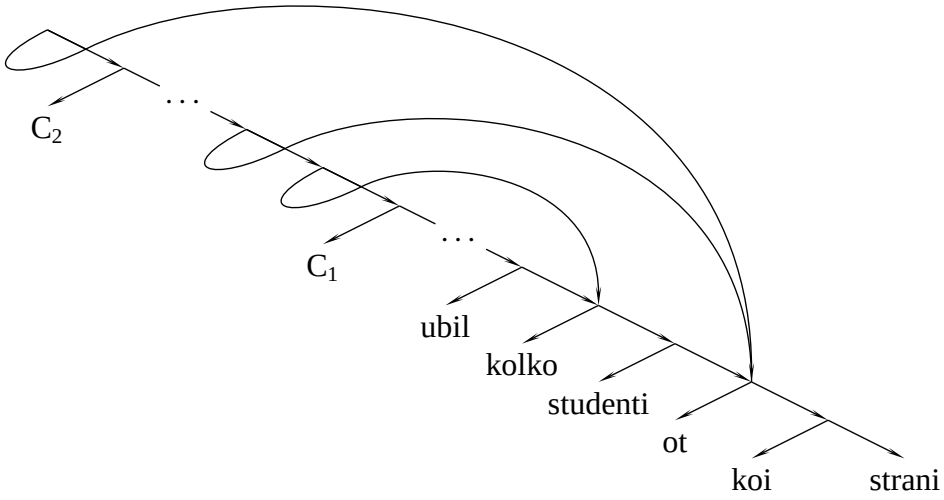
jTree
für Linguisten

TEX–Makros für die Setzung komplexer Bäume



Benutzerhandbuch

John Frampton
j.frampton@neu.edu

17. September 2006
Version 2.3

Ins Deutsche übertragen von:
Benjamin Grimm,
Chris Heckrodt,
Chris Langenberg,
Sören Schaller
und
Andrea Schlegel

Inhalt

1. Einführung

Komplexe Baumschemata, die Linguisten häufig zur Darstellung nutzen, tun sich mit einer linaren Darstellungsweise schwer, die es Menschen erleichtern würde, die beabsichtigte hierarchische Struktur rasch zu erfassen. Das erschwert es, einen TEX-Code so zu schreiben, dass er später noch leicht zu modifizieren oder zu berichtigen ist oder dass ein Teil des Codes für einen anderen Text kopiert werden kann. Diese Probleme sollen durch jTree überwunden werden.

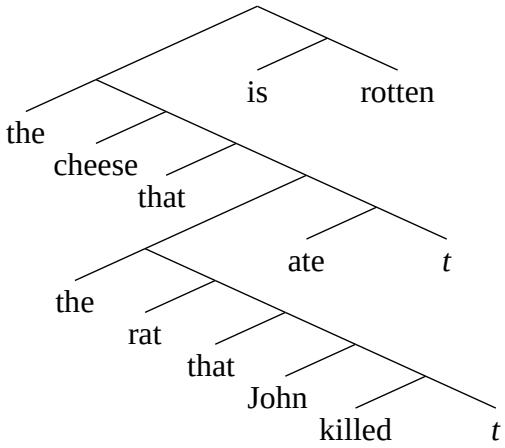
Es ist immer leichter, ein gutes Werkzeug für eine Aufgabe zu entwerfen, wenn die Aufgabe eng gefasst ist. Mein Ziel war es also, einen Baumformatierer zu schreiben, der sich besonders gut für die Art Baumschemata eignet, mit denen ich mich am meisten beschäftige. In der Syntax, die mich am meisten interessiert, sind Bäume häufig recht tief (wie etwa Beispiel 9 in Abschnitt 14 mit 19 Stufen). Runde oder eckige Klammer stellen keine brauchbare Alternative dar, wenn der TEX-Code die gewünschten Charakteristika aufweisen soll. Anstelle tief zu sein, sind die Bäume, die ich normalerweise setzen will, auf eine andere Art einfach: Sie sind in der Regel binär verzweigend und neigen dazu, auf der rechten Seite komplex zu verzweigen. jTree ist dazu gemacht, solche Bäume besonders gut zu setzen. Das stellt offensichtlich einen speziellen Blickpunkt innerhalb der Syntax-Theorie dar.

jTree basiert auf dem weit verbreiteten PSTricks-Paket. Viele der erweiterten Funktionen von jTree erfordern die Gewilltheit, ein gewisses Maß PSTricks zu erlernen. Linguisten wird bei PSTricks viel Nützliches begegnen, auch außerhalb des Kontextes der Darstellung und Kommentierung von Bäumen. Die meisten Befehle nutzen Parameter, die in der PSTricks-Dokumentation erklärt werden. Informationen über PSTricks erhalten Sie hier: *http://www.tug.org/applications/PSTricks/*. jTree benötigt auch das PST-XKey-Paket, welches zum Standard-Mechanismus für die Behandlung von Parametern in PSTricks-basierten Paketen gehört. Im Anhang A finden Sie Informationen zur Installation von PSTricks, PST-XKey und jTree-Paketen.

Manche Baumformatierer streben es an, Entscheidungen über Setzungsorte automatisch zu treffen, um den Formatierer einfach und idiotensicher zu machen. Die Kehrseite davon ist – da komplexe Bäume ohnehin ziemlich schwer zu setzen sind – dass man den zusätzlichen Aufwand aufbringen muss, die automatische Hilfe, die der Formatierer leistet, rückgängig zu machen. jTree verfolgt den Ansatz, sehr wenig automatisch zu setzen, stattdessen aber transparent zu machen, was getan wird und leicht einstellbar zu sein. Viele Funktionen des Bäumebauens in jTree werden durch Parameter kontrolliert, deren Standardeinstellungen gewöhnlich genügen, die aber – wenn nötig – für eine Feinsteuerung erhältlich sind. Der Kern von jTree ist die JTDL, *jTree description language*, eine Sprache zur Baumbeschreibung. Wir wollen mit der Diskussion dieser Sprache beginnen.

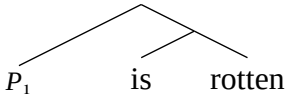
2. Die jTree-Bechreibungssprache (JTDL)

Es gibt zwei Komponenten: 1. einen einfachen Weg, Bäume mit Rechtsverzweigung zu beschreiben (die ich für Bäume verwende, bei denen der linke Ast unverzweigt bleibt); und 2. einen Mechanismus, einen angrenzenden Baum in einen anderen einzubinden. Betrachten Sie als Beispiel den folgenden Baum:

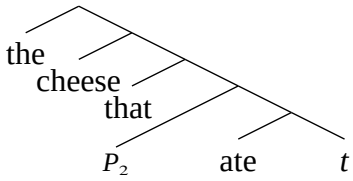


Dieser Baum kann als eine Sequenz von drei Verzweigungen nach rechts und der Anweisung, wie sie kombiniert werden können, beschrieben werden.

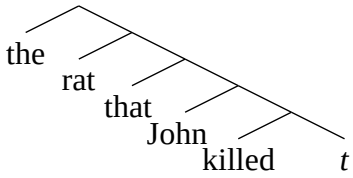
(1) 1.



2. adjoin at P_1:



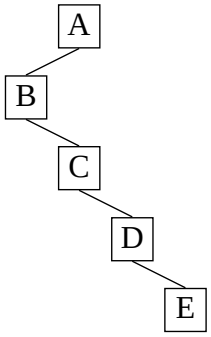
3. adjoin at P_2:

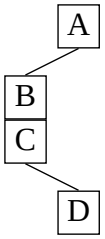


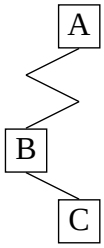
Zunächst klären wir die Frage, wie die nach rechts verzweigenden Bäume beschrieben werden, dann, wie man sie kombiniert.

2.1. Die Beschreibung rechts-verzweigender Bäume

Einige Bäume sind im wesentlichen eine lineare Sequenz von Zweigen und Knoten.

(2) a. 

b. 

c. 

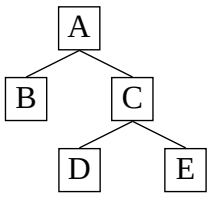
Beschrieben wird ein solcher Baum als eine Sequenz von Zweigen und Labels. Sie werden einfach auf folgende Weise dem Vorgänger hinzugefügt:

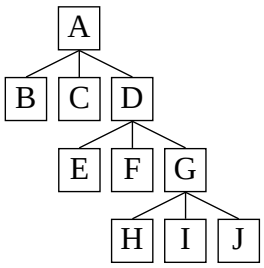
a. [A] ⟨left⟩ [B] ⟨right⟩ [C] ⟨right⟩ [D] ⟨right⟩ [E] .

b. [A] ⟨left⟩ [B] [C] ⟨right⟩ [D] .

c. [A] ⟨left⟩ ⟨right⟩ ⟨left⟩ [B] ⟨right⟩ [C] .

Rechtsverzweigungen können einfach beschrieben werden, indem dieser Sprache ein Operator ^ hinzugefügt wird, der die Anfügung einfach an den Beginn der vorangegangenen Verzweigung hängt. Die Bäume (3) wurden mit der Sequenz (4) beschrieben.

(3) a. 

b. 

(4) a. [A]

⟨left⟩ [B] ^ ⟨right⟩ [C]

⟨left⟩ [D] ^ ⟨right⟩ [E] .

b. [A]

⟨left⟩ [B] ^ ⟨vert⟩ [C] ^ ⟨right⟩ [D]

⟨left⟩ [E] ^ ⟨vert⟩ [F] ^ ⟨right⟩ [G]

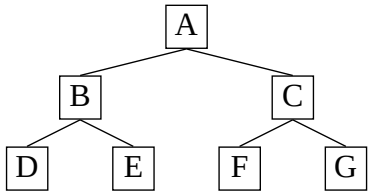
⟨left⟩ [H] ^ ⟨vert⟩ [I] ^ ⟨right⟩ [J] .

Für solche Beschreibungen benötigt der Parser lediglich zwei Positionen. Am Beginn eines Baumes sind beide Positionen, $\mathcal{P}$ und $\mathcal{Q}$, als Standard gesetzt. Wurde ein Label oder ein Zweig geparst, wird $\mathcal{P}$ automatisch zum aktuellen Punkt. Wenn ein Zweig geparst wurde, wird $\mathcal{Q}$ automatisch zum Startpunkt. Wird ^ geparst, wird $\mathcal{P}$ auf $\mathcal{Q}$ gesetzt.

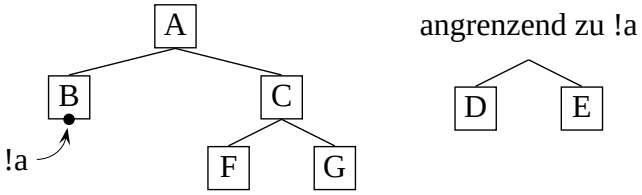
2.2. Knoten von Bäumen

Diese Sprache kann durch das Hinzufügen eines Adjunktionspunktes und der Syntax eines angrenzenden Baumes in einen anderen am spezifizierten Adjunktionspunkt erweitert werden. Die Zerlegung (5b) des Baumes (5a) wird mit (5c) beschrieben.

(5) a.



b.



c. A

⟨wideleft⟩ B !a ^ ⟨wideright⟩ C

⟨left⟩ F ^ ⟨right⟩ G .

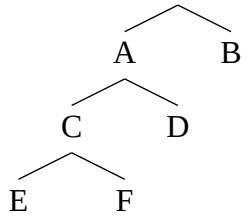
!a = ⟨left⟩ D ^ ⟨right⟩ E .

Trifft der Parser auf einen Knoten, wird er als $\mathcal{P}$ registriert, so dass künftig wieder auf ihn zugegriffen werden kann. In jTree werden die Namen der Knoten mit dem Zeichen ! definiert. Die Zeichenfolge beginnt mit !, gefolgt von einem Leerzeichen. Der zu beschreibende Baum ist ein physikalischer und kein abstrakter Baum. Die Zweige sind Objekte, die ein Abmaß haben und mit einem Namen identifiziert werden. Der Zweig, der am Beispiel oben mit <wideleft> gekennzeichnet wurde, hat andere Maße als der mit <left> markierte Zweig. jTree beruht auf dieser Sprache zur Beschreibung von Bäumen. Es wird die Möglichkeit unterstützt, Zweige auf verschiedene Art zu definieren, zu benennen und Bäume im spezifizierten JTDL-Format zu zeichnen. Wie eine Baumgrafik generiert wird, ist nicht nur abhängig von ihrer Beschreibung, sondern auch von einer Vielzahl an Einstellungen (Parameter, Schriftart etc.). Der TEX-Code zur Generierung von Baumgrafiken hat folgende Form:

```
\jtree
Präliminardefinitionen. Parametereinstellungen
\! = einfache Baumbeschreibung.
Definitionen, Parametereinstellungen, Grafik ohne Maßangabe
\!a = einfache Baumbeschreibung.
Definitionen, Parametereinstellungen, Grafik ohne Maßangabe
\!b = einfache Baumbeschreibung.
Grafik ohne Maßangabe
\endjtree
```

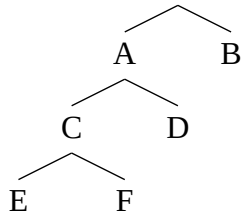
Wird `\jtree` ausgeführt, wird ein Knoten mit dem Namen `!` etabliert. Das Makro `\!` ist so definiert, dass es bei `\!xxx` = $\mathcal{P}$ und $\mathcal{Q}$ genau an diesen mit `!xxx` genannten Knoten setzt, dann mit der Verzweigung der darauffolgenden Baumbeschreibung beginnt und abschließt, wenn es auf einen Punkt trifft. Zum Beispiel:

```
\jtree
\! = <left>{A}!a ^<right>{B}.
\!a = <left>{C}!b ^<right>{D}.
\!b = <left>{E} ^<right>{F}.
\endjtree
```



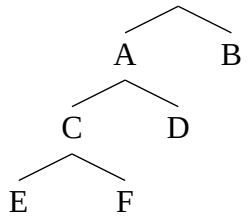
Der folgende Code generiert ein identisches Ergebnis.

```
\jtree
\! = <left>{A}!a ^<right>{B}.
\!a = <left>{C}!a ^<right>{D}.
\!a = <left>{E} ^<right>{F}.
\endjtree
```



Sobald der erste Unterbaum begonnen wurde, ist der im Hauptbaum benannte Name `!a!` nicht mehr vonnöten (in diesem Beispiel) und kann weiterhin zugeordnet werden. Das ist genauso möglich:

```
\jtree
\! = <left>{A}! ^<right>{B}.
\! = <left>{C}! ^<right>{D}.
\! = <left>{E} ^<right>{F}.
\endjtree
```

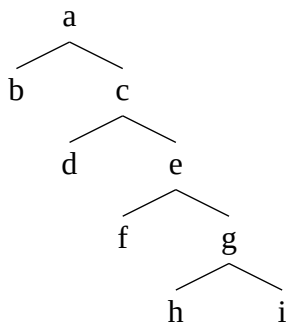


Es ist nichts Besonderes an dem von `\jtree` festlegten Namen `!`, es sei denn, er wurde vor der beginnenden Konstruktion bestimmt.

2.3. Konstruktionen mit Doppelpunkt

Die Kombination <left>*Label*^<right> kommt recht häufig bei binären Baumbeschreibungen vor. JTDL hält dafür ein Kürzel bereit. :*Label* wird als <left>*Label*^<right> interpretiert. Im Gebrauch sieht das so aus:

```
\jtree
\! = {a}
   :{b} {c}
   :{d} {e}
   :{f} {g}
   :{h} {i}.
\endjtree
```

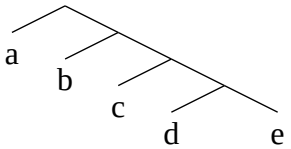


Diese Formatierung dient der besseren Lesbarkeit. Der folgende Code generiert den gleichen Baum:

```
\jtree\! ={a}:{b}{c}:{d}{e}:{f}{g}:{h}{i}.\endjtree
```

Ein weiteres Beispiel:

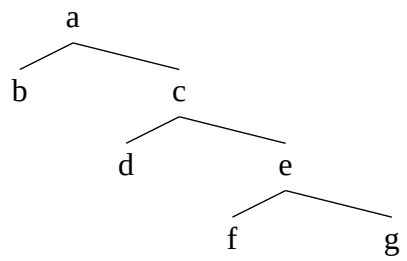
```
\jtree
\! = :{a} :{b} :{c} :{d} {e}.
\endjtree
```



Aus Gründen der Anschaulichkeit sind obige Beschreibung stark vereinfacht. Hier steht :*Label* für <DoppelpunktA>*Label*^<DoppelpunktB>. *pst-jtree* definiert die Zweige <DoppelpunktA> und <left> als identisch, und ebenso die Zweige

<DoppelpunktB> und <right>. Die Zweige können beliebig umdefiniert werden. Zum Beispiel:

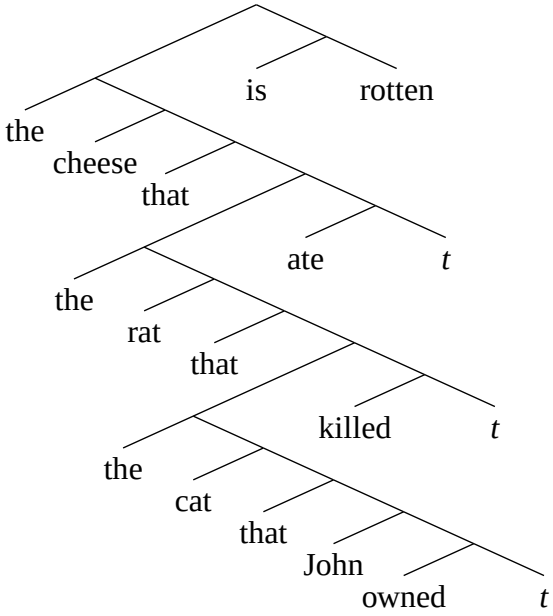
```
\jtree
\defbranch<colonB>(1)(-.5)
\! = {a}
   :{b} {c}
   :{d} {e}
   :{f} {g}.
\endjtree
```



Ein späterer Abschnitt wird detaillierter beschreiben, wie Zweige definiert werden können. Vorerst bleibt zu sagen, dass ein Zweig durch seine Höhe und seinen Neigungsgrad bestimmt wird. Labels werden vom jTree-Parser als {*stuff*} erkannt, dabei kann "stuff" alles sein, was in eine TEX\hbox hineingeht. Wie ein automatischer Raum über und unter Labels eingefügt wird, wie er festgelegt und eingestellt werden kann, heben wir uns für spätere Abschnitte auf.

Das folgende Beispiel verwendet den Doppelpunkt-Kurzbefehl reichlich.

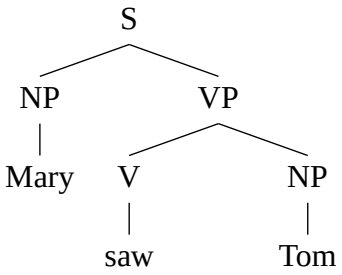
(6)
```
\jtree
\defbranch<Left>(2.3)(1)
\! = <Left>!a ^<right> :{is} {rotten}.
\!a = :{the} :{cheese} :{that}
   <Left>!b ^<right> :{ate} {\it t}.
\!b = :{the} :{rat} :{that}
   <Left>!c ^<right> :{killed} {\it t}.
\!c = :{the} :{cat} :{that} :{John} :{owned} {\it t}.
\endjtree
```



2.4. Einreihige Adjunktion

Zusätzlich zum Operator `:` bietet jTree noch einen anderen Kurzbefehl. Er liefert eine Alternative für Verknüpfungen wie im folgenden Code.

```
\jtree[xunit=2.8em,yunit=1em]
\! = {S}
   :{NP}!a {VP}
   :{V}!b {NP}
   <vert>{Tom}.
\!a = <vert>{Mary}.
\!b = <vert>{saw}.
\endjtree
```



Dies kann auch so geschrieben werden:

```
\jtree[xunit=2.8em,yunit=1em]
\! = {S}
   :{NP}(<vert>{Mary})  {VP}
   :{V}(<vert>{saw})  {NP}<vert>{Tom}.
\endjtree
```

Die Interpretation von `(...)` führt die *einreihige Adjunktion* aus, ohne explizit den Adjunktionspunkt zu benennen. In der Praxis sollte die Beschreibung von solchen Adjunktionen vermieden werden, denn sehr einfache Nonterminale (wie in diesem Beispiel) können sehr schnell zu unverständlich langen Befehlen führen. Kurzum:

JTDL ist dafür gemacht, der Entstehung von Hornissennestern aus Klammern vorzubeugen.

