Transform Expressions to Standard

David Vajda

26. Oktober 2021

Jetzt mache ich aus dem, was ich auf Facebook geschrieben habe, ein Abstract in LaTex, für meine Homepage.

```
https://www.ituenix.de/reverscompiler23.html
https://www.ituenix.de/reverscompiler24.html
```

Jezt poste ich das erst auf meiner Homepage.

```
(4+3)+8+9
(+(*(+(*31)(+(*41)0))1)(+(*81)(+(*91)0)))
```

Jetzt das Problem ist, das mit der Klammersetzung das wollte so nicht, das ist klar. Man kann die Klammern nicht da setzen - wo ich das gemacht habe - das will ich nicht erläutern. Jetzt sind meine Ausdrücke speziell. Da ich die Linksrekursion beseitige - tue ich am Ende von einer Multiplikation immer eine 1 multiplizieren und bei einer Addition immer eine 0. Das ist bedingt durch die Beseitigung der Linksrekursion. Zeigt einen Vorteil. In Ausdrücken bedeutet das das Ende des Teilausdrucks. So, jetzt besteht die Idee, da ich geschachtelte Präfix-Ausdrücke habe, bei jeder Addition und bei der jeder Multiplikation einfach am Anfang eine Klammer zu öffnen. Beginnt die Addition. Öffne ich eine Klammer, beginnt die Multiplikation, vor jedem Operator, ich öffne Klammer. Jetzt ist die Frage - wo schließe ich die. Und wie oft. Und das ist das eigentliche Problem. Die Klammern zu öffnen ist kein Problem, sie zu schließen - ein größeres. Aber die Idee ist einfach. Da jeder Term - also Multiplikation mit einer 1 endet und jeder Ausdruck (Expression) mit einer 0 endet, da das automatisch so ist, schließe ich mit jeder 1 und jeder 0, die Klammer. Die Frage ist wie oft - und die Antwort ist einfach: So oft ich addiert und multipliziert habe. Das klingt für viele schräg - weil für viele heißt: Compiler - wir haben keine Schleifen - ist in dem Fall nicht richtig - und ich weise sie daraufhin - haben sie keine Angst! Dass ich jetzt mit zählen und Schleifen arbeite, heißt, dass das Gesetz von Compilern irgendwo beschädigt ist - das ist es so oder so nicht - sie können den Code lesen, und wenn sie das tun werden sie sehen, das ist nicht der Fall. Das ist eine ideologische Frage - aber ideologisch ist es auch nicht beschädigt. Jetzt zähle ich mit: Wie oft addiere ich - wie oft multipliziere ich. Und mit jeder Addition und jeder Multiplikation, zähle ich mit. Wenn die 1 folgt, bei der Multiplikation schließe ich die Klammer so oft - bis die Zählung der

