## Seminar Compilerbau und Verifikation

# HappyGLR - Parsen mehrdeutiger Grammatiken

**im SoSe 2006 bearbeitet von**
**Christina Bianca Heitzer**

**betreut von**
**Prof. Dr. R. Loogen**

**Fachbereich Informatik**

**08. Mai 2006**

## Zusammenfassung

Die Technik des LR-Parsens erlaubt ein effizientes Parsen eindeutiger, kontextfreier Grammatiken. Was aber, wenn man eine mehrdeutige Grammatik gegeben hat? Mit der Problematik und Lösung dieses Problems wollen wir uns im Folgenden beschäftigen. Dabei betrachten wir zuerst einen von Masaru Tomita entwickelten Algorithmus, der es erlaubt effizient auch mehrdeutige Grammatiken zu parsen. Nach einigen einführenden Beispielen zu den speziellen verwendeten Datenstrukturen, dem 'Graph-Structured-Stack' und dem 'Parse-Forest', wenden wir uns der Implementation in einer funktionalen Umgebung und der Integration in den Haskell Parser-Generator Happy zu.

# Inhaltsverzeichnis


**1 Einführung** **4**

**2 Tomita's Algorithmus** **6**
2.1 Der 'Parse-Forest' . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2 Der 'Graph-Structured-Stack' . . . . . . . . . . . . . . . . . . . . . . 8

**3 Implementation** **12**
3.1 Der 'Parse-Forest' . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2 Die Datenstruktur für den 'Graph-Structered-Graph' . . . . . . . 14

**4 Integration in Happy** **17**

**5 Schlußbemerkung** **18**

**6 Literatur** **19**

# 1 Einführung

Eine Vielzahl von kontextfreien Grammatiken können nicht deterministsich mit der LR-Technik geparst werden, da es bei der Erzeugung der Parse-Tabelle zu Konflikten kommt. Dies ist bei mehrdeutigen Grammatiken immer der Fall, aus ihnen lassen sich zu einem Wort mehrere Ableitungsbäume produzieren. Als ein einfaches Beipsiel nehmen wir die mehrdeutige Grammatik

$$E \rightarrow E + E \mid i \tag{1}$$

Hier lassen sich für die Eingabe i+i+i zwei Ableitungsbäume aufstellen:

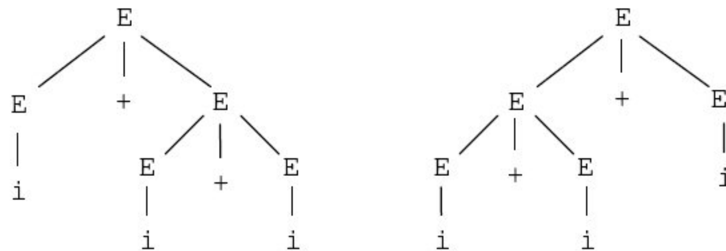



Abbildung 1: Parse-Bäume zur Eingabe i+i+i

Des Weiteren können wir der Parse-Tabelle einen Shift-Reduce-Konflikt entnehmen:

| | Action | | | Goto |
|---|---|---|---|---|
| | i | + | $ | E |
| 0 | sh3 | | | 4 |
| 1 | sh3 | | | 2 |
| 2 | | sh5 | | |
| 3 | | re2 | re2 | |
| 4 | | sh5 | acc | |
| 5 | sh3 | | | 6 |
| 6 | | sh5/re1 | re1 | |

Abbildung 2: Parse-Tabelle zur Grammatik (1)

Natürlich ist bei einer solchen simplen Grammatik die Auflösung der Mehrdeutigkeit durch Präzedenzregeln möglich; bei komplexeren mehrdeutigen Grammatiken, wie wir sie später noch sehen werden, ist dies nicht mehr der Fall.

Um nun mehrdeutige Grammatiken effizient zu parsen, kann eine Erweiterung des LR-Parsens benutzt werden, das 'Generalised-LR-Parsing' - kurz GLR. Ein GLR-Parser behebt die auftretenden Konflikte in der Parse-Tabelle, indem beide Aktionen durchgeführt werden, ausgehend von der GLR-Idee geschieht dies parallel. Hierfür wird der Parse-Stack des einfachen LR-Parsers durch eine Datenstruktur ersetzt, der mit diesen mehrfachen Aktionen umgehen kann. Nach Beendigung eines erfolgreichen Parse-Vorgangs wird somit auch nicht ein einzelner Parse-Baum ausgegeben, sondern ein ganzer Wald von Bäumen, der 'Parse-Forest'.