3. Parameter

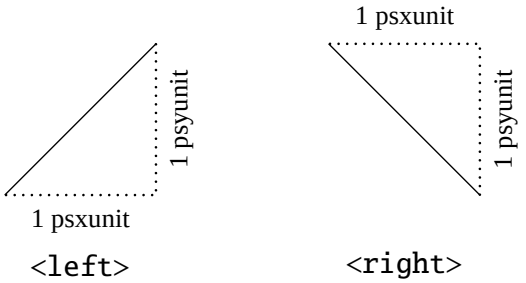
Viele Elemente von jTree beachten die verschiedenen Aspekte der Schriftsetzung. *pst-jtree* beinhaltet die Definitionen:

```
 \defbranch<left>(1)(1)
\defbranch<right>(1)(-1)
```

(Zitierungen von *pst-jtree* werden wie oben in einer Box dargestellt.)

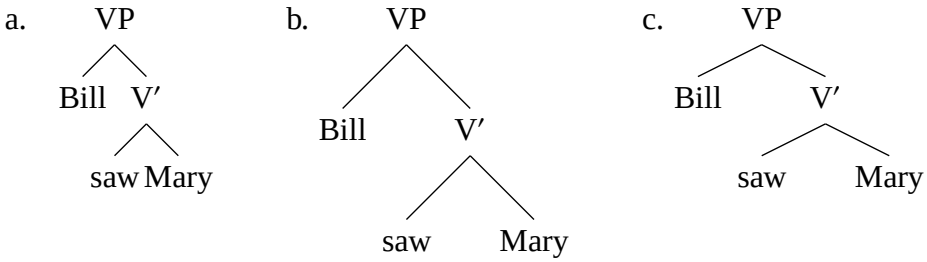
Damit wird <left> als Zweig mit der Höhe 1 psyunit und dem Neigungsgrad 1 definiert; <right> als ein Zweig mit der Höhe 1 psyunit und dem Neigungsgrad −1. Genauer wird die Spezifikation von Zweigen in Abschnitt 4 diskutiert, es reicht zu diesem Zeitpunkt aus zu wissen, dass <left> und <right> –wie unten dargestellt– gezeichnet werden:

(7)



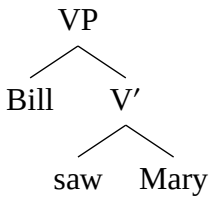
Die Maße der Zweige werden in psunits angegeben, wobei die x-Maße von den y-Maßen unabhängig sind. Dieser Fakt geht mit der Tatsache einher, dass die Zweige in physikalischen TEX-Maßen gerendert werden. Das bietet eine großartige Flexibilität. Die folgenden drei Bäume wurden genau mit demselben jTree-Code generiert, aber mit \psset{unit=1em}, psset{unit=2em} und psset{xunit=2em,yunit=1em}.

(8)



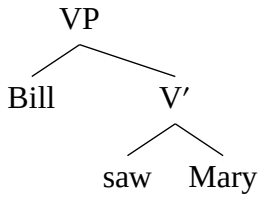
\jtree akzeptiert Parameter direkt, man kann also sagen:

```
\jtree[xunit=1.5em,yunit=1em]
\! = {VP}
   <left>{Bill} ^<right>{V$'$}
   <left>{saw} ^<right>{Mary}.
\endjtree
```



Es können auch einzelne Zweige Parameter besitzen:

```
\jtree[xunit=1.5em,yunit=1em]
\! = {VP}
   <left>{Bill} ^<right>[xunit=3em]{V$'$}
   <left>{saw} ^<right>{Mary}.
\endjtree
```



Die Parameter unit, xunit und yunit sind durch PSTricks definiert. jTree bestimmt einige eigene Parameter. Einer ist scaleby.
Mit \psset{scaleby=*x y*} werden Zweige gezeichnet als ob

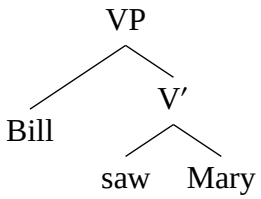
\psset{xunit=*x*\psxunit,yunit=*y*\psyunit}

ausgeführt worden wäre (die ps-Maße wurden jedoch nicht verändert).

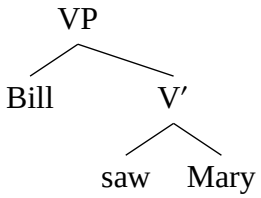
\psset{scaleby=*x*} ist gleichbedeutend mit \psset{scaleby=*x x*}.

Folgendes ist also möglich:

```
\jtree[xunit=1.5em,yunit=1em]
\! = {VP}
   :[scaleby=2]{Bill} {V$'$}
   :{saw} {Mary}.
\endjtree
```




```
\jtree[xunit=1.5em,yunit=1em]
\! = {VP}
   <left>{Bill} ^<right>[scaleby=2 1]{V$'$}
   :{saw} {Mary}.
\endjtree
```

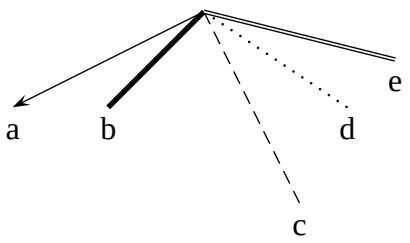


Insgesamt definiert jTree über ein Dutzend Parameter. Gesetzt werden sie mit \psset, aber, was viel wichtiger ist, sie können in besonderen Fällen \jtree

hinzugefügt werden und auch zu den Zweigen und Knoten inerhalb der \jtree-Umgebungen.

Oft ist es eher hilfreich, Zweige mit PSTricks-Parametern zu verändern.

```
\jtree[xunit=3em,yunit=3em]
\! = <left>[scaleby=2 1,
            arrows=->]{a}
   ^<left>[linewidth=2pt]{b}
   ^<right>[scaleby=1 2,
      linestyle=dashed]{c}
   ^<right>[scaleby=1.5 1,
      linestyle=dotted,linewidth=1pt]{d}
   ^<right>[scaleby=2 .5,doubleline=true]{e}.
\endjtree
```

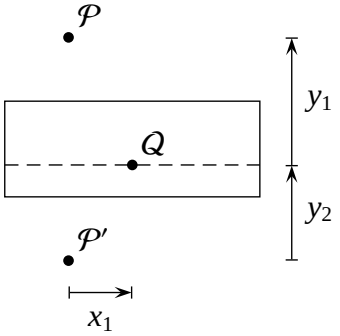


Die PSTricks-Dokumentation kann für viele andere Linien- und Pfeilparameter herangezogen werden, um Zweige verschiedenartig darzustellen.

4. Label

Bei Beschreibungen der Bäume wird der Schriftzug in der hbox `{...}` als *label box* bezeichnet. Nachdem die label box einer Struktur mit einem bestimmten Punkt $\mathcal{P}$ hinzugefügt wurde, wird ein neuer Punkt $\mathcal{P}'$ der gegenwärtige. Die Sachverhalte in diesem Abschnitt betreffen die Positionen von $\mathcal{P}$, der label box und $\mathcal{P}'$ im Verhältnis zueinander. Die Position der label box wird durch die Positionsbestimmung des Mittelpunktes ihrer Grundlinie, hier bezeichnet als $\mathcal{Q}$, angegeben. Die jeweilige Position hängt von der Höhe h und der Tiefe d der label box, sowie fünf weiteren Parametern ab: `labelgapt`, `labelgapb`, `labelstrutt`, `labelstrutb`, und `labeloffset`. Einsteiger sollten sich vor der scheinbaren Komplexität nicht abschrecken lassen. Verschiedene Voreinstellungen sind in *pst-jtree* bereits gesetzt und der Sachverhalt kann größtenteils ignoriert werden, bis ein bestimmter Effekt erwünscht wird. Wenn es soweit ist und ein komplexes Positionsproblem auftaucht, wird die Anpassungsfähigkeit klar verständlich und angenehm zu handhaben sein. Die Positionsregeln sind unten angegeben, mit |`parameter`| kann der Wert bestimmt werden.

(9)



$$x_1 = |\texttt{labeloffset}|$$

$$y_1 = \mathrm{Max}(h, |\texttt{labelstrutt}|) + |\texttt{labelgapt}|$$

$$y_2 = \mathrm{Max}(d, |\texttt{labelstrutb}|) + |\texttt{labelgapb}|$$

Die Folge dieser Regeln ist die, dass der Kopfteil der label box einen Abstand von mindestens |`labelgapt`| unterhalb des Terminus der folgenden Abzweigung oder des labels haben wird, und die Grundlinie der label box wird den Abstand |`labelstrutt`| + |`labelgapt`| unterhalb des Terminus haben, außer die label box ist ungewöhnlich hoch (z.B. $h >$ |`labelstrutt`|). Das bedeutet, dass label boxen niemals zu nah an die Abzweigungen oder Label, denen sie folgen, kommen und die Grundlinie der label boxen wird an unterschiedlichen Abzweigungen mit der gleichen Höhe angepasst werden, außer die label box ist außergewöhnlich hoch. Die gleichen Überlegungen betreffen auch die abhängigen Positonen der label box und des Punktes $\mathcal{P}'$.

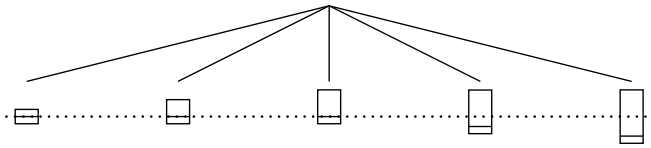
Es gibt noch den Parameter `normallabelstrut`, den man auf wahr oder falsch setzen kann: *true* oder *false*. Steht er auf wahr, so wird jedes Mal, wenn jTree die Maßangaben von label strut ausführen soll, es auf die Angaben der gegenwärtigen Tex strut gesetzt. Speziell wird

```
\psset{labelstrut={\the\ht\strutbox} {\the\dp\strutbox}}
```

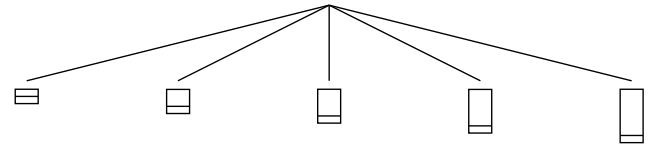
ausgeführt, wenn jTree aufgerufen wird. `\psset{labelstrut=x y}` ist äquivalent zu `\psset{labelstrutt=x, labelstrutb= y}`. Benutzer müssen sich nicht

mit den Einstellungen zu label strut befassen, bis sie spezielle Effekte erwünschen, da *pst-jtree* `normallabelstrut` auf *true* setzt. Die vorgegebenen Einstellungen für die oberen und die unteren Labellücke beträgt .35 ex. `\psset{labelgap=x}` ist äquivalent zu `\psset{labelgapt=x, labegagpb= x}`.

Der Effekt dieses Schemas ist unten dargestellt, mit `normallabelstrut` auf wahr und `labelgapt` sowie `labelgapb` auf .6 ex gesetzt. Die Größe des gezeigten labels wird durch die Darstellung der Grundlinie verdeutlicht. Wenn die Höhe des labels |`labelstrutt`| nicht überschreitet, dann werden die Grundlinien der labels einander angepasst, ansonsten ist die Spitze der label box entsprechend der Angabe in |`labelgapt`| unterhalb des Terminus' der Abzweigung, der es zugeordnet ist, entfernt.

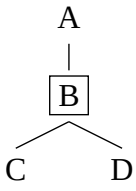


Manchmal ist es nützlich, `labelstrutt` auf 0 zu setzen. In diesem Fall sieht das Resultat so aus:

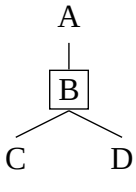


Viele Benutzer werden wahrscheinlich `normallabelstrut` auf *wahr* setzen und nicht mehr weiter dran denken. Aber die meisten Benutzer werden die label gaps von Zeit zu Zeit ändern wollen. Als Vergleich dienen die folgenden Beispiele. In einigen Anwendungen könnte das zweite Beispiel zu bevorzugen sein.

```
\jtree
\! = {A}
   <vert>{\psframebox{B}}
   :{C}{D}.
\endjtree
```



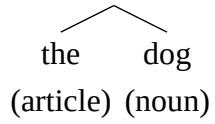

```
\jtree
\! = {A}
   <vert>{\psframebox{B}}[labelgap=0]
   :{C}{D}.
\endjtree
```



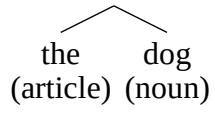
Das wird beim Setzen mit (15) in Abschnitt 11 verwendet.

Der folgende Vergleich ist ebenfalls interessant, da teilweise negative label gaps angegeben werden.

```
\jtree
\! = :({the}{(article)}) {dog}{(noun)}.
\endjtree
```



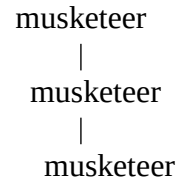

```
\jtree
\! = :({the}{(article)}[labelgapt=-3pt])
   {dog}{(noun)}[labelgapt=-3pt].
\endjtree
```



Betrachten Sie die Beispiele 3 und 4 in Abschnitt 14 für Anwendungen des folgenden Tricks, der den Gebrauch an mehrzeiligen Labeln oft überflüssig macht.

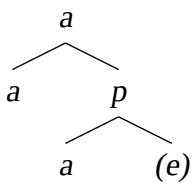
Die label box kann horizontal durch die Benutzung von `labeloffset` platziert werden.

```
\jtree
\! = {musketeer}
   <vert>{musketeer}[labeloffset=1ex]
   <vert>{musketeer}[labeloffset=2ex].
\endjtree
```



Betrachten Sie Beispiel 1 in Abschnitt 14 für eine Illustration dessen, wie man mit dem Ändern von `labeloffset` bestimmte Abstandsprobleme zwischen den labels löst.
Es gibt noch einen weiteren Parameter, der für Label von Bedeutung ist. Der Inhalt, festgelegt durch `everylabel`, wird in eine token-Liste aufgenommen, welche wiederum zu Beginn jedes Labels eingefügt wird.

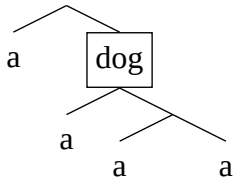
```
\jtree[everylabel=\sl]
\! = {a}
   :{a} {p}
   :{a} {(e)}.
\endjtree
```



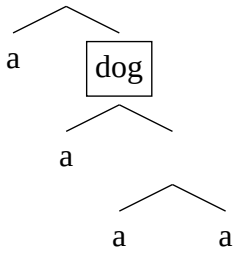
Wenn ein label mit `{\omit...}` oder `{\pnode...}` beginnt, ist der oben genannte vertikale Algorithmus umgangen und das label wird mit der Oberkante gemäß des Levels von $\mathcal{P}$ positioniert. $\mathcal{P}'$ wird dann direkt unter $\mathcal{P}$ gemäß des Levels

der Unterkante der label box platziert. `labeloffset` arbeitet auch weiterhin. Vergleichen Sie folgendes:

```
\jtree[everylabel=\strut,labelgap=3pt]
\! = :{a} {\omit\psframebox{dog\strut}}
   :{a} {\pnode{A1}}
   :{a} {a}.
\endjtree
```




```
\jtree[everylabel=\strut,labelgap=3pt]
\! = :{a} {\psframebox{dog\strut}}
   :{a} {}
   :{a} {a}.
\endjtree
```



---

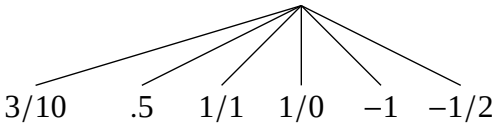
5. Zweige

Die meisten Baumschemata formieren zuerst die Anordnung der Label, bevor sie diese mit den Abzweigungen verbinden. Die Ausdehnung der Abzweigungen werden durch die Position der Label, die sie verbinden, bestimmt. Ein jTree-Zweig bestimmt seine Ausdehnung auf der anderen Seite durch eigene Angaben und legt den Platz des Inhalts, auf den es zeigt, selber fest. Zweige werden durch die Angabe der Höhe und der Neigung definiert. Die Syntax lautet:

`\defbranch<`*name*`>(`*Höhe*`)(`*Neigung*`)`

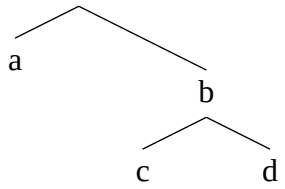
Die Höhe kann TEX-Ausmaße annehmen (pt, ex, em, in, cm, etc.), kann aber auch vollkommen numerisch sein. Ist letzteres der Fall, so wird die Höhe in psyunits abgemessen. Die Neigung berechnet sich aus dem Verhältnis der vertikalen Ausrichtung, angegeben in psyunits, zur horizontalen Ausrichtung, angegeben in psxunits. Linien, die von links unten nach rechts oben verlaufen, haben eine positive Neigung, verlaufen sie allerdings von links oben nach rechts unten, dann ist die Neigung negativ. Normalerweise kann sie durch eine Dezimalzahl ausgedrückt werden. Wenn die horizontale Ausdehnung Null beträgt, wodurch es einer vertikalen Linie gleichkommt, ist eine dezimale Neigung unmöglich (weil die Division Null ergibt) und sie muss durch das Verhältnis ausgedrückt werden.

```
\jtree[unit=2.5em]
\defbranch<1;3/10>(1)(3/10)
\defbranch<1;.5>(1)(.5)
\defbranch<1;1/1>(1)(1/1)
\defbranch<1;1/0>(1)(1/0)
\defbranch<1;-1>(1)(-1)
\defbranch<1;-1/2>(1)(-1/2)
\! = <1;3/10>{$3/10$}
   ^<1;.5>{$.5$}
   ^<1;1/1>{$1/1$}
   ^<1;1/0>{$1/0$}
   ^<1;-1>{$-1$}
   ^<1;-1/2>{$-1/2$}.
\endjtree
```



Beachten Sie, dass die Zeichen in den Zweigen nicht der Einschränkung eines alphabetischen Zeichens unterliegen. Ich habe nie einen Gebrauch dafür gefunden, Zweige über die Angabe der Höhe in Baueinheiten zu definieren, aber irgendjemand wird ihn vielleicht einmal finden.