Multiplikation 0 ergibt. Ebenso bei der Addition, mit 0, am Ende. Es ist klar, dass ich zwei Zähler habe, einen für Adition und einen für Multiplikation. Jetzt ist das Problem gelöst. Wenn ich 1+2+3+4 habe oder 2*3*4*5, dann ist das Problem gelöst. Es wird jetzt aber noch ein Mal kritisch. Nämlich dann, wenn ich Klammern im Ursprünglichen Ausdruck habe, oder das Distributivgesetz. Weil (1+2+3+4) macht Probleme. Aber auch (1+2+3+4)*5 Das Problem ist: Expression wird hier innerhalb von Factor erneut aufgerufen. Mit Klammern. Wenn ich Factor Klammern festgestellt werden, dann wird erneut Expression aufgerufen. Jetzt werden sie sehen, dass das ideologisch nichts abtut - weil schleifen widersprechen ideologisch dem Prinzip von push() und pop() was wir bei Rekursionen erwarten. Tatsächlich können wir die Zähler, nachdem wir expr() innerhalb von factor aufrufen, nicht weiter verwenden. Weil es beginnt ja ein neuer Teilausdruck. Der braucht seine eigene Zähler. Also, was tun? Push und Pop für die Zäher? Das ist im Prinzip richtig. Und die Antwort lautet: Ja, PUSH und POP. Allerdings: Die Implementierung lässt zu wünschen übrig. PUSH und POP ein zu führen, ist hässlich. Aber, da wir so oder so Rekursionen haben, und lokale Variablen, die sich nicht überschreiben, bietet sich etwas anderes an: Wir übergeben, die Zähler, an die Funktion, expr(), expr2(), term, term2(). Factor ist davon nicht betroffen. Das hat keine Übergabeparameter. Aber egal, ob wir expr() zum ersten Mal, bei uns aufrufen, oder innerhalb von factor(), wir rufen es auf mit expr(0,0). Dann sind unsere Zähler mit 0, 0 intilasiert. Die zwei Zähler, werden unabhängig von expr() und term() mit beiden Funktionen, an die anderen weiter geben. Blos zählt die eine die Addition, die andere die Multiplikation.
Erstens die Sortierung funktioniert.

```
4*2*3*1
(+(*1(*2(*3(*41))))0)

1+2+3+4
(+(*11)(+(*21)(+(*31)(+(*41)0))))
1*2*3*4
(+(*1(*2(*3(*41))))0)
4*2*3*1
(+(*1(*2(*3(*41))))0)
(1*2*3*4)
(+(*(+(*1(*2(*3(*41))))0)1)0)
(1*2*3*4)*2
(+(*(+(*1(*2(*3(*41))))0)(*21))0)
```

Erst ein paar Beispiele.
Lustig, ich muss ihnen die Geschichte - das ist für den Compiler eine sehr interessante Geschichte.
Ich erzeuge jetzt prefix-Ausdrücke, die zu 100 Prozent richtig sind. Präfix heißt:

```
(+ 1 2 3 4) f"ur (1+2+3+4)
```

Aber, meine Präfixausdrücke sind speziell. Ich schreibe nicht

```
(+ 1 2 3 4)
(+ 1 (+ 2 ( + 3 4)))

(+
    (*
        (+
            (*
                1
                1
            )
        )
            (+
                (*
                    2
                    1
                )
                (+
                    (*
                        3
                        1
                    )
                    (+
                        (*
                            4
                            1
                        )
                        0
                    )
                )
            )
        )
    )
)
```

Yeaapeeah! Das Leiden war nicht umsonst.
Da habe ich mich ja wohl zu früh gefreut, ich habe meinem eigenem Konzept wiedersprochen - weil, ich kann die ¡br¿ ja nicht entfernen - danach wird ja sortiert. Es gibt allerdings eine Möglichkeit - gerade da ¡br¿ ein Trenner ist, kann ich zu jedem a und b ein + einfügen - und das wird dann mit sortiert.
Sie in ein nützliches Resultat zu verwandeln - dazu habe ich schon eine Idee - wir geben eben doch nach jedem a und b im Code, je nach + und * ein + und * aus, dazwischen, nur nach 0 und 1 nicht.
Also, jetzt wollen wir den Compiler verbessern - wir probieren die Ausgabe zu verbessern - es gibt dazu zwei Möglichkeiten

1. Wir verbessern, die tatsächliche Ausgabe von Anfang an

2. Wir nehmen die bisherige und verwandeln sie in ein nützliches Resultat.

Wie sie vielleicht sehen, ist der Ausdruck nicht unbedingt mit den * und + am Anfang und in der Mitte zu 100 Prozent so zu gebrauchen. Man könnte zu der Überzeugung gelangen, dass gleichzeit infix und prefix Operatoren gemischt verwendet werden. Das kann man verbessern. Die Ausgabe ist sicher nicht zu 100 Prozent perfekt, bisher. Man könnte sich jetzt auf den Infix-Operator

```
(+ 5 4 2 3)
```