Es existieren mehrere Ansätze zur Lösung des Problems. Wir wollen uns hier mit Tomita's Algorithmus beschäftigen. Masaru Tomita entwickelte in den achtziger Jahren einen Algorithmus, indem der Parse-Stack eines LR-Parsers durch einen 'Graph-Structured-Stack' ausgetauscht wird [Tom85]. Wie der Name vermuten lässt, handelt es sich bei dem GSS um eine Graphstruktur, die es ermöglicht die unterschiedlichen Parse-Ergebnisse einer Eingabe zu repräsentieren. Die Operationen Shift und Reduce ähneln den Aktionen des Standard-LR-Parsers. Das Ergebnis eines erfolgreichen Parse ist ein gepackter 'Parse-Forest', der alle gültigen Parses der Eingabe beinhaltet.

Ausgehend von zwei Beispielen wollen wir uns der Lösung dieses Problems annehmen. Zum einen bedienen wir uns der bereits oben angesprochenen simplen, mehrdeutigen Grammatik '$E \rightarrow E + E \mid i$'. Zum anderen betrachten wir eine Grammatik zur Beschreibung natürlicher Sprache, ein Beispiel für eine sehr komplexe Mehrdeutigkeit in der es auch auf die Bedeutung und Ausnutzung von ergänzenden semantischen Informationen ankommt.

# 2 Tomita's Algorithmus

Der traditionelle Ansatz dieses Algorithmus basiert auf imperativen Programmierkonzepten, dennoch sind die beschriebenen Techniken auch für einen funktionalen Ansatz relevant und werden dementsprechend zuerst im Allgeimen, dann im funktionalem Umfeld aufgeführt.

## 2.1 Der 'Parse-Forest'

Die Ausgabe des Parse-Algorithmus ist eine Sammlung von gültigen Parse-Bäumen, entsprechend einer gegebenen Garmmatik und einem Eingabe-String. Am naheliegendsten wäre jetzt natürlich einfach die Ausgabe der Menge aller möglichen Ableitungsbäume. Aber gerade bei Grammatiken mit hoher Mehrdeutigkeit, wie wir sie zum Beispiel bei der natürlichen Sprache finden, ist dies ein Ding der Unmöglichkeit, da die Kosten für die Berechnung, Speicherung und Manipulation solch großer Datenmengen immens sind. Als Beispiel wollen wir den Satz 'I own a car' angeben, der mit der Software LOLITA für die Analyse natürlicher Sprache, über 13.000 mögliche syntaktische Analysen liefert [Cal 97]. Tomita schlägt nun in seinem Algorithmus für die Repräsentation dieser Sammlung von Bäumen eine Datentruktur, genannt 'Parse-Forest', vor. Um diese Struktur effizient zu gestalten, werden zwei spezielle Techniken benutzt:

- 'Subtree sharing'
- 'Local ambiguity packing'

Es ist oft der Fall, dass bei verschiedenen Ableitungen dennoch gleiche Substrukturen vorhanden sind. Betrachte das Beispiel 'I saw Fred in the car'. Zwei mögliche syntaktische Analysen sind folgende:

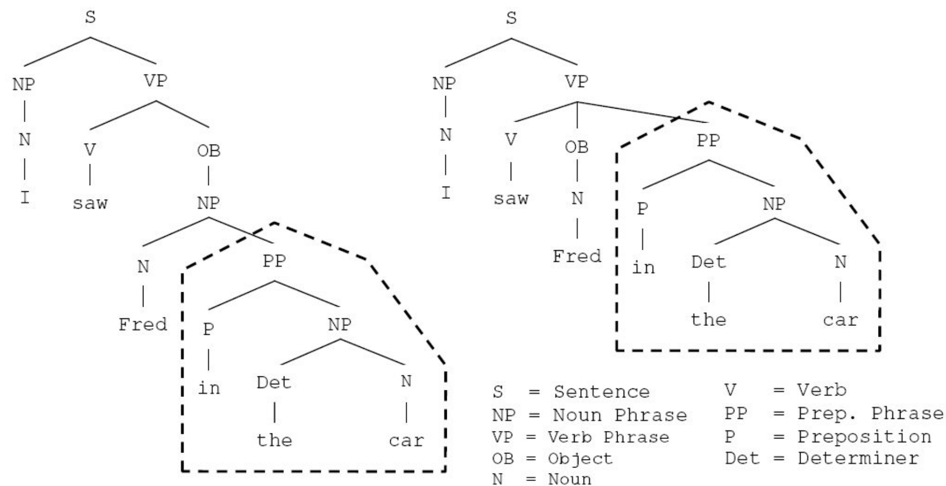



Abbildung 3: Identische Strukturen im Syntaxbaum - 'subtree-sharing'

Die erste Interpretation legt die Betonung auf 'Fred', die zweite auf 'saw'. Trotzdem ist der Präpositionalsatz beider Interpretationen gleich, was man für die Methode des 'subtree-sharings' ausnutzen kann, wo identische Bestandteile durch ein einziges Objekt repräsentiert werden.