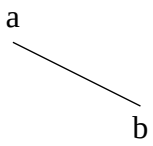
```
\jtree[xunit=2em,yunit=1em]
\defbranch<Right>(2em)(-1)
\! = <left>{a} ^<Right>{b}
   <left>{c} ^<right>{d}.
\endjtree
```



Letztlich sei noch zu erwähnen, dass die Neigungsspezifizierung eines Zweiges nicht einer auf Papier gezeichneten entsprechen wird, bis die xunits und die yunits angeglichen sind. Eine stellt das Verhältnis der xunits gegenüber den yunits dar, die andere zu den Baueinheiten. Der Zweig <right> ist mit der Neigung −1 spezifiziert, aber:

```
\jtree[xunit=.5em,
   yunit=2em]
\! = {a}<right>{b}.
\endjtree
```


```
\jtree[xunit=4em,
   yunit=2em]
\! = {a}<right>{b}.
\endjtree
```

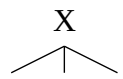


jTree ist auf binäre Baumstrukturen spezialisiert, aber *pst-jtree* legt einige Zweige vorher so fest, wodurch die Erstellung von Bäumen mit mehr Abzweigungen vereinfacht wird. Die gesamte Bestandsliste der vordefinierten Zweige (branches) sieht folgendermaßen aus:

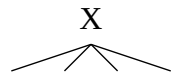
```
 \defbranch<left>(1)(1)
\defbranch<right>(1)(-1)

\defbranch<4wideleft>(1)(2/3)
\defbranch<4left>(1)(2)
\defbranch<4right>(1)(-2)
\defbranch<4wideright>(1)(-2/3)

\defbranch<wideleft>(1)(1/2)
\defbranch<wideright>(1)(-1/2)

\defbranch<bigleft>(2)(1)
\defbranch<bigright>(2)(-1)

\defbranch<vert>(1)(1/0)
\defbranch<shortvert>(.5)(1/0)

\defbranch<colonA>(1)(1)
\defbranch<colonB>(1)(-1)
```

Natürlich kann der Benutzer Zweige eigens, seinen Bedürfnissen entsprechend, definieren. <colonA> und <colonB> sind die vom colon-Makro verwendeten Zweige. Sie sind so bestimmt, dass sie <left> und <right> entsprechen, können aber den Vorlieben angepasst werden.

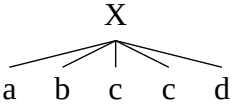
```
\jtree
\! = {X}
   <left>{a} ^<vert>{b} ^<right>{c}.
\endjtree
```




```
\jtree
\! = {X}
   <4wideleft>{a} ^<4left>{b}
      ^<4right>{c} ^<4wideright>{d}.
\endjtree
```




```
\jtree
\! = {X}
   <wideleft>{a} ^<left>{b} ^<vert>{c}
      ^<right>{c} ^<wideright>{d}.
\endjtree
```



---

6. Triangeln und varTriangeln

Triangeln können durch folgende Syntax definiert werden:

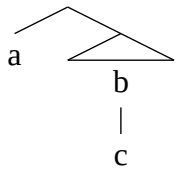
`\deftriangle<`*Name*`>(`*Höhe*`)(`*NeigungA*`)(`*NeigungB*`)`

Eine Triangel wird bereits vordefiniert.

```
\deftriangle<tri>(1)(1)(-1)
```

<tri> kann wie jeder andere Zweig gebraucht werden.

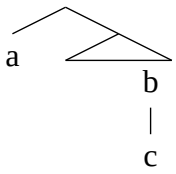
```
\jtree
\! = :{a} <tri>{b} <vert>{c}.
\endjtree
```



---

pst-jtree definiert den Parameter `triratio`, der dafür verwendet werden kann, die Position des neuen aktuellen Punktes entlang der Unterkante der Triangel festzulegen. Wenn *width* die Breite der Triangel ist, dann befindet sich der Punkt gemäß der Formel $x = triratio \times width$ rechts von der linken Ecke der Triangel. Die oberhalb dargestellte Struktur wird durch `triratio=.5` gesetzt, dem Vorgabewert, der auch den derzeitigen Punkt in die Mitte der Unterkante festlegt.

```
\jtree
\! = :{a} <tri>[triratio=.8]{b} <vert>{c}.
\endjtree
```

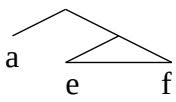


Wenn eine Triangel festgesetzt wird, wird die Weite berechnet und das Makro `\triwd` wird definiert, welches in Abhängigkeit zur Weite errechnet wird. *pst-jtree* beinhaltet:

```
\def\triline#1{\hbox to\triwd{#1}}
```

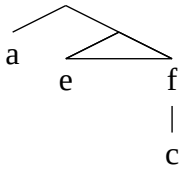
Die folgende Darstellung kann sich manchmal als nützlich erweisen:

```
\jtree
\! = :{a} <tri>{\triline{e\hfil f}}.
\endjtree
```



Bedenken Sie außerdem, dass Zweige auch überschrieben werden können. Etwas analoges zu dem folgenden ist manchmal brauchbar:

```
\jtree
\! = :{a} <tri> ^:{e} {f} <vert>{c}.
\endjtree
```



6.1. Die variable Triangel (varTriangel)

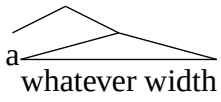
varTriangeln passen ihre Werte dem label entsprechend an, auf das sie zeigen. Eine varTriangel wird durch die Angabe der Höhe spezifiziert. Die Syntax lautet folgendermaßen:

`\defvartriangle<`*Name*`>(`*Höhe*`)`

jTree definiert eine varTriangel schon vorher.

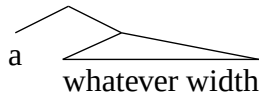
```
\defvartriangle<vartri>(1)
```

```
\jtree
\! = :{a} <vartri>{whatever width}.
\endjtree
```

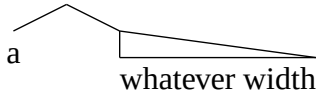


Der Parameter `triratio` hat für varTriangeln eine andere Bedeutung als für gewöhnliche Triangeln. Er ermittelt, wo sich die Mitte der Grundlinie einer Triangel hinsichtlich des obersten Scheitelpunktes befindet. Die Entfernung des Grundlinienmittelpunktes ist gemäß der Formel $triratio \times width$ entsprechend weit rechts vom linken Punkt der Grundlinie entfernt. Es ist einfacher die Formalien (wie unten) zu illustrieren, anstatt sie nur zu nennen.

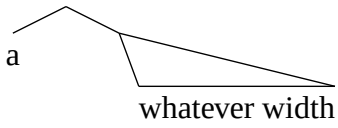
```
\jtree
\! = :{a} <vartri>
   [triratio=.3]{whatever width}.
\endjtree
```




```
\jtree
\! = :{a}
   <vartri>[triratio=0]{whatever width}.
\endjtree
```

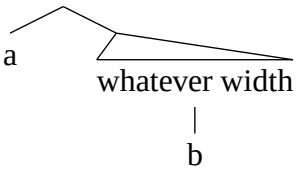



```
\jtree
\! = :{a}
   <vartri>[triratio=-.1,scaleby=2]
      {whatever width}.
\endjtree
```



Im Allgemeinen besteht keine Verzweigung des Labels, das der varTriangel folgt, also gibt es auch keine Erstellung eines Anschlusspunktes für das Label er befindet sich in der Mitte der Basis.

```
\jtree
\! = :{a} <vartri>
      [triratio=.1]{whatever width}
   <vert>{b}.
\endjtree
```

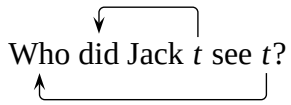


Ein Zweig kann einem anderen folgen, aber seit die Weite einer varTriangel von der Weite des folgenden Labels abhängt, muss eine varTriangel (mit optionalen Parametern) direkt von einem Label gefolgt werden. Ist dies nicht der Fall, wird ein Fehler angezeigt.

7. Knoten, genannt @tags

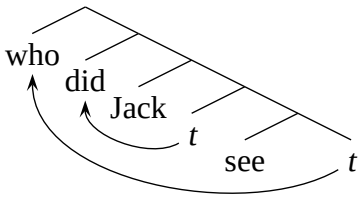
jTree beinhaltet direkte Unterstützung für PSTricks' leistungsfähiges *pst-node*-Packet. Selbst bei linearen Strukturen ist *pst-node* für Linguisten von Nutzen. Die Eigenschaft, Knoten zu definieren und sie mit Pfeilkurven verschiedenster Art zu verbinden, erlaubt es dem Benutzer, Sachen auf einfache Art und Weise zu programmieren, wie zum Beispiel:

```
\rnode{A1}{Who} \rnode{B1}{did}
   Jack \rnode{B2}{\sl t\/}
   see \rnode{A2}{\sl t\/}?}
\psset{linearc=2pt}
\ncbar[angle=-90]{->}{A2}{A1}
\ncbar[angle=90]{->}{B2}{B1}
```



Bei Baumstrukturen ist das pst-node-Makro `\nccurve` nur teilweise nützlich. **Achtung**: Knoten, die sich auf durch *pst-node*-Befehle definierte Namensstrukturen beziehen, sind keine Baumknoten.

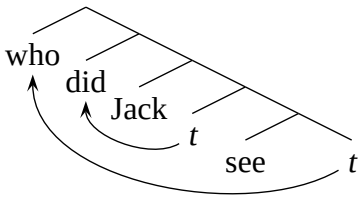
```
\jtree
\! = :{\rnode{A1}{who}}
   :{\rnode{B1}{did}} :{Jack}
   :{\rnode{B2}{\sl t}} :{see}
   {\rnode{A2}{\sl t}}.
\psset{arrows=->,angleA=-150,
   angleB=-90}
\nccurve{A2}{A1}
\nccurve{B2}{B1}
\endjtree
```



---

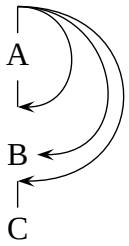
jTree erleichtert den Gebrauch von *pst-node* durch Behelfsmittel, @tags genannt, wie unten dargestellt ist:

```
\jtree
\! = :{who}@A1 :{did}@B1 :{Jack}
   :{\sl t}@B2 :{see} {\sl t}@A2 .
\psset{arrows=->,angleA=-150,
   angleB=-90}
\nccurve{A2}{A1}
\nccurve{B2}{B1}
\endjtree
```

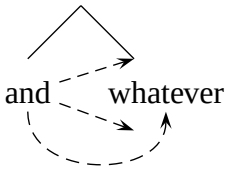


Ein jTree-@tag besteht es aus einem `@`, gefolgt von einer beliebigen Reihenfolge von Buchstaben, die eine gültige PSTricks-node-Bezeichung ergeben, sowie einem Leerraum. Wenn @tag einem label folgt (Parameter sind Teil eines labels), dann werden drei Knoten geschaffen. Wenn @tag zum Beispiel `@P2` ist, entstehen Knotenpunkte mit den Bezeichnungen `P2:t`, `P2` und `P2:b`.
`P2:t` ist der aktuelle Punkt, bevor sich später das Label an die Struktur angliedert und `P2:b` wird zum neuen Punkt, nachdem das Label Teil der Struktur geworden ist. `P2` ist ein Knotenbehältnis, welche die label box beinhaltet – mit dem Beziehungspunkt in der Mitte. Hinsichtlich `P2` ist `{`*stuff*`}@P2` äquivalent zu `{\rnode{P2}{`*stuff*`}}`. Wenn @tag keinem Label folgt, wird ein einzelner Knotenpunkt an der aktuellen Position $\mathcal{P}$ erstellt.
Diese Ideen sind im Folgenden illustriert:

```
\jtree
\! = @AA
   <vert>{A}
   <vert>{B}[labelgapt=12pt]@P2
   <vert>{C}.
\psset{arrows=->,angleA=0,angleB=0,
   ncurv=1.4}
\nccurve[nodesep=0]{AA}{P2:t}
\nccurve[nodesepA=0]{AA}{P2}
\nccurve[nodesep=0,ncurv=1.6]{AA}{P2:b}
\endjtree
```



```
\jtree[scaleby=1 2,labelgap=1.2ex]
\! = :{and}@A
   {whatever}[labeloffset=1em]@AA .
\psset{linestyle=dashed,arrows=->}
\ncline{A}{AA:t}
\ncline{A}{AA:b}
\nccurve[angleA=-90,angleB=-90,ncurv=1]{A}{AA}
\endjtree
```



In Kapitel 13 gibt es weitere Informationen über Knoten und die Nutzung von `\nccurve`.

8. Die Konstruktion mit Doppelpunkt detaillierter

Einige Einzelheiten der Syntax des Doppelpunkt-Aufbaus wurden bis zum jetzigen Zeitpunkt noch recht stiefmütterlich behandelt. Die Idee eines Doppelpunkt-Aufbaus ist im Grunde einfach: der Doppelpunkt wird durch den Zweig <colonA> ersetzt und nachdem ein paar Dinge verarbeitet wurden, wird ^<colonB> eingefügt. Jedoch muß präzisiert werden, an welcher Stelle das Einfügen von ^<colonB> stattfindet. Zum Beispiel: Wo wird in der folgenden Darstellung ^<colonB> eingefügt?

```
:{a} [labelgap=0] @AA !a @BB {c}
```

Die Regel ist diese: Das untenstehende Template wird so weit wie möglich ausgefüllt. Anschließend wird : durch <colonA> ersetzt und ^<colonB> nach dem Zielbereich (target) eingefügt.

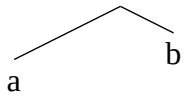
(10) : [*pars*] {*stuff*} [*pars*] *@tag* $\begin{pmatrix} (\dots) \\ !tag \end{pmatrix}$

target

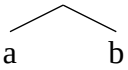
Es gibt zwei Beschränkungen. Erstens kann der Zielbereich nicht vollkommen leer sein. :<left>... oder ::..., z.B. ist nicht möglich. Zweitens sind Parameter immer Parameter von etwas; daher kann der [*pars*]-Terminus im Zielbereich nur bestehen, wenn ein {*stuff*}-Terminus vorhanden ist.

Schauen Sie sich die folgenden Beispiele an:

(11) a.
```
\jtree
\! = :[scaleby=2]{a} {b}.
\endjtree
```

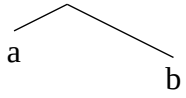


b.
```
\jtree
\! = :{a} [scaleby=2]{b}.
\endjtree
```



Man könnte meinen, daß der rechte Zweig in (11b) skaliert wurde. Aufgrund von [scaleby=2], welches in den Zielbereich geht, ist er es aber nicht. Wie in (10) zu sehen, ist er so zergliedert, daß der Labelparameter und <colonB> nach dem Zielbereich eingefügt werden. Als Labelparameter hat es keine Auswirkung. Die folgende Redewendung in Kapitel 14 wird fortwährend benutzt, um sicherzustellen, daß die Parameter nicht mit den Labelparametern verwechselt werden.

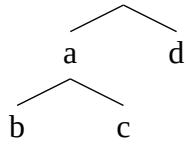
(12)
```
\jtree
\! = :{a}() [scaleby=2]{b}.
\endjtree
```



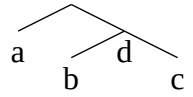
Das Template (10) ist hier ausgefüllt mit dem Zielbereich {a}(), so daß <colonB> vor [scaleby=2] eingefügt wird.

Der folgende Kontrast ist interessant. In (13a) ist der Zielbereich `{a}(:{b}{c})`, sodaß die innerlineare Adjunktion am linken Zweig auftritt. In (13b) allerdings, besetzt `!a` den Adjunktions-Slot im Template, so daß `{a}!a` der Zielbereich ist und die Abzweigung am rechten Zweig erfolgt.

(13) a.
```
\jtree
\! = :{a} (:{b}{c}) {d}.
\endjtree
```



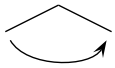
b.
```
\jtree
\! = :{a}!a (:{b}{c}){d}.
\endjtree
```



Zu bemerken ist, daß in (13b) die innerlineare Adjunktion den aktuellen Punkt $\mathcal{P}$ unverändert läßt. Dieser wird vom folgenden Label überschrieben.

Ein Zielbereich kann vollständig aus einem @tag bestehen.

(14)
```
\jtree
\! = :@A @B .
\nccurve[angleA=-60,
   angleB=240]{->}{A}{B}
\endjtree
```



Der Vollständigkeit halber, folgt ein kompletter Bericht der Grammatik zur Beschreibung von Bäumen.

8.1. Die Syntax der Sprache zur Beschreibung von Bäumen

Die Beschreibung der Bäume wird im Folgenden spezifiziert durch eine kontextfreie Grammatik und einige Vermutungen wie sie funktioniert. Zunächst zur Grammatik. Der Terminus *string*, der in einigen Erläuterungen auftaucht, ist ein Strang (engl. string) von Zeichen, der in eine TeX-Kontrollsequenz eingespeist wird. Die verschiedenen Zustände an den strings, die sich in unterschiedlichen Kontexten bemerkbar machen können, sind wie folgt.

tree_description → *simple_tree_description* (*tree_description*)

simple_tree_description → \! *string* = (*tree_body*) .

tree_body → *tree_item* (*tree_body*)

tree_item → *sprout, target, vartri_group, colon_group, operator*

sprout → < *string* > ([*pars*])

$$\textit{target} \rightarrow \begin{pmatrix} \textit{label} \\ \textit{@tag} \end{pmatrix} \begin{pmatrix} \textit{!tag} \\ \textit{inline\_adjunction} \end{pmatrix}$$

label → { *stuff* } ([*pars*]) (*@tag*)

inline_adjunction → (*tree_body*)

vartri_group → <*string*> (*parameters*) *label*
colon_group → : ([*pars*]) *target* ([*pars*])
@tag → @*string*
!tag → !*string*
operator → ^
pars → valid PSTricks parameter settings, ∅

Die umrahmten Zeichen oben sind Symbole der Sprache zur Beschreibung von Bäumen, nicht der Sprache zur Beschreibung der Grammatik. “Stuff” ist alles, was in eine TEX\hbox hineingeht. Ein leeres Kästchen ist ebenso möglich (eine Erweiterung, die das PSTricks-Parametersystem nicht bietet).
Note that an empty parameter specification `[]` is permitted, and is actually useful at times. This is an extension of the PSTricks parameter system, which disallows empty parameters.

Der string, der `\!` folgt, muss so sein, dass *!string* der Name eines Adjunktionspunkt ist, der schon definiert wurde; entweder durch `\jtree` oder durch ein !tag in einem vorhergegangenem geparsten Zielbereich. Der <*string*>, der in einem Spross auftaucht, muss den Namen eines Zweiges oder Dreiecks tragen, und der <*string*>, der in einer vartri group (Gruppe von variablen Triangeln) auftaucht, muss den Namen einer varTriangel tragen. Ein Name mit einem leeren string oder einem string, welcher Leerstellen enthält, ist möglich, aber nicht ratsam. Der string, der in einem @tag auftaucht, muss einen gültigen PSTricks-node-Namen tragen, und der zusätzlichen Restriktion, dass er keine Leerstellen aufweisen darf. Der string, der in einem !tag auftaucht (man achte auf das `!`, statt @), wird zum Namen des Verbindungspunktes. Auch er darf keine Leerstellen enthalten! Es ist zu bemerken, dass ein Leerzeichen nach !tags und @tags folgen muss. Davon abgesehen, ist jTree tolerant, was Leerzeichen oder deren Abwesenheit zwischen Einheiten (items) betrifft.

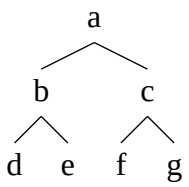
Parsing ist insofern einzigartig, da Zielbereiche und Label immer in Links-zu-Rechts-Aufteilung maximiert werden und nie leer sind. Da Labels in jedem Fall `{` und `}` enthalten, sind sie nie leer.

9. Ausdehnung und Auswertung der Kontrollsequenzen im Parsen

9.1. Der "-Ausweg vom Parsen von Bäumen

Wenn der Parser auf " trifft, wertet er den nächsten Token oder die nächste Gruppe aus, und verfährt weiter mit seiner Parserei. Eine Auswertung sollte kein Material beisteuern, da sie nicht geparsed wird. Aber eine Auswertung kann gewisse Parametereinstellungen verändern, die wiederum einen Einfluß darauf haben können, wie das restliche Material gesetzt ist. Zum Beispiel:

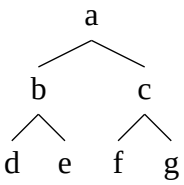
```
\jtree
\! = {a} :{b}!! {c}
   "{\psset{scaleby=.5 1}} :{f} {g}.
\!! = :{d} {e}.
\endjtree
```



`\! ... .` etabliert keine eingeschlossene Gruppe. Die Veränderung in der Skalierung besteht somit weiterhin auch für die darauffolgenden Unterbäume.

Den selben Effekt erhält man, wenn man die " wegläßt, falls das Reskalieren außerhalb des Parsens des Baumes stattfindet.