für

```
5+4+2+3
```

einigen. Das andere ist, selbst, wenn das nicht zu 100 Prozent perfekt ist, jetzt, dann lasse ich es einerseits dem Nutzer offen, das selber zu interpretieren - er wird seine Interpretation finden. Auf der anderen Seite, kann man das mit wenigen Handgriffen machen. Die Hauptsache ist gelöst worden. Es kommt bei verschiedenen Ausdrücken derselbe Ausdruck raus - das ist die Hauptsache und falsch ist er nicht. Schaut man die Vorgängerversion an, dann stimmt die Reihenfolge der Opranden - und es werden nur Operatoren dazu eingefügt. Die kann man auch einfach anders einfügen. Das mache ich heute abend nicht. Für heute ist das erledigt. Das mache ich vielleicht morgen. Das andere ist: Dass das mit wenigen Handgriffen gemacht ist, aber der Nutzer auch die Möglichkeit haben soll, das selber zu interpretieren. Das öffnet ihm den Weg. Es ist oft nicht weg ebnend, gar nicht selber zu denken. Und oft führt gerade so ein weg, zum Denken. Auch bei mir. Entscheidend ist, dass dieser Ausdruck zu Stande kommt. Es gibt mehrere Möglichkeiten den Ausdruck weiter zu verarbeiten - also, wie gesagt, vielleicht kann man noch an der Darstellung arbeiten, auf jeden Fall kann man sie lesbarer machen. Das geht dadurch, dass man entweder infix Operatoren oder Präfix Operatoren einfügt. Auf der anderen Seite: Gibt es dazu zwei Möglichkeiten: Die jetzige algorithmische Umwandlung, kann beinhalten, dass Präfix Infix bevorzugt besser darstellbar ist, oder umgekehrt. Und selbst, wenn die Darstellung jetzt noch verbessert werden kann, gibt es dazu zwei Möglichkeiten. Das erste ist: Wir haben eine eindeutige Darstellung, die auch richtig ist. Wir können auf jeden Fall, zwei Ausdrücke vergleichen. Trotzdem können wir die Darstellung als richtig betrachten. Um sie besser zu machen, geht es, indem wir an dem bisherigen Code arbeiten, indem wir etwas hin zu fügen, oder verändern, gerade bei der Ausgabe - oder wir lassen, und führen so zu sagen, eine zweite Maschine. Diese zweite Maschine, die Umwandlung in präfix und infix machen, oder sie kann sogar überhaupt präfix zu infix machen und umgekehrt. Was jetzt wichtig ist: Wir haben jetzt erst Mal bei Termen 3 Zusammenhänge

1. Kommutativgesetz
2. Assoziativgesetz
3. Distributivgesetz.

Was für uns immer besonders schön aussieht, ist das Distributivgesetz. Das macht uns so mit der Anwendung mit der Hand am meisten Spaß. Kommutativgesetz und Assoziativgesetz benutzen wir oft, vielleicht, oder: Es steht einfach im Raum. Das wirkt wichtig, besonders, das Kommutativgesetz, aber wenn wir jetzt einen Ausdruck hinschreiben, klar können wir jetzt Assoziativität machen - aber: Das machen wir einfach so. Dabei wird es gerade, die beiden, genau zusammen - man muss das zusammen sehen - beim Computer ausgerechnet wichtig, Assoziativgesetz und Kommutativgesetz zusammen zu verwenden. Das Assoziativitätsgesetz ist uns automatisch mit dem Vorgänger, nämlich dem Reverse-Compiler gelungen. Wenn wir den Ausdruck zurückübersetzen, verliert er automatisch seine Klammern. Das sagt nichts darüber, ob zwei Ausdrücke, die gleich sind, aber umgestellt, vom Computer als gleich erkannt werden. Auf dem Computer ist das Distributivgesetz, etwas Fizelarbeit. Mehr Fitzelarbeit. Und trotzdem genügt es nicht. Natürlich würde sich der Nutzer über das Kommutativgesetz freuen - aber ist längst nicht so elementar und fundamental, wie Kommutativ und Assoziativgesetz, in Wirklichkeit können wir uns damit anfreunden, dass Produkte von Summen nicht aufgelöst wurden, wir können uns nicht damit anfreunden, dass auf dem Computer eben, nur, weil Ausdrücke anders stehen und weil, Klammern anders gesetzt sind, Ausdrücke nicht identisch erkannt werden, obwohl sie es sind. Auch für das Distributivgesetz, müssen die Kriterien, Kommutativ und Assoziativ erfüllt sein. Wir könnten zwar sagen: Nachher sind alle Summanden aufgelöst. Sie stehen damit nicht gleicher Reihenfolge, somit müssen wir so oder so Kommutativgestz anwenden. Das alleine macht die Sache aus. Was jetzt bedienst ist, ist Kommutativgesetz und Assoziativgesetz - Distributivgesetz, ist ein wenig mehr Fummeln, dafür unkomplizierter, aber das geht dann auch. Jetzt zum Compiler. Wir haben verschiedene Ausgaben. Das dürfen wir nicht vergessen:

