsse des Stapels,
3. sowie Zeiger auf Stapel, Knotenliste und Besucherfunktion

Bei hinreichend kleiner Stapelgrösse – es darf, wie beschrieben, nicht mehr Knoten auf dem Stapel geben als aktuell noch zu erzeugen sind – wird der Zeiger für den aktuellen Knoten auf das nächste Listelement gesetzt. Dann werden nacheinander die vier rekursiven "Aufstiege" aufgerufen, bis der Wurzelknoten erreicht ist. Dann liegt der nächste vollständige Ausdrucksbaum vor und kann der Besucherfunktion `f` übergeben werden.

```
void doNode( node* current, int stackSize,
             node* stack[], node* root, visitor f ) {

  int currentIndex = current - root;

// Stack too large:
// it will not be possible to connect all references
  if (stackSize > currentIndex + 1) return;

  if (currentIndex > 0) {

    current--;

// First option: Go on with a node with 2 leaves
    if (currentIndex >= 2) {
      addNodeWithTwoLeaves(
        current, stackSize, stack, root, f );
      }

// If there is (at least) one usable reference node
// with 1 leaf, 1 ref
    if (currentIndex >= 1 && stackSize >= 1) {
      addNodeWithOneLeaf( LEFT,
        current, stackSize, stack, root, f );
      addNodeWithOneLeaf( RIGHT,
        current, stackSize, stack, root, f );
      }

// If there are two node references available:
// use them in the current node
    if (stackSize >= 2) {
      addNodeWithNoLeaves(
        current, stackSize, stack, root, f );
      }

  }

  else {

// Terminal point: Do something with the expression
    f( root );

    }

  }
```

Die Funktionen `addNode...()` sind auf die naheliegende Weise implementiert: Sie erzeugen die benötigte Anzahl von Blättern und bilden die benötig-

ten inneren Knotenreferenzen, indem sie Elemente vom Stapel nehmen. Dann legen sie die Referenz auf den gerade gebildeten neuen inneren Knoten auf den Stapel und rufen `doNode()` auf. Es folgt hier beispielhaft die Funktion `addNodeWithOneLeaf()`:

```
void addNodeWithOneLeaf( childPos side, node* current,
                         int stackSize, node* lastStack[],
                         node* root, visitor f ) {

  node *stack[stackSize];
  for (int i=stackSize-1;i>=0;i--) stack[i] = lastStack[i];

// Build one leaf
  *current = initialNode;
  current--;

  *current = initialNode;
  current->isLeaf = false;  // tag as operation

// Make one of the child nodes point to the next free op
  switch (side) {
    case LEFT:
      current->left  = stack[stackSize-1];
      current->right = current+1;
      break;
    case RIGHT:
      current->right = stack[stackSize-1];
      current->left  = current+1;
      break;
    default:
      assert( false ); // Not allowed
    }

// Replace top of stack by new node
  stack[stackSize-1] = current;

  doNode( current, stackSize, stack, root, f );

  }
```

Eine Besonderheit der Implementierung in C ist hier noch zu bemerken: Der Stapel wird ja in der Funktion verändert. Bei der Rückkehr aus der Funktion sollte aber die vorherige Version des Stapels wiederhergestellt werden. Für eine *by value* Übergabe wäre das automatisch. Da C aber keine Wertübergabe von Arrays unterstützt (und das ist in 99% der Fälle auch gut so), muss zu Beginn der Funktion eine lokale Kopie des Stapels gebildet werden, mit der dann in diesem Ausführungszweig weitergearbeitet wird.

In der Funktion `addNoteWithOneLeaf` gibt es einen zusätzlichen Parameter `side` vom Aufzählungstyp `childPos`, der angibt, auf welcher Seite der innere Knoten einzufügen ist:

```
// Constants for child's position
typedef enum { LEFT, RIGHT } childPos;
```

Da Aufzählungen aber in C nicht typsicher sind, sollte der `default`-Zweig des `switch/case` auf jeden Fall aufgefürt und mit einer Assertion versehen sein (wie ich es oben bei der Verwendung gemacht habe).

Üblicherweise wird eine binäre Operation als 3-Tupel (oder Array der Länge 3) notiert:

```
[ op, arg1, arg2]
```

Definiert man eine Funktion `printNode`, die die Knoten und ihre Argumente in dieser Notation ausgibt, so erhält man bei Aufruf von `getExpressions` z.B. mit size = 4 die folgende Auflistung der 14 Ausdrucksbäume mit vier inneren Knoten:

```
[op,[op,[op,num,num],num],[op,num,num]]
[op,[op,num,[op,num,num]],[op,num,num]]
[op,[op,[op,num,num],[op,num,num]],num]
[op,num,[op,[op,num,num],[op,num,num]]]
[op,[op,num,num],[op,[op,num,num],num]]
[op,[op,[op,[op,num,num],num],num],num]
[op,num,[op,[op,[op,num,num],num],num]]
[op,[op,num,[op,[op,num,num],num]],num]
[op,num,[op,num,[op,[op,num,num],num]]]
[op,[op,num,num],[op,num,[op,num,num]]]
[op,[op,[op,num,[op,num,num]],num],num]
[op,num,[op,[op,num,[op,num,num]],num]]
[op,[op,num,[op,num,[op,num,num]]],num]
[op,num,[op,num,[op,num,[op,num,num]]]]
```

Die vollständige Version des hier vorgestellten Programms `etrees.c` ist auf Pastebin unter der URL `http://pastebin.com/mehTM4Z0` verfügbar.