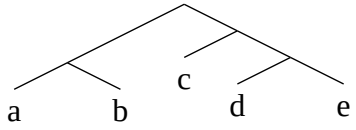
```
\jtree
\! = {a} :{b}!a {c}!b .
\psset{scaleby=.5 1}
\!a = :{d} {e}.
\!b = :{f} {g}.
\endjtree
```



9.2. Kontrollsequenzausdehnung

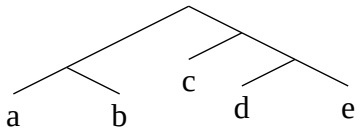
Wenn der Parser auf eine Kontrollsequenz oder einem aktiven Schriftzeichen in einer Baumbeschreibung trifft, wird er durch das Ausgewertete ersetzt bevor er weiterfährt zu parsen. Die Ausdehnung wird geparsed ohne weitere Ausdehnung ihres eigentlichen Token, welches einen unendlichen loop verhindert, falls die Ausdehnung eine unausgebbare Kontrollsequenz erbringt.

```
\jtree
\def\Colon{:[scaleby=2.3]}%
\! = \Colon !a :{c} :{d} {e}.
\!a = :{a} {b}.
\endjtree
```



pst-jtree enthält die Definition `\def\jtlong{[scaleby=2.3]}`, also könnten wir auch schreiben:

```
\jtree
\! = :\jtlong !a :{c} :{d} {e}.
\!a = :{a} {b}.
\endjtree
```



Diese Technik macht solchen Code deutlich transparenter. (6) auf Seite 7, zum Beispiel, würde dann so geschrieben werden:

```
\jtree[xunit=2.2em,yunit=1em]
\! =  :\jtlong !a :{is} {rotten}.
\!a = :{the} :{cheese} :{that} :\jtlong !b :{ate} {\it t}.
\!b = :{the} :{rat} :{that} :\jtlong !c :{killed} {\it t}.
\!c = :{the} :{cat} :{that} :{John} :{owned} {\it t}.
\endjtree
```

Die vollständige Liste von Parameterveränderungen, die in *pst-jtree* in Makros enkodiert sind, sind:

```
 \def\jtlong{[scaleby=2.3]}
\def\jtshort{[scaleby=.5]}
\def\jtwide{[scaleby=2 1]}
\def\jtbig{[scaleby=2]}
\def\jtjot{[scaleby=1.3]}
```

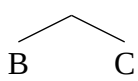
Freilich sollten Benutzer je nach ihren Anforderungen und Vorstellung die Makros abändern.

10. Feinjustierungen

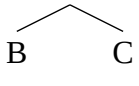
10.1. *baretopadjust*

Die Grundlinie einer von `\jtree...\endjtree` erstellen Box, ist für gewöhnlich die Grundlinie ihres Wurzellabels (root labels). Falls allerdings das root label leer ist, senkt es die Grundlinie zu tief ab, zumindest für meinen Geschmack. Ein Vergleich:

a.

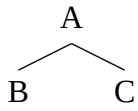


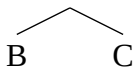
b.



jTree nimmt die richtige Entscheidung wahr und erhöht das Baumdiagramm, wenn das root label leer ist, durch die Anzahl, die mit dem Parameter `baretopadjust`

spezifiziert wird. Die Standardeinstellung ist 1.4ex, aber man kann es einstellen, wie man will. So sieht die Standardeinstellung aus:

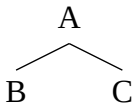
a. 

b. 

10.2. *treevshift*

jTree bietet zudem den Parameter `treevshift`, welcher standardmäßig auf 0 gestellt ist. Er ist unabhängig von `baretopadjust` und kann dazu benutzt werden, das Diagramm hinauf- und hinunterzubewegen, falls gewünscht.

```
$\to$\qquad
\jtree[treevshift=1.2em]
\! = {A} :{B} {C}.
\endjtree
```

→ 

10.3. *everytree*

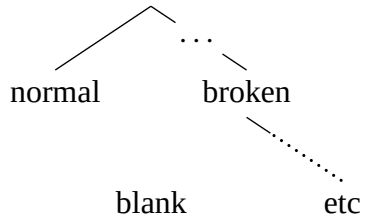
Wenn man sagt, z.B., `\psset{everytree=\psset{style=treestyle}}`, und man hat `treestyle` mithilfe einer PSTricks-Styledefinition definiert, dann werden diese Bäume in eben diesem Style erstellt. `\psset{everytree=`*x*`}` steckt *x* in die Token-Liste `\jteverytree`, welche zu Beginn jeder `\jtree`... `\endjtree`-Konstruktion eingefügt wird. Sie ist gruppiert, sodaß ihr Augenmerk eingeschränkt auf die Baumerstellung liegt. Eingefügt wird sie, bevor die `\jtree`-Parameter in Kraft treten, sodaß sie ggf. durch Parametereinstellungen überschrieben wird.

Wenn man will, kann man `\jteverytree` direkt bestimmen. Dasselbe gilt im übrigen auch für `everylabel`, welches die Tokenliste `\jteverylabel` verwendet.

10.4. Eigene Zweige

jTree zeichnet Zweige mithilfe des PSTricks-Makros `\psline`, aber es tut dies in einer indirekten Art und Weise. Eigentlich ruft sie das Makro `\branch@type` zum Zeichnen auf; und jTree enthält die Zeile `\let\branch@type=\psline`. Wenn der Benutzer sagt, daß `\psset{branch=\customline}` sein soll, dann wird `\branch@type` durch eine `\customline`, anstatt einer `\psline` erstellt. Falls eine `\customline` hinzureichend definiert wurde, dann wird sie verwendet, um Zweige zu zeichnen. Ein paar alternative Makros zum Zeichnen von Zweigen sind bereits in jTree bereitgestellt. Damit der unternehmungslustige Benutzer z.B. Zickzack-Zweige zeichnen kann, muß er die `\customline` nur entsprechend dem PSTricks-Modell definieren. jTree bietet zudem `\blank` (siehe Beispiel 10 in Abschnitt 14 für eine nutzbringende Darstellung), `\brokenbranch` (siehe Beispiel 14 in Abschnitt 14), und `\etcbranch` (siehe Beispiel 13 in Abschnitt 14).

```
\jtree[xunit=3em,yunit=2em]
\! = :{normal}()
        [branch=\brokenbranch]{broken}
   :[branch=\blank]{blank}()
      [branch=\etcbranch]{etc}.
\endjtree
```



Die Proportion von “etc branch”, die gepunktet (dotted) ist, wird kontrolliert durch den Parameter `etcratio`. *pst-jtree* enthält `\psset{etcratio=.75}`, aber der Benutzer kann falls nötig selbst Hand anlegen. Der Style vor dem gepunkteten Teil wird definiert durch

```
\newpsstyle{etc}{nodesepB=0,nodesepA=1pt,linestyle=dotted,
  linewidth=1.2pt,dotsep=2pt}
```

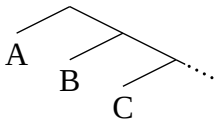
Der Benutzer kann die Spezifikation dieses Styles mit einer neuen Spezifikation überschreiben.

pst-jtree enthält die Makrodefinition:

```
\def\etc{[branch=\etcbranch,scaleby=.7]}
```

Also könnte man schreiben:

```
\jtree
\! = :{A} :{B} :{C}() \etc.
\endjtree
```



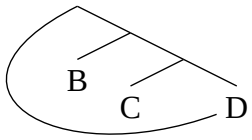
10.5. Die Pseudo-Parameter *dirA* und *dirB*

Nehmen wir an, daß man wünscht `\nccurve` zu benutzen, um eine Kurve zu zeichnen, die einen Knoten A in der selben Richtung hinterlässt, in der ein standardisierter linker Zweig A hinterlassen würde. Dazu stellst man `angleA` auf den entsprechenden Wert. Der richtige Winkel kann errechnet werden durch einfache Trigonometrie, aber die Kalkulation hängt vom Verhältnis der psxunits zu den psyunits ab. Es ist wünschenswert, eine trigonometrische Kalkulation an der Seite zu vermeiden und einen Baum so zu kodieren, daß Veränderungen an einer Einheit nicht die Geometrie im Ganzen verändert. `dirA` und `dirB` wurden zur Lösung dieses Problems eingeführt. `\psset{dirA=(-1:-1)}` wird den Winkel `angleA` so setzen, daß ein durch `\nccurve` gezeichneter Pfad den Anfangsknoten in der Richtung des Vektors $(-1,-1)$ belässt. Als Pseudoparameter genannt (meine Terminologie), wird er, weil `\psset{dirA=`*x*`}` ausgeführt wird, um einen

Effekt an `angleA`, und nicht an `dirA` zu erzielen. Es sei zu bemerken, daß ein Doppelpunkt benutzt wird, damit der `\psset`-Parser nicht verwirrt wird. `dirB` arbeitet fast auf die gleiche Weise wie `angleB`, aber der Vektor, welcher `dirB` bestimmt, deutet rückwärts entlang der Kurve, welche die Richtung ist, die `angleB` mißt.

Die Benutzung von dirA unten bewirkt, daß der "kurvige Zweig" (curved branch) sich zu den geraden Zweigen einreiht. Zu beachten ist, daß die Kurve ziemlich steif (ein hoher Wert von `ncurv`) eingestellt ist, sodaß sie sich genügend hinaus beugt.

```
\jtree
\! = @A1
   <right>
   :{B}
   :{C} {D}@A2 .
\nccurve[dirA=(-1:-1),angleB=200,
   ncurv=2,nodesepA=0]{A1}{A2}
\endjtree\kern1em
```

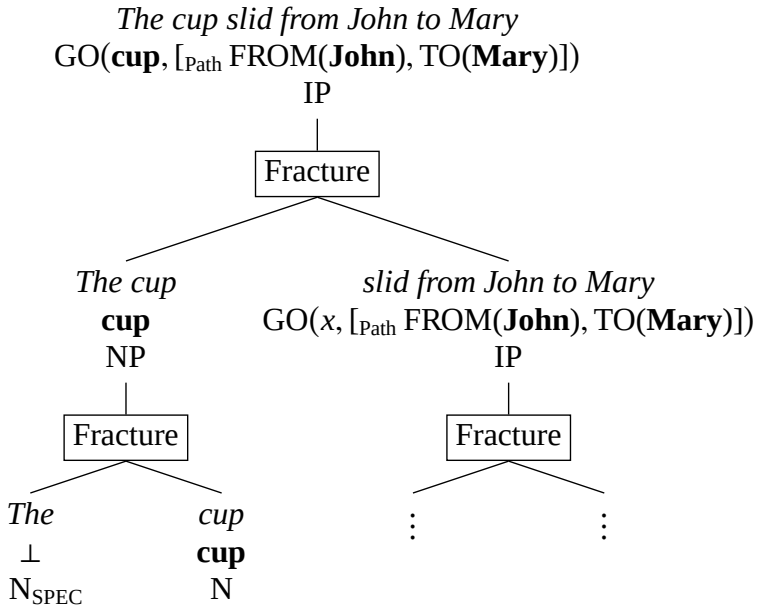


Das Beispiel in 14 ist eine gute Darstellung des Nutzens dieses Parameters.

11. Wie man komplexe Bäume erstellt

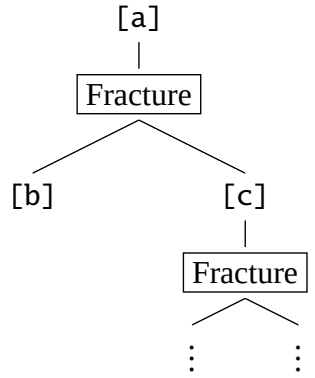
jTree hat einige Features, die es leicht machen, Bäume schrittweise zu erstellen. Das ist ein großer Vorteil bei komplexen Bäumen, da die Software es einem nicht einfach macht, den genauen Ursprung eines Fehlers zu lokalisieren. Sehen wir uns z.B. den Baum an, der zur Illustrierung der Eigenschaften von `qtree` (ein beliebtes Makropaket zum Zeichnen von Bäumen, verfügbar auf CTAN) verwendet wurde.

(15)



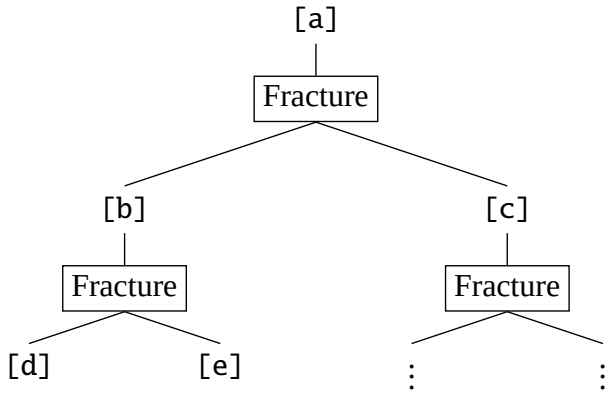
Wir fangen wie üblich mit dem rechten teil des Baumes an und fügen Verbindungspunkte für das komplexe Material hinzu, das auf der linken Seite gebraucht wird. Ähnlich den Adjunktionspunkten erlaubt es jTree, Labels zu verwenden, um später Sachen einzufügen. Die Makros `\stuff` und `\defstuff` sind in *pst-jtree* definiert und werden wie unten benutzt:

```
\jtree
\def\fracture{\psframebox{Fracture}}
\! = {\stuff[a]}
   <vert>{\fracture}
   :\jtbig{\stuff[b]}!a
      \jtbig{\stuff[c]}
   <vert>{\fracture}
   :{$\vdots$} {$\vdots$}.
\endjtree
```



Wir vervollständigen dann die Struktur an den markierten Stellen, fügen das fehlende ein und nehmen alle anderen Änderungen vor die nötig sind um einen gut aussehenden Baum zu erhalten. Einige Sachen sind schon offensichtlich: da ist zu viel Platz unter “Fracture” in der Box; die Äste des Baumes treffen nicht auf die “fracture box” und der Baum muss entlang der x-achse gestreckt werden. Wir warten noch mit dem Auffüllen der fehlenden Sachen bis zum Ende und konzentrieren uns auf die Problembehebung und darauf, die Struktur zu korrigieren. Das produziert:

```
\jtree[xunit=3em,yunit=1em]
\def\Fracture{<vert>{\omit\psframebox[boxsep=.4ex]
   {Fracture$\vphantom j$}}}%
\! = {\stuff[a]}
   \Fracture
   :\jtbig{\stuff[b]}!a
      \jtbig{\stuff[c]}
   \Fracture
   :{$\vdots$}
      {$\vdots$}.
\!a = \Fracture
   :{\stuff[d]}
      {\stuff[e]}.
\endjtree
```



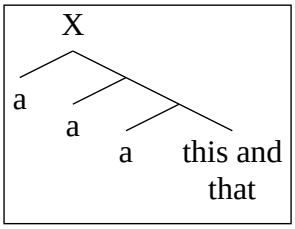
Jetzt sind wir bereit, die fehlende Sachen einzufügen. Da die ”go-path” Konstruktion kompliziert ist, zweimal erscheint und möglicherweise nützlich für diese Art von Arbeit ist, wurde es zu einem Makro zusammengefasst. *pst-jtree* beinhaltet `\multiline` und `\endmultiline` Makros um komplexe multiline-labels zu konstruieren. `\multiline` öffnet eine vbox und `\halign{\hfil#\hfil\cr`. `\endmultiline` schließt alles wieder. Der vertikale Abstand bekam einige Berücksichtigung. Der komplette Code für (15) ist:

```
\def\GO#1#2#3{$\rm GO\bigl({#1},[_{Path}\,
   FROM({#2}),TO({#3})]\bigr)$}

\jtree[xunit=3em,yunit=1em]
\defstuff[a]{\multiline
   \it The cup slid from John to Mary\cr
   \GO{\bf cup}{\bf John}{\bf Mary}\cr
   IP\endmultiline}
\defstuff[b]{\multiline
   \it The cup\cr
   \bf cup\cr
   NP\endmultiline}
\defstuff[c]{\multiline
   \it slid from John to Mary\cr
   \GO{\mit x}{\bf John}{\bf Mary}\cr
   IP\endmultiline}
\defstuff[d]{\multiline
   \it The\cr
   $\perp$\cr
  \quad N$\rm _{SPEC}$\endmultiline}
\defstuff[e]{\multiline
   \it cup\cr
   \bf cup\cr
   N\endmultiline}
\def\Fracture{<vert>{\omit\psframebox[boxsep=.4ex]
   {Fracture$\vphantom j$}}}%
\! = {\stuff[a]}
   \Fracture
   :\jtbig{\stuff[b]}!a \jtbig{\stuff[c]}
   \Fracture
   :{$\vdots$} {$\vdots$}.
\!a = \Fracture
   :{\stuff[d]} {\stuff[e]}.
\endjtree
```

12. Die bounding-box

PSTricks generiert dimensionslose Grafiken aber jTree verwendet eine menge Zeit damit die Größe der Bäume zu bestimmen und sie in eine angemeßen Große Box zu stecken. Zum Beispiel:

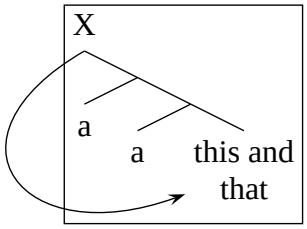
```
\psframebox[boxsep=0]{\jtree
\! = {X} :{a} :{a} :{a}
   {\multiline
      this and\cr
      that\endmultiline}.
\endjtree}
```



Die Bestimmung der Größe ist nicht perfekt. jTree ist nicht schlau genug um den weißen Raum aufgrund von `labelgapt`, `labelgapb`, `labelstrutt` und `labelstruttb` zu erkennen. Aber das ist nicht schlimm.

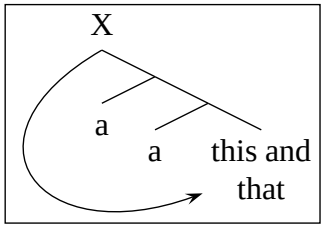
Wenn PSTrick benutzt wird um Pfeile zu zeichnen, verlaßen sie häufig die jTree Zeichen-Box.

```
\psframebox[boxsep=0]{\jtree
\! = {X}@A1
   <right>
   :{a}
   :{a}
      {\multiline
         this and\cr
         that\endmultiline}@A2 .
\nccurve[angleA=210,angleB=200,
   ncurv=2,nodesepA=0]{->}{A1:b}{A2}
\endjtree}
```



Das muss von Hand behoben werden, indem man die richtigen Wortabstände einfügt.

```
\psframebox[boxsep=0]{\kern2.4em
\jtree
\! = {X}@A1
   <right>
   :{a}
   :{a}
      {\multiline
         this and\cr
         that\endmultiline}@A2 .
\nccurve[angleA=210,angleB=200,
   ncurv=2,nodesepA=0]{->}{A1:b}{A2}
\endjtree}
```



13. Knoten und die Verbindung zwischen ihnen

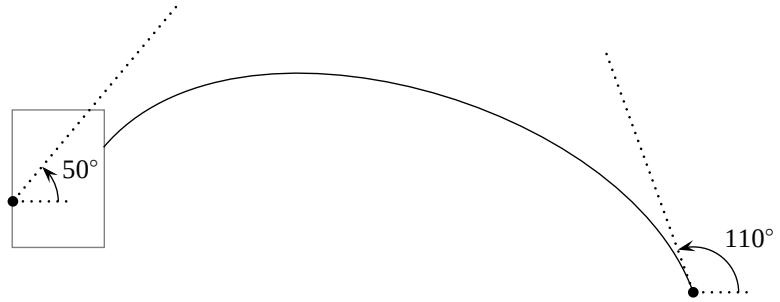
Knoten haben eine Form, ein referenz-Punkt und einen Namen. *pst-node* erlaubt dem Benutzer box, elliptisch, rund und Punkt-Knoten zu definieren. Zusätzlich zu der Form haben Knoten ein Referenzpunkt. Er ist in der Mitte des elliptischen, runden oder punkt-Knotens. Er kann auch in der Mitte eines Box-Knotens sein aber für Box-Knoten gibt es auch noch andere Optionen: die Enden und Zentren der Ecken und Basislinien. *pst-node* hat eine Vielzahl an Befehlen, die es erlauben, verschiedenste Verbindungen zwischen Knoten zu zeichnen. Wie die Verbindung aussehen soll, kann durch verschiedene Parameter (linewidth, linestyle, arrows, etc.) festgelegt werden. Der am meist nützliche Befehl um eine Verbindung zu zeichnen lautet `\nccurve`. Im nachfolgenden wird genau darauf eingegangen, wie `\nccurve` arbeitet.