1. Maschinensprachen-Output
2. Die Funktionen geben bei arithmetischen Ausdrücken selber die Werte zurück, indem sie addieren und multiplizieren. Rückgabewert
3. Wir können einen Interpreter im Code selber laufen lassen
4. Wir können einen Syntaxbaum erstellen
5. Einen Syntaxbaum mit Klassen - in dem expr(), term(), factor() selber Klassen sind
6. Die Stackmaschine

Das sind unterschiedliche Ausgabeformate. Jetzt etwas anderes zu den Ausdrücken. Meiner Meinung nach habe ich gehört, man spricht von Standard-Ausdrücken. Ich weiß nicht, ob es das Wort gibt. Meiner Meinung nach habe ich das mal gehört - wenn es das ist, was ich jetzt meine, dann weiß ich es jetzt. Es gibt viele Möglichkeiten

```
(6+5)*(4+3+2+1)
```

dar zu stellen: Die eine Möglichkeit wäre:

```
(1+2+3+4)*(5+6)
```

Das wäre wohl eine günstige Form - für den Standard, so zu sagen. Weil man schreibt einfach den Ausdruck mit den kleineren Summanden zu erst, so, dass alles sortiert ist. Das ergäbe einen gewissen Sinn. Eine andere Frage, wäre, wie viele Elemente, hat diese Menge. Und die Antwort ist erstaunlich: Die größte vorstellbare Menge, die wir uns vorstellen können: ist die Menge von Ausdrücken. Das macht einen Sinn. Denn wir haben die Standardform des Ausdruckes - und wir betrachten jeden Ausdruck als Element einer Menge. Weil uns nicht mehr egal ist, wie der sortiert ist. Bei den rationalen Zahlen kennen wir den üblichen weg. Genau haben wir N x N natürliche Zahlen. Wir kennen ja diese Matrix, wo wir mit den Pfeilen von einem zum anderen gehen. Da wir aber n/m haben, sind es N x N. Die Frage ist, was passiert bei unserem Ausdruck. Das werden entsprechend der Summanden NxNx...xN. Da ein Ausdruck aber beliebig lang werden kann, unendlich lang - ist die Menge gigantisch. Die Menge der Reellen Zahlen ist viel größer als die Menge der rationalen. Was uns klar ist, ist, dass jeder irrationale Zahl, über Summen dargestellt werden kann. Das heißt, wir berechnen, Pi und Wurzel (2) als Reihe. Das heißt, als Summe. Da es unendlich viele Summen, auf beliebig rationale Zahlen gibt, ist die Menge gigantisch. Was allerdings bei einer Zahl, keine Rolle spielt

```
2+3 = 5
3+2 = 5
```

So ist, 3.14... immer 3.14, das heißt die Zahl ändert sich nicht. Wenn wir den Ausdruck als Menge begreifen. Spielt es dann allerdins sehr wohl eine Rolle, ist es:

```
2+2+1 = 5
3+2 = 5
2+1+2 = 5
...
```