Angenommen es gibt ein Box-Knoten und ein Punkt-Knoten wie unten zu sehen. Die Punkte stellen die Referenzpunkte dar.

Die Abbildung unten zeigt wie

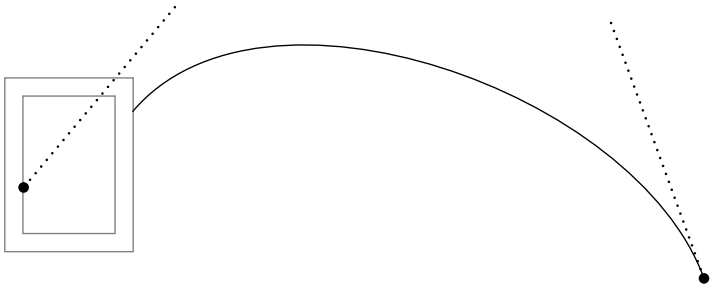
```
\nccurve[angleA=50,angleB=110]{A}{B}
```

gezeichnet wird.

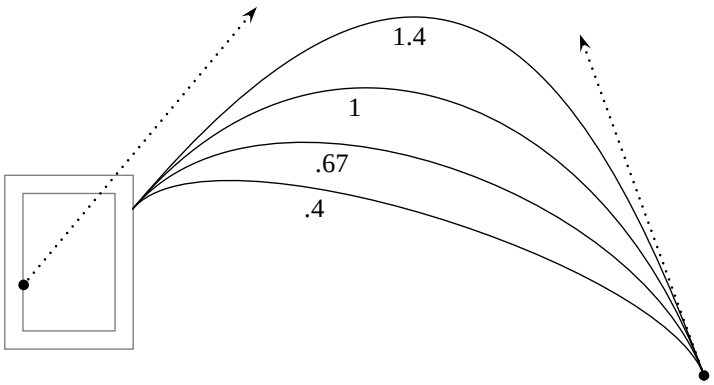


Diese einfache Abbildung kann durch eine Nummer von Parametern verändert werden. Zusätzlich zu den gewöhnlichen Parametern wie `linewidth`, `linestyle` und `arrows` die festlegen wie eine geometrische Linie gezeichnet wird, gibt es 6 weitere Parameter, die die Form der Linie direkt beeinflußen: `ncurvA`, `ncurvB`, `nodesepA`, `nodesepB`, `offsetA` und `offsetB`. Die Parameter können einzeln oder in paaren geändert werden. `\psset{nodesep=`*x*`}` impliziert `\psset{nodesepA=`*x*`,nodesepB=`*x*`}`. `ncurv` und `offset` arbeiten auf die selbe Weise. Wir werden die Parameter nacheinander untersuchen.

Mit `\psset{nodesepA=`*x*`}` wird die Dimension der Knoten-Box um *x* verändert, bevor die Linie berechnet wird.

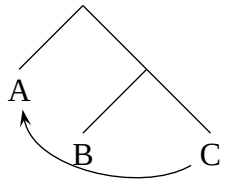


Der Effekt von `ncurv` ist raffiniert. Man kann sich vorstellen, das die Linie in die Richtung der Pfeile (siehe unten) “gezogen” wird. Mit `ncurv` gibt man die Stärke an mit der man zieht. Der standard-Wert ist .67. Die Abbildung unten zeigt ein paar Einstellungen von `ncurv`.



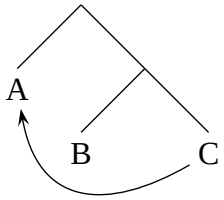
Das ist eine einfache Illustration zur Benutzung von ncurv. Es gibt ein Schriftsatz-Problem mit dem Zeiger unten welches behoben werden muss.

```
\jtree[unit=2em]
\! = :{A}@A :{B} {C}@C .
\nccurve[angleA=210,angleB=-80]{->}{C}{A}
\endjtree
```



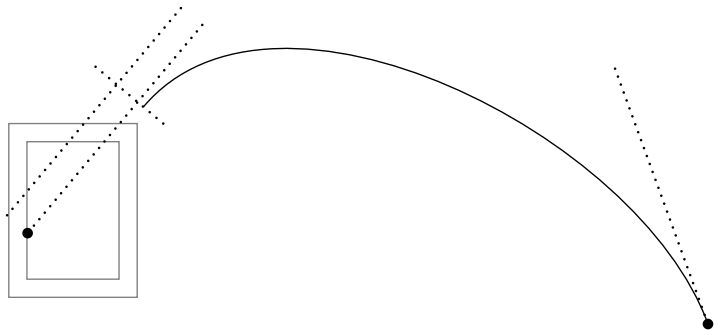
Eine Möglichkeit zur Lösung des Problems besteht darin, den Wert des ncurv Parameters zu erhöhen.

```
\jtree[unit=2em,nodesep=.6ex]
\! = :{A}@A :{B} {C}@C .
\nccurve[angleA=210,angleB=-80,
   ncurv=1.1]{->}{C}{A}
\endjtree
```



\psset{ncurv=*x*} hat den selben Effekt wie \psset{ncurvA=*x*,ncurvB=*x*}. Manchmal ist es vorteilhaft den Parametern verschiedene Werte zu geben. Experimente mit den Werten werden Ihnen schnell ein Gefühl für diese Einstellung geben.

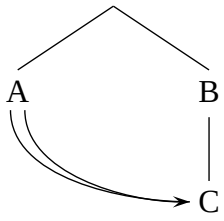
Zuletzt kommen wir zu den 2 Offset Parametern. Mit \psset{offsetA=*x*} wird der Anfangs-Punkt um die Distanz *x* senkrecht zur Anfangs-Richtung versetzt. Die Richtung ist links falls *x* positiv ist.



Mit \psset{offsetB=*x*} wird der End-Punkt um die Distanz *x* senkrecht zur end- Richtung versetzt. Denk daran, daß die end- Richtung auf den Knoten zeigt und nicht entlang der Linie.

Das ist eine einfache Anwendung, die für einige spezielle Schwerpunkte angebracht sein könnte.

```
\jtree[xunit=3em,yunit=2em]
\! = :{A}@A {B} <vert>{C}@C .
\psset{angleA=-90,angleB=180}
\nccurve[offsetA=.5ex]{->}{A}{C}
\nccurve[offsetA=-.5ex]{A}{C}
\endjtree
```

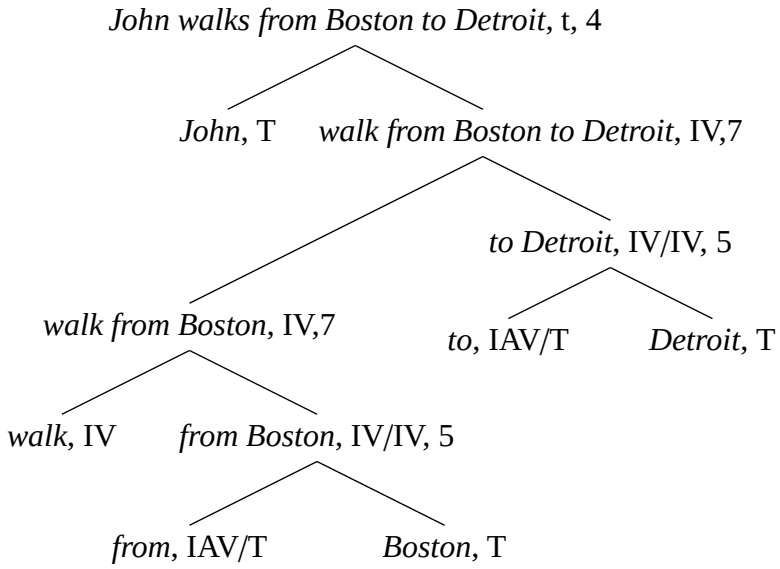


Dieses Konstrukt wird einige male in Beispiel 6 in Abschnitt 14 verwendet.

14. Beispiele

Dieser Abschnitt, der letzte, zeigt einige Beispiele mit vollständigem Code. Diese illustrieren Techniken um eine Vielzahl von Problemen zu lösen, die auftreten, wenn man einen komplexen Baum erstellt. Beispiel 1 und 2 zeigen verschiedene Techniken um Abstands-Probleme zu lösen. Die restlichen Beispiele zeigen das Zusammenspiel zwischen *pst-jtree* und *pst-node* mit deren Makros um Knoten zu erstellen und sie verschiedenartig zu verbinden.

Example 1



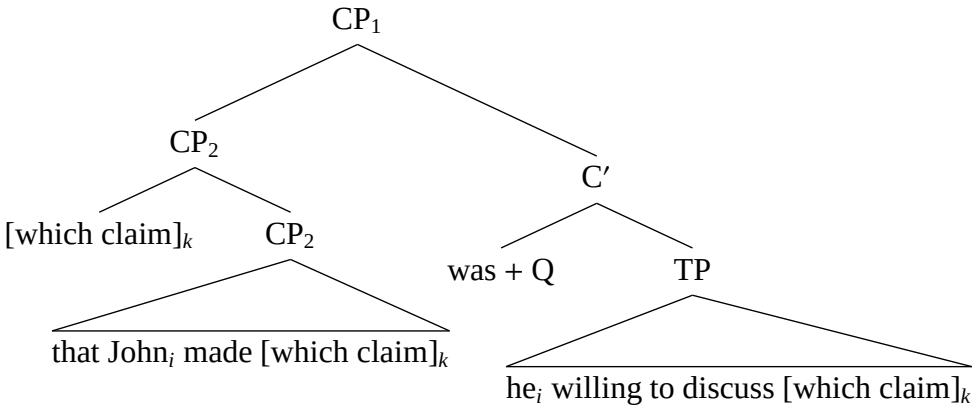
(Dowty, David. 1979. *Word Meaning and Montague Grammar*, p. 215.)

```
\jtree[xunit=4em,yunit=2em,everylabel=\strut\it]
\! = {John walks from Boston to Detroit\rm, t, 4}
   :{John\rm, T} {walk from Boston
                     to Detroit\rm,IV,7}[labeloffset=1.5em] [1]
   :\jtlong{walk from Boston\rm, IV,7}!a
      {to Detroit\rm, IV/IV, 5}
   :[scaleby=.8]{to\rm, IAV/T}() [scaleby=.8]{Detroit\rm, T}. [2]
\!a = :{walk\rm, IV} {from Boston\rm, IV/IV, 5}
   :{from\rm, IAV/T} {Boston\rm, T}.
\endjtree
```

1. Die Beschriftung ist versetzt, um den Abstand zu verbessern. Die Beschriftung ist nicht in der Mitte, aber sie ist nicht soweit versetzt, das es einen ablenken könnte.

2. Die Null-Inline-Adjunktion `()` verhindert, daß `[scaleby=.8]` als Parameter interpretiert wird.

Example 2



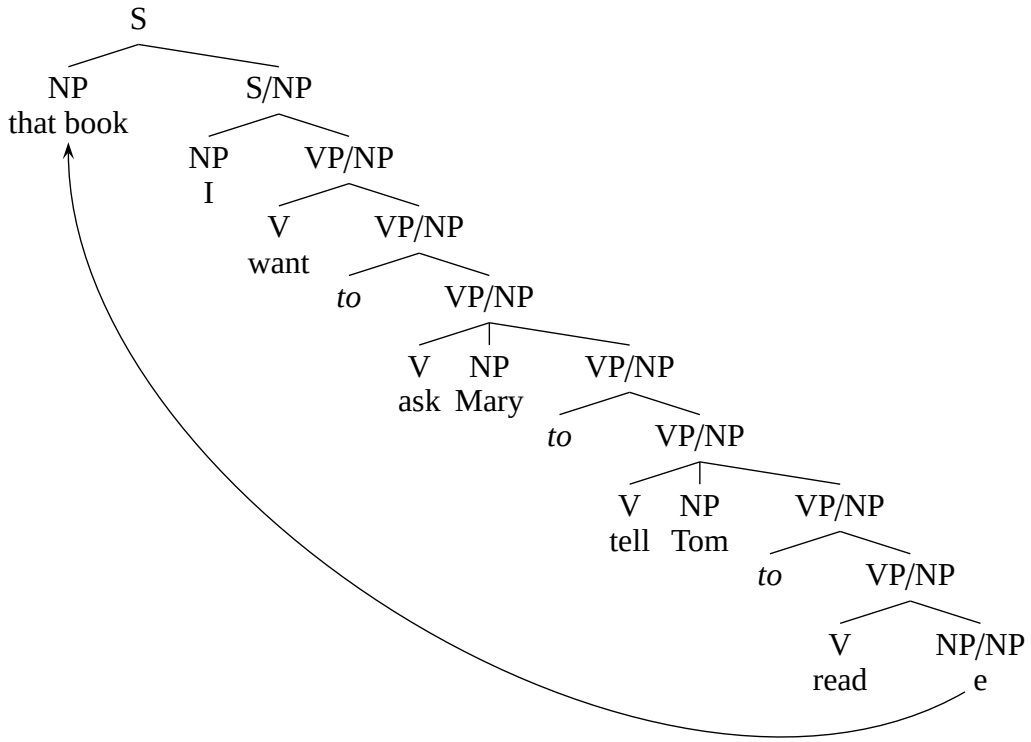
(Nunes, Jairo. 2004. *Linearization of Chains and Sideward Movement*, p. 149.)

```
\jtree[xunit=3em,yunit=1.4em]
\def\what{[which claim]$_k$}%
\! = {CP$_1$}
   :[scaleby=1.7]{CP$_2$}!a [scaleby=2.5]{C$'$}[1]
   :{$\rm was+Q$} {TP}
   <vartri>[scaleby=1.6,triratio=.4][2]
      {he$_i$ willing to discuss \what}.
\!a = :{\what} {CP$_2$}
   <vartri>[scaleby=1.6,triratio=.6][2]
      {that John$_i$ made \what}.
\endjtree
```

1. 1. Die zwei Abschnitte sind nicht gleich skaliert, da die Struktur im linguistischen Sinne nicht symmetrisch ist.

2. 2. Die Einstellungen von `triratio` ermöglichen eine kompaktere Darstellung

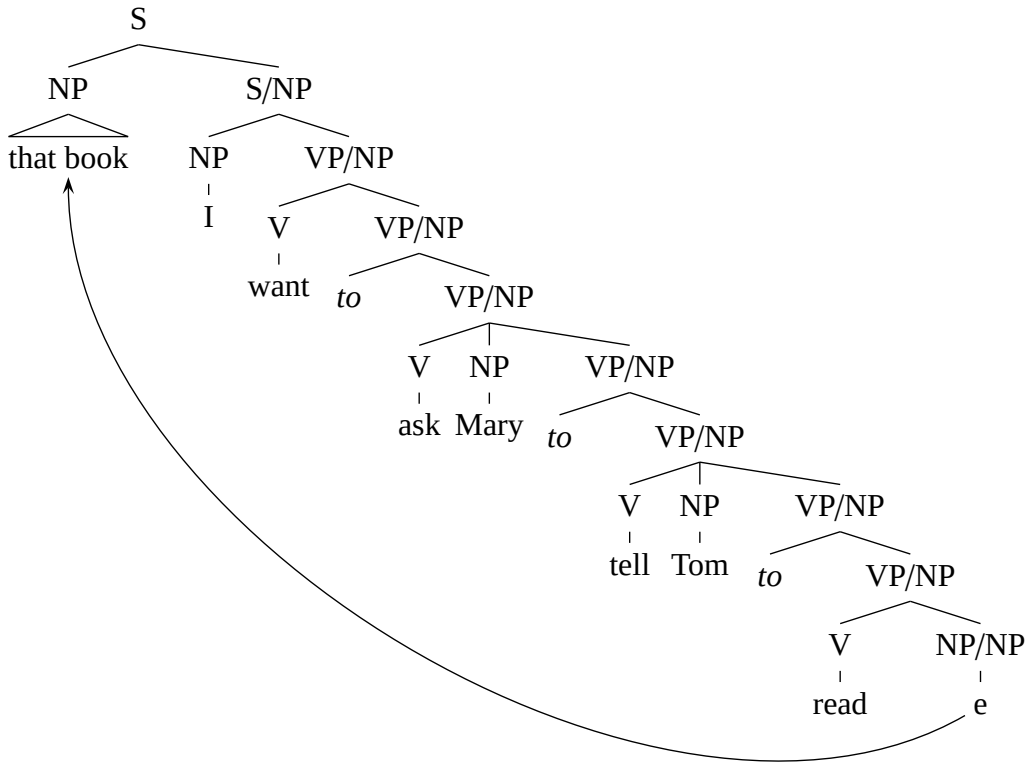
Example 3



(Dies ist eine Adaption eines Beispiels aus der Dokumentation zu ”Avery Andrew’s tree formating preproceor.)

```
\jtree[xunit=2.2em,yunit=.7em]
\def\\{[labelgapb=-.5ex]}%
\! = {S}
   :({NP}\\{that book}@A1 ) \jtwide{S/NP}
   :({NP}\\{I}) {VP/NP}
   :({V}\\{want}) {VP/NP}
   :{\it to} {VP/NP}
   <left>({V}\\{ask}) ^<vert>({NP}\\{Mary}) ^<wideright>{VP/NP}
   :{\it to} {VP/NP}
   <left>({V}\\{tell}) ^<vert>({NP}\\{Tom}) ^<wideright>{VP/NP}
   :{\it to} {VP/NP}
   :({V}\\{read}) {NP/NP}\\{e}@A2 .
\nccurve[angleA=210,angleB=-90]{->}{A2}{A1}
\endjtree
```

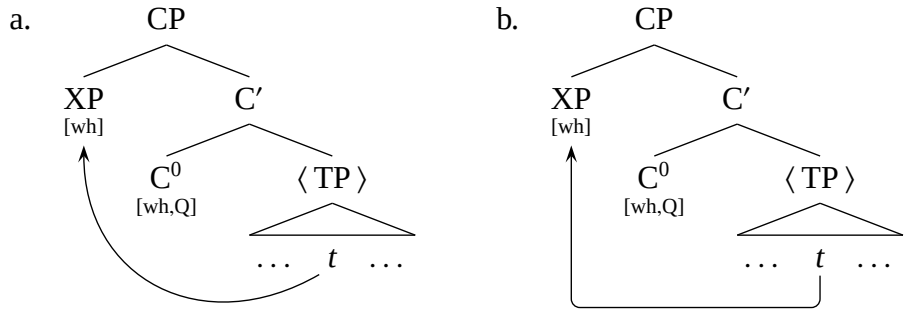
Einige könnten den folgenden Formatierungsstiel bevorzugen. Der Code ist fast identisch mit dem von oben. \\ hat eine andere Definition und <vartri> wird anstelle von \\ benutzt.




```
\jtree[xunit=2.2em,yunit=.7em]
\def\\{<shortvert>}%
\! = {S}
   :({NP}<vartri>{that book}@A1 ) \jtwide{S/NP}
   :({NP}\\{I}) {VP/NP}
   :({V}\\{want}) {VP/NP}
   :{\it to} {VP/NP}
   <left>({V}\\{ask}) ^<vert>({NP}\\{Mary}) ^<wideright>{VP/NP}
   :{\it to} {VP/NP}
   <left>({V}\\{tell}) ^<vert>({NP}\\{Tom}) ^<wideright>{VP/NP}
   :{\it to} {VP/NP}
   :({V}\\{read}) {NP/NP}\\{e}@A2 .
\nccurve[angleA=210,angleB=-90]{->}{A2}{A1}
\endjtree
```

Example 4

Der Kontrast zeigt die 2 Hauptarten von Zeigern die PSTricks zur Verfügung stellt um Knoten in einem Baum zu verbinden. Meine Lieblingsvariante ist (a), denn sie hebt deutlicher den Unterschied der Beziehung zwischen Zeiger- und Baumabhängigkeit hervor. Aber Geschmäcker sind ja verschieden.



(Die Version rechts ist von Jason Merchant.)

Der Code für (a) ist:

```
\jtree[xunit=2.6em,yunit=1em]
\def\\{[labelgapb=-1.2ex]}%\@1
\everymath={\rm}%
\! = {CP}
   :({XP}\\{$\scriptstyle [wh]$}@A1 ) {$C'$}
   :({$C^0$}\\{$\scriptstyle [wh,Q]$})
      {$\langle\, TP\,\rangle$}
   <tri>{\dots\quad\rnode[b]{A2}{\it t}\quad\dots}.\@2
\nccurve[angleA=-150,angleB=-90,ncurv=1]{->}{A2}{A1}
\endjtree
```

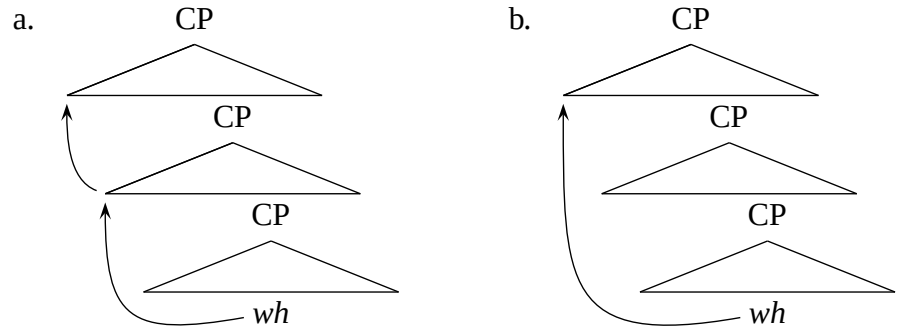
1. \\ wird benutzt um die Lücke zwischen den Kategorie Beschriftungen und den Kenndaten unter ihnen zu schließen.

2. Der Knoten A2 wurde mit \rnode eingefügt, anstatt @A2 zu verwenden, da @A2 kein Zeiger für das ganze Label ist.
2. The node A2 is inserted via \rnode rather than by using @A2 because it is not a pointer from the whole label, but from the trace inside the label. \rnode[b] is used, which places the reference point at the bottom of the box that the trace is put in. This works better visually, since it keeps the pointer from coming too close to the ellipsis.

For (b), substitute the following for the \nccurve pointer.

```
\ncbar[angleA=-90,angleB=-90,armA=1em,
   armB=1em,linearc=.6ex]{->}{A2}{A1}
```

Example 5

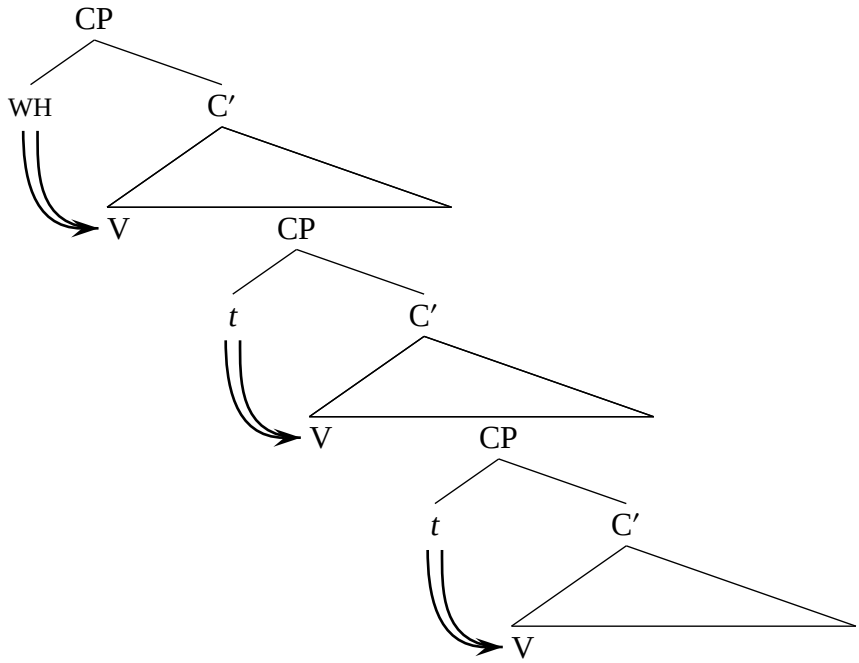


(Chung, Sandy. 1998. *The Design of Agreement*, p. 365.)

```
\jtree
\! = {CP}
   <left>@A1 ^<tri>[triratio=.65]{CP}\@1
   <left>@A2 ^<tri>[triratio=.65]{CP}
   <tri>{\it wh}@A3 .
\psset{angleB=-90,arrows=->}
\nccurve[angleA=190,ncurv=1.3]{A3}{A2}
\nccurve[angleA=160]{A2}{A1}
\endjtree

\jtree
\! = {CP}
   <left>@A1 ^<tri>[triratio=.65]{CP}
   <tri>[triratio=.65]{CP}
   <tri>{\it wh}@A3 .
\nccurve[angleA=190,angleB=-90,ncurv=1.3]{->}{A3}{A1}
\endjtree
```

1. Die <left>...^<tri> Konstruktion wurde verwendet so das das @tag gefolgt von <left> und das Label <tri> unabhängig positioniert werden können. Das Dreieck überschreibt einfach den linken Zweig. In anderen Situationen wäre etwas wie <left>[branch=\blank] vielleicht paßender.

Example 6



(Chung, Sandy. 1998. *The Design of Agreement*, p. 246.)

```
\jtree[xunit=2em,yunit=1.4em,labelgapb=0,triratio=0]\@1
\deftriangle<tri>(1.8)(1)(-.5)
\defbranch<colonB>(1)(-.5)
\! = {CP}
   :{\sc WH}@A {C$'$}
   <tri>{\rlap{V}}@AA ^<tri>[triratio=.55]{CP}\@2
   :{\it t}@B {C$'$}
   <tri>{\rlap{V}}@BB ^<tri>[triratio=.55]{CP}
   :{\it t}@C {C$'$}
   <tri>{\rlap{V}}@CC .
\psset{linewidth=1pt,ncurvB=1.1,nodesepA=1ex,
   angleA=-90,angleB=180,offsetA=.5ex}
\nccurve{A}{AA}
\nccurve{B}{BB}
\nccurve{C}{CC}
\psset{offsetA=-.5ex,arrows=->}
\nccurve{A}{AA}
\nccurve{B}{BB}
\nccurve{C}{CC}
\endjtree
```

1. Da alle Knoten-Beschriftungen groß geschrieben sind und daher die Grundlinie nicht unterschreitet, wurde der Abstand mit `labelgapb=0` verbessert. `triratio` wurde auf 0 gesetzt so das man die Instanzen von V korrekt positionieren kann.

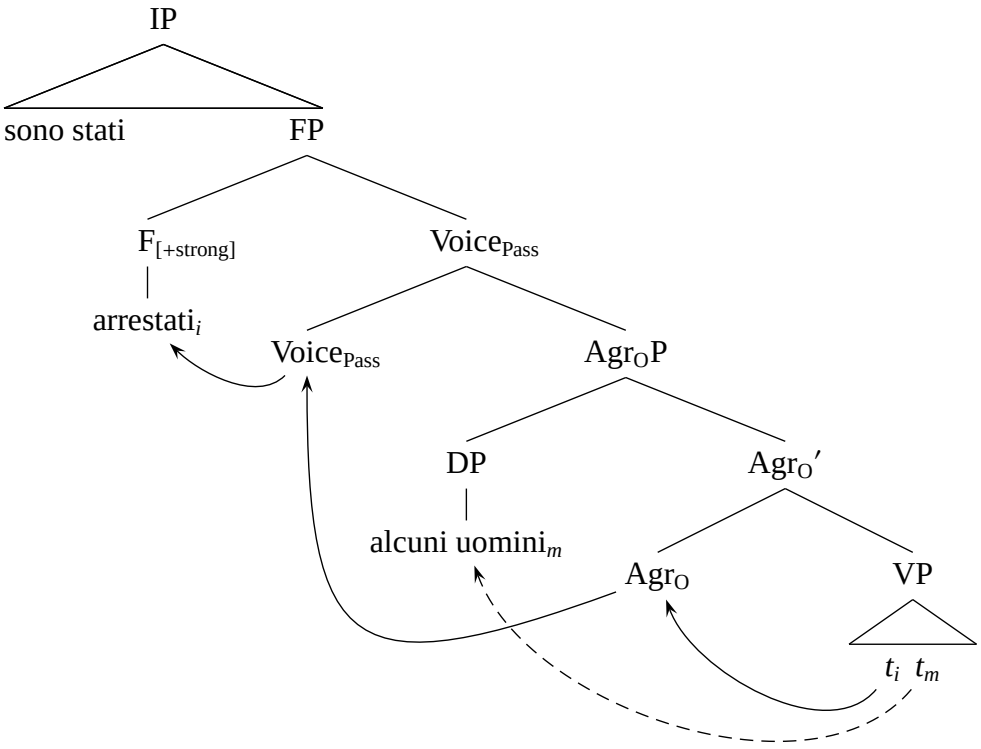
2. Das Dreieck wurde überschrieben, so das V und CP, mithilfe von unterschiedlichen Werten von `triratio`, unabhängig voneinander positioniert werden können.

Es folgt ein weniger eleganter Versuch, der den selben Erfolg ohne ein Offset erzielt.

```
\jtree[xunit=2em,yunit=1.4em,labelgapb=0,triratio=0]
\deftriangle<tri>(1.8)(1)(-.5)
\defbranch<colonB>(1)(-.5)
\def\gap{\hskip1ex}%
\def\\{[labelstrutb=0]}%
\! = {CP}
   :{\sc WH}\\({\pnode{A1}\gap\pnode{A2}}) {C$'$}\@1
   <tri>{\rlap{V}}@A ^<tri>[triratio=.55]{CP}
   :{\it t}\\({\pnode{B1}\gap\pnode{B2}}) {C$'$}
   <tri>{\rlap{V}}@B ^<tri>[triratio=.55]{CP}
   :{\it t}\\({\pnode{C1}\gap\pnode{C2}}) {C$'$}
   <tri>{\rlap{V}}@C .
\psset{linewidth=1pt,ncurvB=1.1,nodesepA=1ex,
   angleA=-90,angleB=180}
\nccurve{A1}{A}
\nccurve{B1}{B}
\nccurve{C1}{C}
\psset{arrows=->}
\nccurve{A2}{A}
\nccurve{B2}{B}
\nccurve{C2}{C}
\endjtree
```

1. Es ist notwendig `labelstrutb` 0 zu setzen (via \\) damit sich die inline-Verbindung unten an der Label-Box befindet. `labelgapb` ist im gesamten Baum 0.

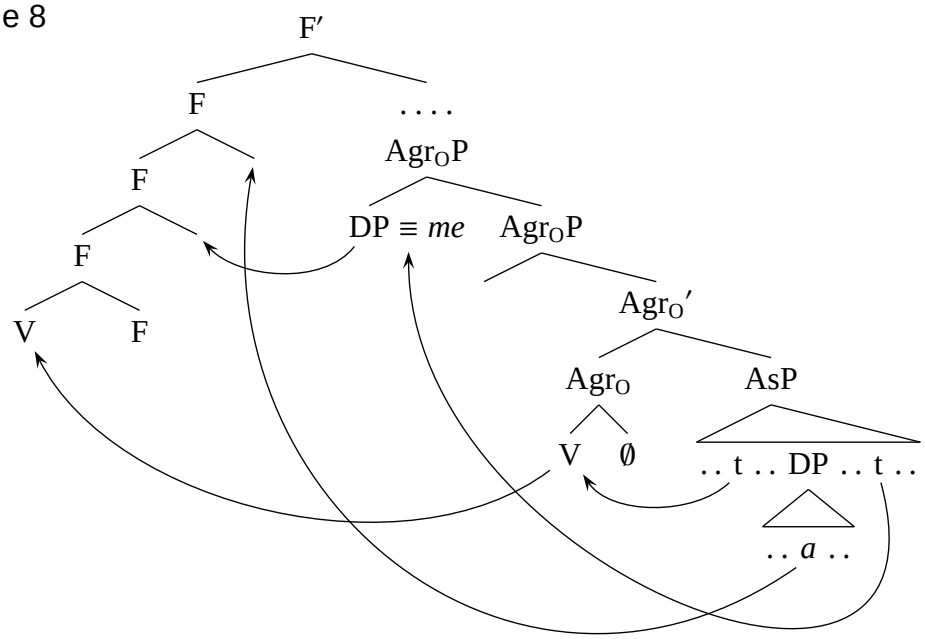
Example 7



(Caponigro, Ivano und Carson Schütze. 2003. Parameterizing Passive Participle Movement, *Linguistic Inquiry* 34.2, p. 300.)

```
\jtree[xunit=5em,yunit=2em]
\! = {IP}
   <tri>{\triline{sono stati\hfil}}  ^<tri>[triratio=.95]{FP}
   :{F$_{\rlap{$\scriptstyle\rm [+strong]$}}$}!a
      {Voice$_{\rlap{$\scriptstyle\rm Pass$}}$}
   :{Voice\rlap{$_{\rm Pass}$}}@A2  {$\rm Agr_OP$}
   :{DP}!b  {${\rm Agr_O}'$}
   :[scaleby=.8 1]{$\rm Agr_O$}@A3  [scaleby=.8 1]{VP}
   <tri>[scaleby=.4 .7]
      {\rnode{A5}{$t_i$}\hskip1ex \rnode{A6}{$t_m$}}.
\!a = <shortvert>{arrestati$_i$}@A1 .
\!b = <shortvert>{alcuni uomini$_m$}@A4 .
\psset{arrows=->}
\nccurve[angleA=225,angleB=-45]{A2}{A1}
\nccurve[angleA=200,angleB=-90,ncurv=1.5]{A3}{A2}
\nccurve[angleA=-130,angleB=-70]{A5}{A3}
\nccurve[angleA=-130,angleB=-70,linestyle=dashed]{A6}{A4}
\endjtree
```

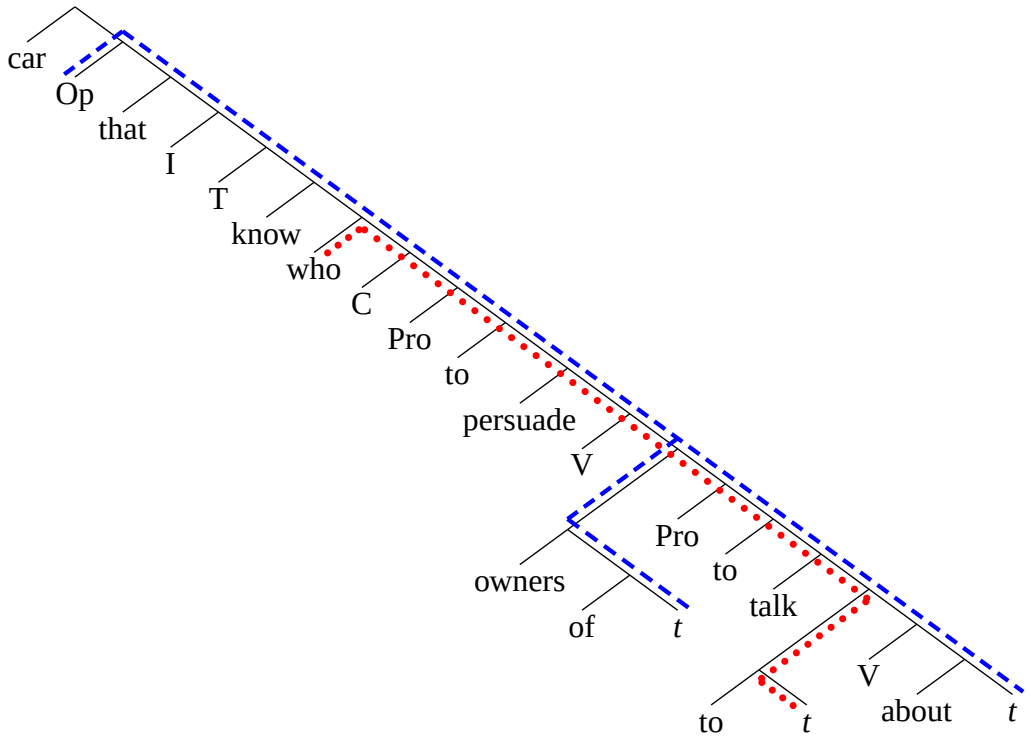
Example 8



(Uriagereka, Juan. 1995. Syntax of Clitic Placement in Western Romance, *Linguistic Inquiry* 26.1:115.)

```
\jtree[xunit=1.8em,yunit=.9em]
\def\*{\xleaders\hbox{\kern1pt.\kern1pt}\hfil}%
\deftriangle<triA>(1.3)(1)(-1/2)
\! = {F$'$}
   <wideleft>{F}!a
      ^<wideright>{\hbox to 2em{\*}}{$\rm Agr_O P$}
   <left>{DP\rlap{\rnode{B2}{$\;\equiv\;$}\it me}}@B1
      ^<wideright>{$\rm Agr_O P$}
   <left>  ^<wideright>{$\rm {Agr_O}'$}
   <left>{$\rm Agr_O$}!b  ^<wideright>{AsP}
   <triA>{\triline
      {\* \rnode[b]{D1}{t} \* DP \* \rnode[b]{D2}{t} \*}}
   <tri>[scaleby=.8 1.3]{\triline{\* \rnode[b]{E1}{\it a} \*}}.
\!a = :{F}!a1 @A1 .
\!a1 = :{F}!a2 @A2 .
\!a2 = :{V}@A3 {F}.
\psset{scaleby=.5 1}
\!b = :{V}@C1 {$\emptyset$}.
\psset{arrows=->}
\ncarc[arcangle=50]{C1}{A3}
\ncarc[arcangle=50]{B1}{A2:t}
\ncarc[arcangle=50]{D1}{C1}
\nccurve[angleA=-75,angleB=-90,ncurv=1.2,nodesepB=1ex]{D2}{B2}
\nccurve[angleA=-145,angleB=-100,ncurv=1]{E1}{A1:t}
\endjtree
```

Example 9



(Richards, Norvin. 2001. *Movement in Language: Interactions and Architectures*, S. 262.)

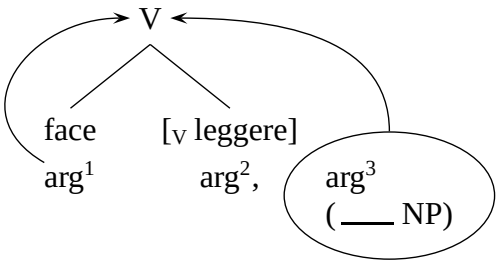
```
\jtree[xunit=1.5em,yunit=1.1em,labelgap=0]
\def\A#1{\pnode(0,.3){A#1}}%
\def\B#1{\pnode(0,-.3){B#1}}%
\! = :{car} {\omit\A1}
   :({\omit\A0}{Op})
   :{that}
   :{I}
   :{T}
   :{know} {\omit\B1}
   :({\omit\B0}{who})
   :{C}
   :{Pro}
   :{to}() \jtjot
   :{persuade}() \jtjot
   :{V} {\omit\A2}
   :\jtlong{\omit\A4}!a
   :{Pro}
   :{to}
   :{talk} {\omit\B2}
   :\jtlong{\omit\B3}!b
```

```
   :{V}
   :{about}() {\omit\A3}{\it t}.
\!a = :{owners}() \jtjot
   :{of} {\omit\A5}{\it t}.
\!b = :{to} {\omit\B4}{\it t}.
\psset{linestyle=dashed,linewidth=.3ex,
   linecolor=blue,nodesep=0}
\def\fudge{.5}%
\ncline[nodesepA=-\fudge]{A0}{A1}
\ncline[nodesepB=-\fudge]{A1}{A3}
\ncline{A2}{A4}
\ncline[nodesepB=-\fudge]{A4}{A5}
\psset{linestyle=dotted,linewidth=.5ex,linecolor=red}
\ncline[nodesepA=\fudge]{B0}{B1}
\ncline{B1}{B2}
\ncline{B2}{B3}
\ncline[nodesepB=\fudge]{B3}{B4}
\endjtree
```

Einige Veränderungen wurden vorgenommen nachdem der erste Beweis untersucht wurde. `xunit` wurde so verändert, daß die Darstellung so groß wie möglich war. `\jtjot` wurde benutzt um ein paar Zweige leicht auszudehnen um den Abstand zu verbeßern und `\fudge` wurde eingeführt, um die Endpunkte der gepunkteten und gestrichelten Linien zu verändern.