Hier ist die Summe nicht, wie bei der reellen Zahl, einfach die Zahl.
Ich denke, ich gehe jetzt ein bisschen raus - das habe ich mir verdient.

```
https://www.ituenix.de/final-compiler-2021-10-15.zip
https://www.ituenix.de/final-compiler-2021-10-15.tar.gz
https://www.ituenix.de/reverscompiler21.html
https://www.ituenix.de/reverscompiler22.html
```

Erst mal gucken, ob es funktioniert hat, vor der Aufregung - und es hat funktioniert.
Es hat doch funktioniert keine Aufregung.

```
(2+(4+3+2+1)*(8+6+5+4)*(4+6+5+8))*3
(+(*((+(*((+(*(1)1)+(*(2)1)+(*(3)1)+(*(4)1)0))*((+(*(4)1)+(*(5)1)+
```

```
(*(6)1)+(*(8)1)0))*((+(*(4)1)+(*(5)1)+(*(6)1)+(*(8)1)0))1)+(*(2)1)0))*(3)1)0)
3*((8+6+5+4)*(4+3+2+1)*(8+6+5+4)+2)
(+(*((+(*((+(*(1)1)+(*(2)1)+(*(3)1)+(*(4)1)0))*((+(*(4)1)+(*(5)1)+(*(6)1)
+(*(8)1)0))*((+(*(4)1)+(*(5)1)+(*(6)1)+(*(8)1)0))1)+(*(2)1)0))*(3)1)0)
```

```
(8+6+5+4)*(4+3+2+1)*(8+6+5+4)
(+(*((+(*(1)1)+(*(2)1)+(*(3)1)+(*(4)1)0))*((+(*(4)1)+(*(5)1)+
(*(6)1)+(*(8)1)0))*((+(*(4)1)+(*(5)1)+(*(6)1)+(*(8)1)0))1)0)
(4+3+2+1)*(8+6+5+4)*(4+6+5+8)
(+(*((+(*(1)1)+(*(2)1)+(*(3)1)+(*(4)1)0))*((+(*(4)1)+(*(5)1)+
(*(6)1)+(*(8)1)0))*((+(*(4)1)+(*(5)1)+(*(6)1)+(*(8)1)0))1)0)
```

```
(8+6+5+4)*(4+3+2+1)*(8+6+5+4)
(+(*((+(*(1)1)+(*(2)1)+(*(3)1)+(*(4)1)0))*((+(*(4)1)+(*(5)1)+
(*(6)1)+(*(8)1)0))*((+(*(4)1)+(*(5)1)+(*(6)1)+(*(8)1)0))1)0)
```

```
https://www.ituenix.de/sweetexpr.zip
https://www.ituenix.de/sweetexpr.tar.gz
https://www.ituenix.de/reverscompiler17.html
```

```
(8+6+5+4)*(4+3+2+1)*(8+6+5+4)
(+(*((+(*(1)1)+(*(2)1)+(*(3)1)+(*(4)1)0))*((+(*(4)1)+(*(5)1)+
(*(6)1)+(*(8)1)0))*((+(*(4)1)+(*(5)1)+(*(6)1)+(*(8)1)0))1)0)
```

Aus

```
(8+6+5+4)*(4+3+2+1)*(8+6+5+4)
```

wurde:

```
1 1 1 2 1 3 1 4 0 1 4 1 5 1 6 1 8 0 1 4 1 5 1 6 1 8 0 1 0
```

```
https://www.ituenix.de/reverscompiler15.html
```

```
https://www.ituenix.de/sortexprthebighit.tar.gz
https://www.ituenix.de/sortexprthebighit.zip
```