Eine Lösung mit Zuhilfenahme von `offset` ist auch möglich, allerdings ist es sehr viel schwerer damit sicherzustellen, daß sich die Segmente gut zusammenfügen.

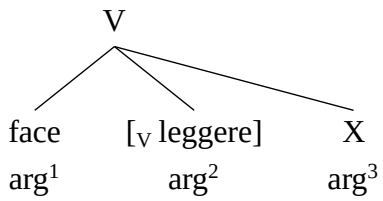
Example 10



(Zubizaretta, Maria Luisa. 1985. Morphology and Morphosyntax: Romance Causatives, *Linguistic Inquiry* 16.2, S. 276.)

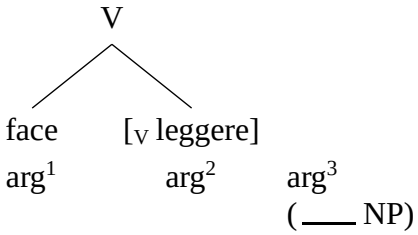
```
\jtree[xunit=2.5em,yunit=2em]
\def\ovalstuff{\vtop{\hbox{arg$^3$}%
   \hbox to 4em{(\thinspace \leaders\hrule\hfil\ NP)}}}%
\! = {V}@B1
   <left>{face}({arg$^1$}@B2 )
   ^<right>{$\rm [_V\,$leggere]}({arg$^2$,})
   ^<right>[scaleby=3 1,branch=\blank]
      {}{\ovalnode[framesep=1ex,boxsep=false]{K}{\ovalstuff}}.
\psset{arrows=->,nodesepA=0}
\nccurve[angleA=150,angleB=180,ncurv=1.2]{B2}{B1}
\nccurve[angleA=90,angleB=0,ncurv=.8]{K}{B1}
\endjtree
```

Um zu verstehen wie der Baum zusammengesetzt wurde, beachte folgendes:




```
\jtree[xunit=2.5em,yunit=2em]
\! = {V}
   <left>{face}({arg$^1$})
   ^<right>{$\rm [_V\,$leggere]}({arg$^2$})
   ^<right>[scaleby=3 1]{X}
   {arg$^3$}.
\endjtree
```

Dann:




```
\jtree[xunit=2.5em,yunit=2em]
\def\ovalstuff{\vtop{\hbox{arg$^3$}%
   \hbox to 4em{(\thinspace \leaders\hrule\hfil\ NP)}}}%
\! = {V}
   <left>{face}({arg$^1$})
   ^<right>{$\rm [_V\,$leggere]}({arg$^2$})
   ^<right>[scaleby=3 1,branch=\blank]{}
      {\ovalstuff}.
\endjtree
```

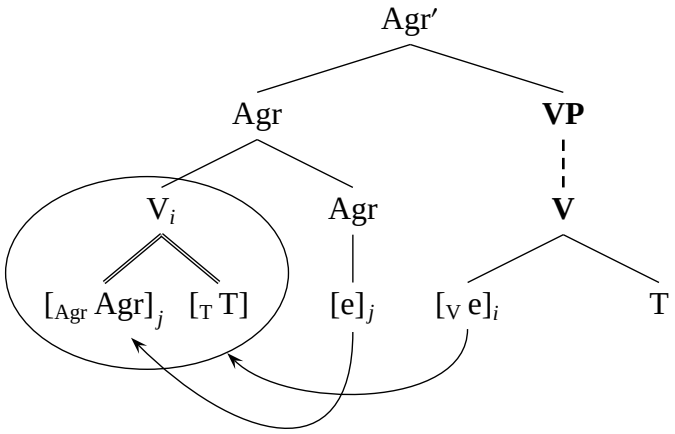
Der ovale Knoten wurde um `\ovalstuff` gebaut mithilfe von `\ovalnode`. Die Syntax ist:

`\ovalnode[`*pars*`]{`*name*`}{`*stuff*`}`

mit optionalen Parametern. Die Box wurde an allen Seiten durch |`framesep`| vergrößert und ein Oval wurde durch die vier Ecken der entstandenen Box gezeichnet. Wenn |`boxsep`| auf *false* gesetzt wird, ist die Knotenkonstruktion für TEX unsichtbar, ansonsten erschafft `\ovalnode` eine TEX-Box mit der Größe des Ovals.

Die Parameter Einstellungen bei `\ovalnode` in dem Beispiel sind wichtig! Wenn |`boxsep`| nicht *false* wäre, würde die Anordnung gestört werden.
|*framesep*| = 1 ex bedeutet, daß das Oval mit einem Abstand von 1 ex um die Box gezeichnet wird. Das hinterläßt eine visuell wichtige Spalte zwischen dem Oval und der umliegenden Box.

Example 11



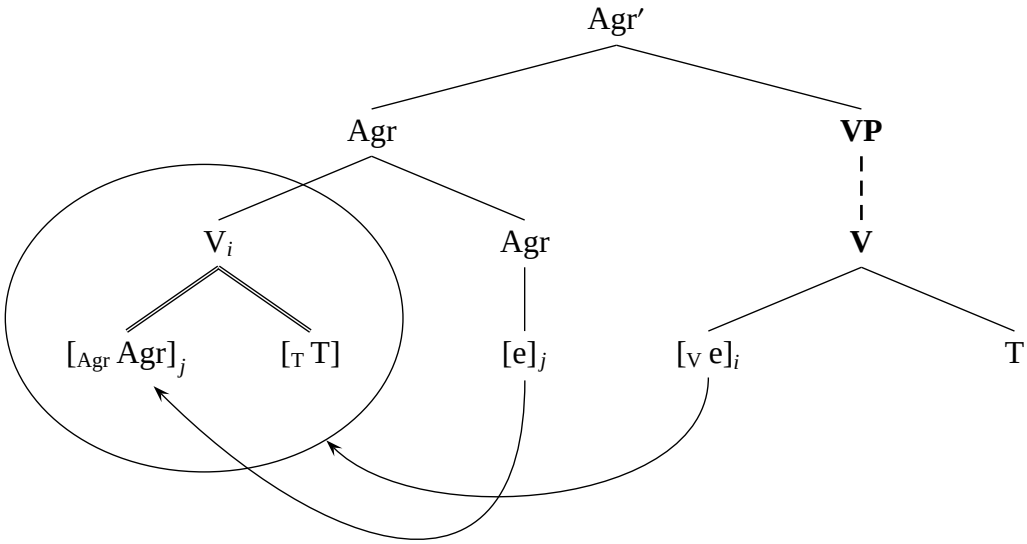
(Koopman, Hilda. 1995. On Verbs That Fail to Undergo V-Second, *Linguistic Inquiry* 26.1:150.)

```
\jtree[xunit=3em,yunit=1.5em]
\def\scaleA{[scaleby=1.6 1]}%
\def\scaleB{[scaleby=.6 1,doubleline=true,doublesep=.1ex]}%
\def\mkovalnode{\rput(-1ex,-.8)
   {\ovalnode[framesep=\psxunit]{K}{\hskip2em}}}%
\! = {$\rm Agr'$}
   :\scaleA{Agr}!a \scaleA{\bf VP}
   <vert>[linestyle=dashed,linewidth=1pt]{\bf V}
   :{${\rm [_V\,e]}_i$}@A1 {T}.
\!a = :{V$_i$}!b {Agr}
   <vert>{[e]$_j$}@A2 .
\!b = {\omit\mkovalnode}
   :\scaleB{${\rm [_{Agr}\,Agr]}_j$}@A3 \scaleB{$\rm [_T\,T]$}.
\psset{angleA=-90,angleB=-45,arrows=->}
\nccurve[nodesepB=0]{A1}{K}
\nccurve[ncurv=1.3]{A2}{A3}
\endjtree
```

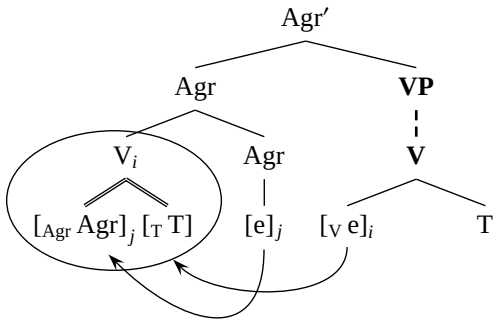
Die Schwierigkeit bei diesem Problem ist es, daß Oval zu zeichnen und in einen Knoten zu konvertieren, so das eine Knoten-Verbindung auf das Oval zeigen kann. `\mkovalnode`, das auf der Spitze des `!b` Teilbaums evaluiert wird, tut die ganze Arbeit. `\rput(-1ex,-.8)` positioniert das Zentrum des Ovales leicht links zu der Spitze und 80% des Weges zum Boden der 2 Zweige der Spitze (da sie eine Höhe von 1 haben). Die Größe des Ovals wird festgelegt durch |`framesep`| und die Box `\hbox{\hskip2em}`. Es braucht etwas ”trial and error” um die die richtigen Einstellungen für `\mkovalnode` zu finden, die ein brauchbares Oval erzeugen.

Es ist bemerkenswert, dass die Baum-Formatierung so einfach geändert werden kann. Das macht es sehr einfach, die Größe des Baums zu ändern, um ihn zum Beispiel vom Text zu einer Fußnote mit kleinerer Schrift zu verwandeln.

`\twelvepoint\jtree[xunit=4.8em,yunit=2em]`, ohne einen anderen Code einzubauen, produziert:



`\tenpoint\jtree[xunit=2.6em,yunit=1em]`, wieder ohne einen anderen Code einzubauen, produziert:



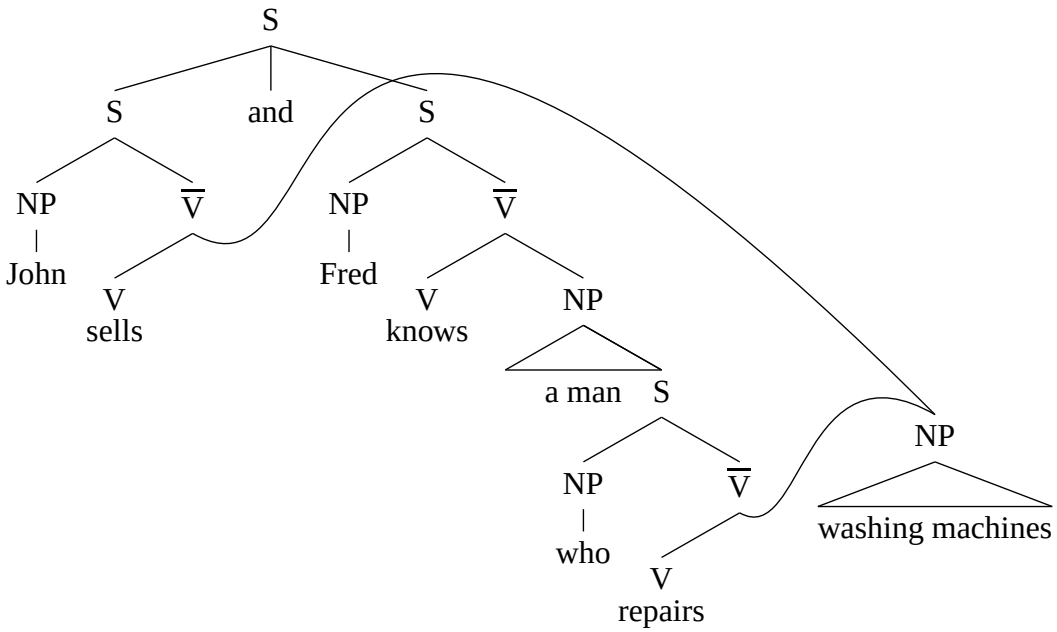
Das Oval ist etwas zu schmal, aber das kann einfach durch eine Erhöhung von `framsep` in Ordnung gebracht werden.

Example 12

Dieses Beispiel und die folgenden vier illustrieren einige Techniken, um multidominante Strukturen darzustellen. Ich habe dargelegt, dass diese Bewegung als Erstellung von multidominanten Bäumen mit geteilten Strukturen gesehen werden sollte. Mein Verdacht ist, dass die Schwierigkeit in der Darstellung der resultierenden Strukturen in einem Schriftsatzdiagramm die Idee weniger ansprechend erscheinen lässt. Es wäre bedauerlich, die Mängel der Schriftsatztechnologie auf die Entwicklung des theoretischen Werks Einfluss nehmen zu lassen. Was die Probleme beim Schreiben von *pst-jtree* aufwog, war, dass ich es als einen möglichen Beitrag zu der Entwicklung der Theorien über mentale Berechnung sehe.

Der Einfluss der Schriftsatztechnologie auf die linguistische Theorie sollte nicht unterschätzt werden. Die weitverbreitete Unaufmerksamkeit bezüglich autosegmentaler Strukturen in der Phonologie ist zumindest teilweise ein Abbild des Zustands der Schriftsatztechnologie.

Weil McCawley (1982) der erste Linguist zu sein scheint, der versuchte, Muiltidominanz zu nutzen, um ein komplexes Syntax-Problem zu lösen, werden wir mit einem seiner Beispiele beginnen (rechte Knoten Hebung).



```
\jtree[xunit=2.45em,yunit=1.4em,dirA=(1:-1),nodesep=0]
\def\\{[labelgapb=-4pt]}%
\def\V{$\rm \overline V$}%
\! = {S}
   <wideleft>{S}!a ^<vert>{and} ^<wideright>{S}
   :({NP}<shortvert>{Fred}) {\V}
   :({V}\\{knows}) {NP}
   <tri>{a man} ^<right>
   <right>[scaleby=3.5 1,branch=\blank]{NP}@A3 !b ^{S}
   :({NP}<shortvert>{who}) {\V}@A2
   <left>({V}\\{repairs}).
\!a = :({NP}<shortvert>{John}) {\V}@A1
   <left>{V}\\{sells}.
\!b = <vartri>{washing machines}.
\nccurve[angleB=150,ncurvB=1.4]{A2:b}{A3:t}
\nccurve[angleB=135,ncurvA=.5,ncurvB=2.6]{A1:b}{A3:t}
\endjtree
```

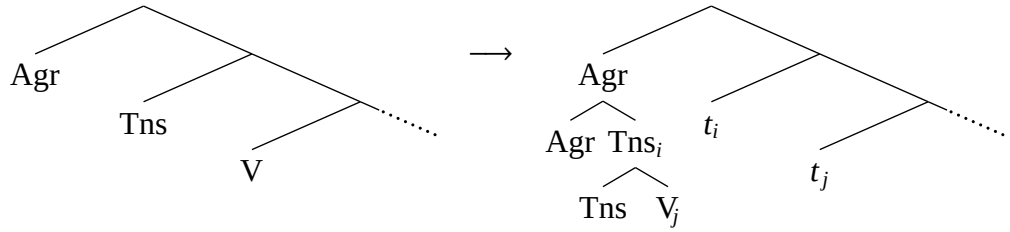
1.Ein leerer Ast wird benutzt, um den Knoten zu positionieren, der zwischen den beiden Verbindungen besteht.

2. Beachten Sie den hohen Wert von ncurvB, der gebraucht wird, um die Kurve genau festzulegen.

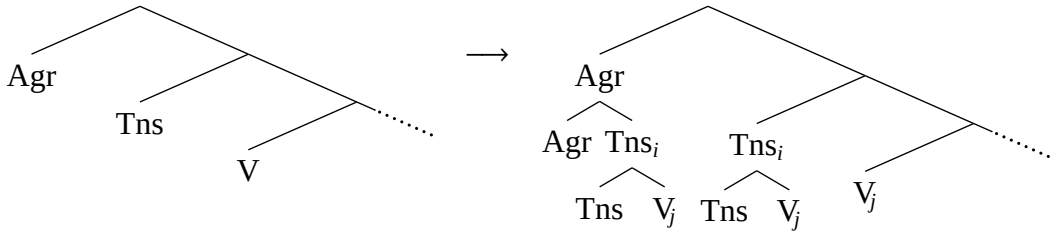
Example 13

Theorien des Verb Raisings

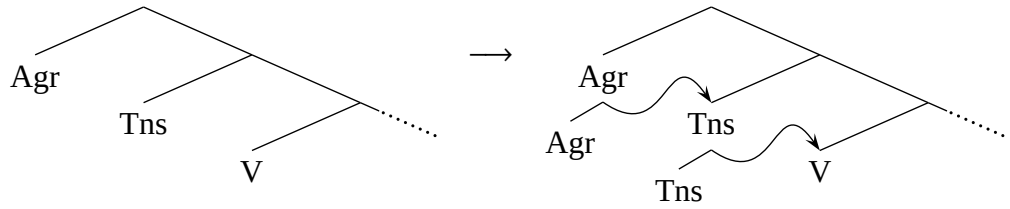
Trace-Theorie



Copy-Theorie



Geteilte Struktur




```
\def\Vj{V$\mskip-5mu _j$}
\psset{xunit=3.4em,yunit=1.5em,treevshift=1.3em}
```

Trace-Theorie

```
\jtree
\! = :{Agr} :{Tns} :{V}() \etc.\@1
\endjtree
\quad $\longrightarrow$\quad
\jtree
\! = :{Agr} !a
   :{$t_i$}
   :{$t_j$}() \etc.
\psset{scaleby=.3 .4}
\!a = :{Agr} {Tns$_i$}
   :{Tns} {\Vj}.
\endjtree
```

1. Ausführungen zu \ect können im Abschnitt 10 nachgeschlagen werden.

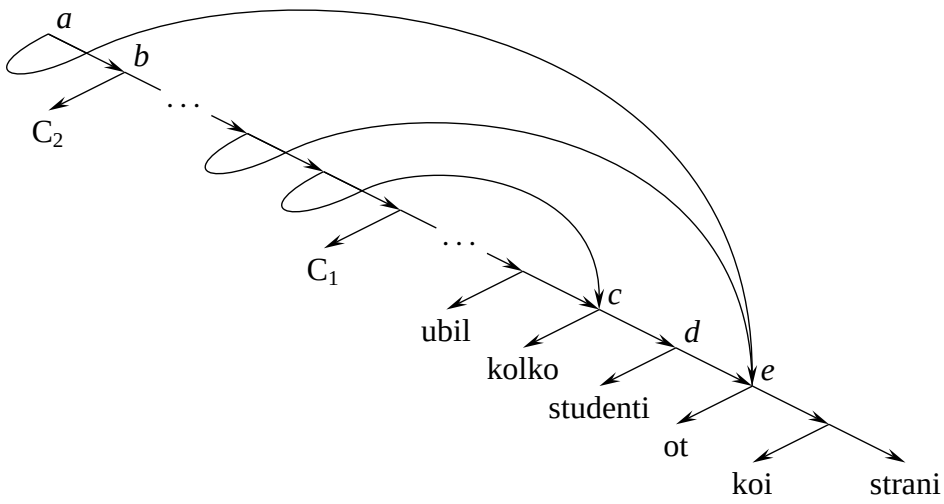
Copy-Theorie

```
\jtree
\! = :{Agr} :{Tns} :{V}() \etc.
\endjtree
\quad $\longrightarrow$\quad
\jtree
\! = :{Agr}!a [scaleby=1.45]
   :{Tns$_i$}!b
   :{\Vj}() \etc.
\psset{scaleby=.3 .4}
\!a = :{Agr} {Tns$_i$}
   :{Tns} {\Vj}.
\!b = :{Tns} {\Vj}.
\endjtree
```

Geteilte Strukturen

```
\jtree
\! = :{Agr} :{Tns} :{V}() \etc.
\endjtree
\quad $\longrightarrow$\quad
\jtree
\! = :{Agr}@A1 !a
   :{Tns}@A2 !b
   :{V}@A3 \etc.
\psset{scaleby=.3 .4,angleA=-35,angleB=125,ncurv=1.5,nodesep=0}
\!a = <left>{Agr}.
\!b = <left>{Tns}.
\nccurve{->}{A1:b}{A2:t}
\nccurve{->}{A2:b}{A3:t}
\endjtree
```

Example 14




```
\jtree[xunit=2.4em,yunit=1.2em,arrows=->,nodesep=0]
\def\broken{[branch=\brokenbranch,scaleby=1.6]}%\@1
\def\stub{<right>[scaleby=.5,arrows=-]}%\@2
\def\\#1{\rput[bl](.6ex,.4ex){\it #1}}%\@3
\! = {\omit\\a}@A1
   \stub @K1  ^<right>{\omit\\b}
   :{C$_2$}()  \broken @A2
   \stub @K2  ^<right>@A3
   \stub @K3  ^<right>
   :{C$_1$}()  \broken
   :{ubil}  {\omit\\c}@A4
   :{kolko}  {\omit\\d}
   :{studenti}  {\omit\\e}@A5
   :{ot} :{koi} {strani}.
\psset{dirA=(1:1),angleB=90,ncurvA=.6,ncurvB=1}\@4
\nccurve{K1}{A5}
\nccurve{-}{K2}{A5}
\nccurve{K3}{A4}
\psset{dirA=(-1:-1),dirB=(-1:-1),ncurv=4,arrows=-}\@5
\nccurve{A1}{K1}
\nccurve{A2}{K2}
\nccurve{A3}{K3}
\endjtree
```

1. Um Informationen über die Zweig-Parameter zu erhalten, können Sie in Abschnitt 5 nachlesen.

2. `\stub` wird benutzt, um Knoten auf halbem Weg auf bestimmten rechten Zweigen zu positionieren und so das Zeichnen der komplexen Pfeile zu erleichtern.

3. `\\` dient dazu, die Tags *a, b, c, . . .* zu positionieren.

4. dirA wird genutzt, um zu gewährleisten, dass die komplexen Pfeile parallel zu den linken Zweigen verlaufen, wenn diese die rechten Zweige kreuzen.

5. Ein sehr hoher Wert von ncurv bewirkt, dass die Kurve genügend ausgebeugt ist.

Eine Alternative zu \stub ist \psinterpolate um die Kreuzungspunkte zu positionieren.

```
\jtree[xunit=2.4em,yunit=1.2em,arrows=->,nodesep=0,
   arrowlength=3.6,arrowsize=2pt,arrowinset=.4]
\def\broken{[branch=\brokenbranch,scaleby=1.6]}%
\def\\#1{\rput[bl](.6ex,.4ex){\it #1}}%
\! = {\omit\\a}@A1
   <right>{\omit\\b}@A1a
   :{C$_2$}()  \broken @A2
   <right>@A3
   <right>@A3a
   :{C$_1$}()  \broken
   :{ubil}  {\omit\\c}@A4
   :{kolko}  {\omit\\d}
   :{studenti}  {\omit\\e}@A5
   :{ot} :{koi} {strani}.
\psinterpolate(A1)(A1a){.5}{K1}
\psinterpolate(A2)(A3){.5}{K2}
\psinterpolate(A3)(A3a){.5}{K3}
```

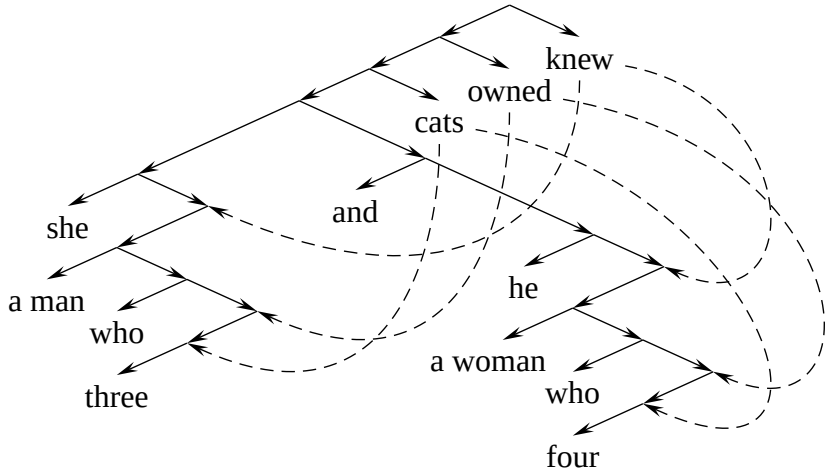
. . . wird wie oben fortgesetzt.

Example 15

Um zu zeigen, dass die rechte Knoten Hebung sich nicht generell auf die Konstituenten bezieht, hat Chris Wilder ein Beispiel gegeben:

> Er hat einen Mann, der drei, und sie hat eine Frau, die vier,
> Katzen besitzt, gekannt.

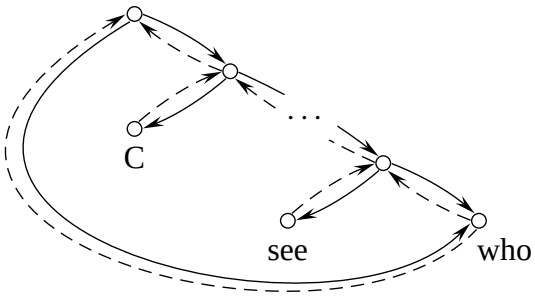
In meiner eigenen Arbeit über rechte Knoten Hebung, hatte ich die Gelegenheit, das folgende zu erstellen. Es enthält möglicherweise einige nützliche Dinge für jTree-Anwender, deshalb wurde es aufgenommen.




```
\jtree[dirA=(1:-1),nodesepA=0,nodesepB=.8ex,arrows=->,
   arrowlength=3.6,arrowsize=2pt,arrowinset=.4]
\! = :!a {\rnode{K1}{knew}}.
\!a = :!b {\rnode{O1}{owned}}.
\!b = :!c {\rnode{C1}{cats}}.
\!c =
   :\jtlong !d [scaleby=1.8]
   :{and}() [scaleby=2.4]
   :{he}() @K2
   <left>\jtjot !e .
\!d =
   :{she}() @K3
   <left>\jtjot !f .
\!e =
   :{a woman}[labeloffset=-1ex]
   :{who}() @O2
   <left>@C2
   <left>{four}.
\!f =
   :{a man}
   :{who}() @O3
   <left>@C3
```

```
   <left>{three}.
\psset{linestyle=dashed,arrows=<-}
\nccurve[angleB=-10,ncurvB=2,ncurvA=1.2]{O2}{O1}
\nccurve[angleB=-90,ncurvA=1.4]{O3}{O1}
\nccurve[angleB=-10,ncurvB=1.8,ncurvA=1.6]{K2}{K1}
\nccurve[angleB=-90,ncurvA=1.4]{K3}{K1}
\nccurve[angleB=-90,ncurvA=1.4]{C3}{C1}
\nccurve[angleB=-10,ncurvB=1.8,ncurvA=1.6]{C2}{C1}
\endjtree\kern6em
```

Example 16




```
\def\bilink(#1,#2)(#3,#4){%
   \pcarc(#1,#2)(#3,#4)%
   \pcarc[linestyle=dashed](#3,#4)(#1,#2)%
}
\jtree[xunit=3em,yunit=1.8em,style=arrows2,
   dirA=(-1:-1),branch=\bilink,nodesep=3pt,
   arcangle=10,offset=1pt,labelgapt=!3pt]
\def\@{\pscircle(0,0){3pt}}%
\! =
   {\pnode{A1}\@}
   <right>{\omit\@\pnode(.8,-.8){A3}}
   :({\omit\@}{C}) [scaleby=1.6,arcangle=7]{\omit\@}
   :({\omit\@}{see})
      ({\omit\@\pnode{A2}}{who}[labeloffset=.8em]).
\nccurve[angleB=225,ncurvA=1.95,ncurvB=1,offset=1.6pt]{A1}{A2}
\nccurve[angleB=227,ncurvA=2,ncurvB=1.02,offset=-1.6pt,
   linestyle=dashed,arrows=<-]{A1}{A2}
\rput(1.8,-1.8){\pscircle*[linecolor=white]{1em}}%
\rput(1.8,-1.8){\dots}
\endjtree
```

15. Kompatibilität

1. Frühere Versionen von jTree hatten die Syntax in (16b), im Gegensatz zu der Syntax in (16a), die in dieser Dokumentation erklärt und benutzt wurde.

(16) a. `\jftree`
Präliminardefinitionen. Parametereinstellungen.
`\! =` *einfache Baumbeschreibung*`.`
Definitionen, Parametereinstellungen, Grafik ohne Maßangabe
`\!a =` *einfache Baumbeschreibung*`.`
Definitionen, Parametereinstellungen, Grafik ohne Maßangabe
`\!b =` *einfache Baumbeschreibung*`.`
Grafik ohne Maßangabe
`\endjtree`

b. `\jftree`
Präliminardefinitionen. Parametereinstellungen
`\start` *einfache Baumbeschreibung*`.`
Definitionen, Parametereinstellungen, Grafik ohne Maßangabe
`\adjoin at !a` *einfache Baumbeschreibung*`.`
Definitionen, Parametereinstellungen, Grafik ohne Maßangabe
`\adjoin at !b` *einfache Baumbeschreibung*`.`
Grafik ohne Maßangabe
`\endjtree`

Die alte Syntax kann, wenn gewünscht, weiterhin verwendet werden. Es ist sogar möglich, die alte und die neue Syntax zu mischen.

2. TEX verwendet die Kontrollsequenz `\!` für negative Abstände im Mathematikmodus. jTree benennt `\!` innerhalb `\jtree` ... `\endjtree` um. Wenn gewünscht, kann diese Umbenennung durch Neudefinition der Zeichenliste `\jtEverytree` unterdrückt werden. Wenn `\jtree` aktiviert ist, werden zwei Zeichenlisten erweitert und bewertet. Erst `\jtEverytree` und dann `\jteverytree`. Die erste sollte für das Setup benutzt und wenig verändert werden, die zweite kann je nach Bedarf modifiziert werden. `\jteverytree` kann durch die Einstellparameter verändert werden. Bei `\jtEverytree` ist das nicht möglich, es kann aber neu definiert werden, indem *pst-jtree* bearbeitet oder in einer Setup-Datei, die nach dem Laden *pst-jtree* geladen wird, neueingestellt wird. Die Standardeinstellung ist:

```
\jtEverytree={\let\!\adjoinop}
```

`\!` wird unterdrückt, wenn es geändert wird in::

```
\jtEverytree={}
```

Wenn die spezielle Anwendung von `\!` unterdrückt wird, muss die Syntax (16b) genutzt werden.

In meiner eigenen Datei steht:

```
\jtEverytree={\everymath={\rm}\let\!\adjoinop}
```

Beim Setzen von linguistischen Bäumen bevorzuge ich häufig im Mathematikmodus den Roman-Schriftsatz gegenber dem kursiven Mathematikschriftsatz.

3. Der Parser der Baumdarstellung setzt voraus, dass die Symbole

$$\{., :, <, >, (,), ", @, !, \hat{}\}$$

nicht aktiv sind und den gleichen Kategorien-Code haben, den sie auch hatten als *pst-jtree* geladen wurde. Normalerweise haben alle außer ^ (mit Kategorien-Code 7, "superscript") Kategorien-Code 12, "other". Einige Makro Sets ändern diese Symbole. Das muss innerhalb `\jtree` ... `\endjtree` unterdrückt werden. Besonders das Babel-Paket macht erheblich Gebrauch von aktiven Symbolen. Unterdateien für bestimmte Sprachen erstellen generell verfügbare Makros, die diese Symbole in den Normalzustand umwandeln. Die dafür relevanten Befehle können in `\jtEverytree` oder `\jteverytree` eingebunden werden. Norerto Quiben, der von diesem Problem berichtete, hat herausgefunden, dass

```
\jteverytree={\spanishdeactivate{<>}}
```

das Kompatibilitätsproblem mit Spanish Babel löst. Seitdem wurde `\jtEverytree` jTree hinzugefügt und es scheint geeignet für den normalen Nutzer von Spanish Babel, den Deaktivierungsbefehl dort abzulegen als in dem flüssigeren `\jteverytree`. Nutzer, die andere Kompatibilitätsprobleme mit Babel lösen, können mir ihre Lösungen schicken, damit diese in eine zukünftige Version des Nutzerhandbuchs aufgenommen werden können.

4. Nutzer mit ungelösten Kompatibilitätsproblemen sollten mir davon berichten: *j.frampton@neu.edu*.

Appendix A. Installation und Abeitsumgebung

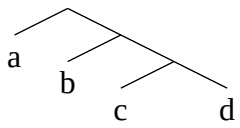
Ich nehme an, Sie haben bereits PSTricks und die XKeyVal-Pakete installiert, nun müssen Sie die Datei pst-jtree.tex an einem Ort ablegen, an dem sie gefunden werden kann. Das Verzeichnis, das *pst-jtree.tex* enthält, ist ein naheliegender Ort, aber wenn Sie wissen, wie, ist es vermutlich besser, wenn Sie Ihren eigenen lokalen TEX-Unterbaum anlegen, sodass das Update ihrer TEX-Dateien mit einer neuen TEX-Distribution nicht *pst-jtree.tex* zerstört. Wenn Sie jTree mit LaTeX nutzen wollen, müssen Sie dasselbe mit *pst-jtree.sty* machen. Wenn schließlich die Wiederherstellung der TEX-Dateien mit einer Indexierungsmethode abgeschlossen ist, müssen Sie das Indexierungsprogramm durchlaufen lassen, damit die Speicherstellen von *pst-jtree.tex* und *pst-jtree.sty* richtig indexiert sind. Die Distribution von jTree besteht nur aus drei Dateien:*pst-jtree.tex, pst-jtree.sty* und *pst-jtree-doc.pdf* (welches Sie sich gerade ansehen).

Bevor Sie fortfahren, sollten Sie sicherstellen, dass Sie ein einfaches Beispiel ausführen und ansehen können. Wenn Sie ein Nutzer von LaTeX sind, nutzen Sie Verfahren (17a), als TEX-Nutzer (17b).

(17) a.
```
\documentclass{article}
\usepackage{pstricks}
\usepackage{pst-xkey}
\usepackage{pst-jtree}
\begin{document}
\jtree
\! = :{a} :{b} :{c}{d}.
\endjtree
\end{document}
```

b.
```
\input pstricks
\input pst-xkey
\input pst-jtree
\jtree
\! = :{a} :{b} :{c}{d}.
\endjtree
\bye
```

Ihr DVI-Viewer mag ausreichend Postscript-Code verstehen, um die DVI-Datei darzustellen. Sie sollten den unten stehenden Baum sehen, links ausgerichtet.

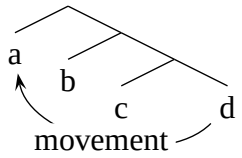


PSTricks arbeitet mit dem TEX-Befehl `\special`, um den Postscript-Code in die DVI-Datei, die TEXproduziert, einzubauen. Einige DVI-Viewer werden den eingebauten Postscript-Code einfach ignorieren und Sie werden einen Konverter für DVI zu Postscript (z.B. das Programm *dvips*) brauchen, um eine Postscript-Datei zu erstellen, die mit einem Postscript-Viewer wie Ghostscript angesehen werden kann. Auch wenn Ihr DVI-Viewer die Ausgabe von Beispiel (17) handhaben kann, das eine sehr einfache Einrechnung von Postscript hat, so ist die erfolgreiche Nutzung von jTree, die alle PSTricks-Tricks umfasst, nur dann gewährleistet, wenn Sie *dvitops*-Konverter benutzen. Andernfalls werden Sie nie die volle Darstellungskraft von jTree erleben. Wenn Ihr DVI-Viewer erfolgreich

(17) darstellen kann, versuchen Sie (18a). Wenn er die Postscripteinrechnung erfolgreich erfasst, wird er (18b) produzieren.

(18) a.
```
\jtree
\! =
   :{a}@A1
   :{b}
   :{c} {d}@A2 .
\nccurve[angleA=225,angleB=-80]{->}{A2}{A1}
\mput*{movement}
\endjtree
```

b.



Ich nehme an, der Test hat nicht funktioniert und Ihr DVI-Viewer hat (18b) nicht produziert. Der Sinn dieser Übung war, Ihnen zu zeigen, dass Sie die Konvertierung von Postscript zu DVI und die Fähigkeit, Postscript-Dateien anzusehen. Bei meiner eigenen Arbeit lasse ich diesen Schritt oft aus (um einige Minuten zu sparen) und verwende meinen DVI-Viewer (Windvi), der bei grundlegenden Postscripteinrechnungen gute Arbeit leistet. Aber ich nutze Postscript schon lange genug, um sofort zu bemerken, wenn die Einrechnungen zu komplex für Windvi werden und schalte dann um zu vollständiger Konvertierung und Ansicht mit Ghostscript, dem Standard-Viewer für Postscript.

Konvertieren Sie die Ausgabe von (18a) zu einer Postscript-Datei und sehen Sie sie mit einem Postscript-Viewer an. Jetzt sollten Sie (18b) sehen. Bis Sie die DVI- zu einer Postscript-Datei umwandeln und ansehen können, können Sie nicht wissen, ob eine fehlerhafte Darstellung durch einen Programmierungsfehler oder die Unfähigkeit Ihres DVI-Viewer, den in die DVI-Datei eingebauten Postscript-Code korrekt darzustellen. Wenn Sie die Testdatei nicht erfolgreich erstellen und die DVI- zu einer Postscript-Datei konvertieren und ansehen können, versuchen Sie es mit PSTricks und/oder mit Konvertierung von DVI zu Postscript bevor Sie fortfahren.

Arbeitsumgebung: Weil jTree eher auf Verstellbarkeit als auf automatischer Größeneinstellung beruht, ist es wichtig, eine gute TeX/LaTeX-Umgebung zu schaffen, die Sie die Ergebnisse von Änderungen schnell und ohne Probleme sehen lässt. Eine gute interaktive Tex-Umgebung verringert die Zeit zwischen der Änderung im Editorprogramm und dem Anschauen des Ergebnisses auf dem Bildschirm. Ihr Editorprogramm , der DVI-Viewer, und der Postscript-Viewer sollten aktiv bleiben und Sie sollten imstande sein, den einen oder den anderen in den Vordergrund zu bringen. Ihre Viewer sollten so konfiguriert sein, dass sie in der Datei bleiben, die sie darstellen. Wenn eine neue DVI- oder Postscript-Datei erstellt wird, sollte Ihr Viewer sie automatisch laden und am selben Platz

positioniert sein (Seite und xy-Position) wie in der alten Datei. Sie werden die Durchlaufzeit zwischen Bearbeitung und Ansehen des Ergebnisses auf wenige Sekunden reduzieren können (bei schnellem PC).

Index der Befehle, Parameter und speziellen Einstellungen

Die Symbole =, <, >, ^, [,], {, }, (,), :, @, und ! haben eine spezielle Bedeutung für das Parsing der Baumdarstellung. Beachten Sie die Diskussion der Syntax der Baumdarstellung auf Seite 25.