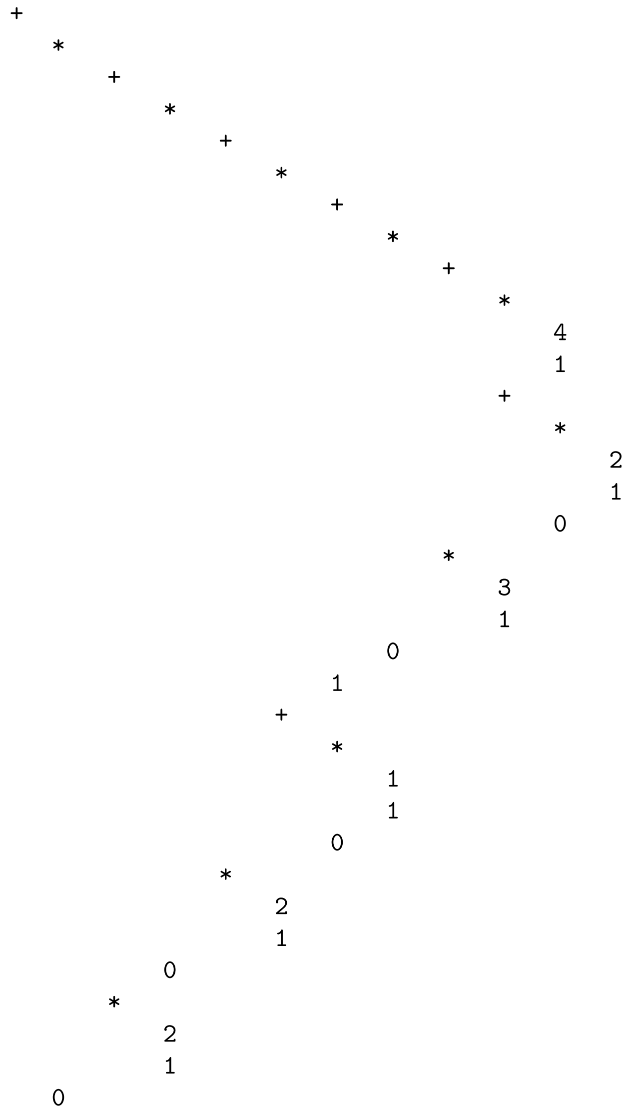
```
https://www.ituenix.de/reversecompiler.zip
https://www.ituenix.de/reversecompiler.tar.gz
```

Das ist der korrekte Syntaxbaum:

```
+
    *
        +
            *
                +
                    *
                        +
                            *
                                +
                                    *
                                        4
                                        1
                                    +
                                        *
                                            2
                                            1
                                        0
                                *
                                    3
                                    1
                            0
                        1
                    +
                        *
                            1
                            1
                        0
                *
                    2
                    1
            0
        *
            2
            1
    0
```

Also, wie gesagt, das Programm funktioniert - das war ein Rechtschreibfehler - lange Suche, wo ist das Problem - und am Ende, zeigt sich, das Problem ist ein Rechtsschreibfehler - stundenlanges suchen und am Ende stelle ich fest - Anweisungen werden nicht mehr ausgeführt - wenn das in JavaScript passiert, dass Anweisungen, bis zu einem gewissen Punkt ausgeführt werden und dann nicht mehr - dann ist das kein Syntaxfehler - bei einem grammatikalischen Fehler, wird nichts ausgeführt, aber, wenn Anweisungen ab einem Punkt nicht mehr ausgeführt werden, dann heißt das eigentlich, da wird eine Funktion oder Varia-

ble syntaktisch korrekt aufgerufen, die es nicht gibt. Und ich suche und suche, nach einem echten Fehler - am Ende merke ich, beim Suchen, dieses Fehlers, hoppla, da wird eine Anweisung nicht mehr ausgeführt - das deutet auf so einen Fehler hin - da wurde eine Funktion oder Methode aufgerufen, die es nicht gibt. Falls sie sich gewundert haben, was ich da suche - was an einem Compiler logisch falsch sein - eigentlich nichts - es war auch nicht falsch - ich habe eine falsche Methode aufgerufen: Das ist der korrekte Syntaxbaum:



Jetzt mit Syntaxbaum:

`https://www.ituenix.de/reverscompiler6.html`

Ich habe den verfickten Fehler endlich gefunden - ich suche die ganze Zeit im Baum - und mache und mache - und dann stelle ich fest - da stand - `document.window.write ()`, `statt window.document.write ()`, es war also wegen einem Ausgabefehler - und ich suche und suche.