Des Weiteren kann man in Abbildung 3 und 4 sehen, dass die oberste Ebene auch in beiden Ableitungsbäumen identisch ist:

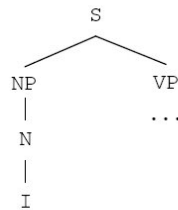



Abbildung 4: Identische Strukturen im Syntaxbaum - 'local ambiguity packing'

Diese Gleichheit kann nun für das sogenannte 'local ambiguity packing' ausgenutzt wedren, wo verschiedene Söhne, die den gleichen Vater besitzen, in einen Knoten gepackt werden. Dieser Knoten kann nun wie ein normaler Knoten in den 'Forest' eingebunden werden.

Durch Benutzung der beiden oben beschriebenen Techniken kann der Satz 'I saw Fred in the car' wie folgt repräsentiert werden:

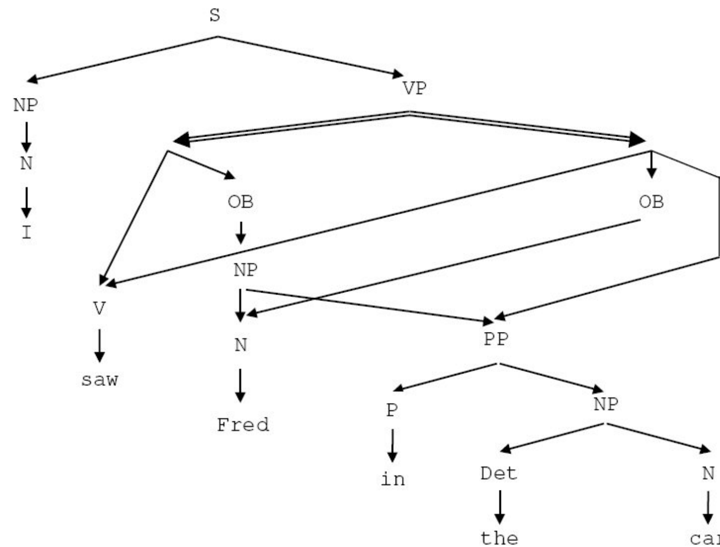



Abbildung 5: Der duch 'subtree-sharing & 'local ambiguity packing' entstandene 'Forest'

Die Pfeile mit den Doppelstrichen deuten an, dass es sich bei dem Knoten mit dem Label 'VP' um einen gepackten Knoten handelt, der zwei Analysen repräsentiert.
Gemeinsam benutzte Teilbäume sind diejenigen, auf die mehr als ein Pfeil zeigt. Wie man zudem hier sieht, handelt es sich bei dem 'Forest' um einen gerichteten, azyklischen Graphen (directed, acyclic graph - DAG).

## 2.2 Der 'Graph-Structured-Stack'

Der 'Graph-Structered-Stack' (GSS) ist die fundamentale Datenstruktur in Tomita's Algorithmus. Er kann als Parse-Stack aufgefasst werden, der die üblichen Operationen shift und reduce bereitsstellt, allerdings basiert er auf einem Graph statt auf einer Liste. Der Graph ist ein gerichteter, azyklischer Graph mit Knoten, die den jeweiligen Zustand im Parse-Vorgang darstellen und Kanten, die diese miteinander verbinden. Jeder Knoten beinhaltet hierbei eine Nummer und jede Kante beinhaltet eine 'Forest'-Struktur, die die bis hierher abgewickelte Analyse der Eingabe repräsentiert.
Im Unterschied zu den standard shift/reduce Operationen, die auf dem obersten Element des Stack ausgeführt werden, werden die Operationen hier auf einem bestimmten Knoten v ausgeführt, der ein Element an oberster Stelle des Stack repräsentiert, denn es können mehrere Elemente zur gleichen Zeit an oberster Stelle stehen. Die Operationen sind nun wie folgt aufgebaut:

- shift(v): Eine neue Kante wird erzeugt, die von v abgeht und einen 'Forest' beinhaltet, der die Analyse des geshifteten Tokens wiedergibt (ein einzelnes Blatt) und in einem neuen Knoten endet, der mit der Zustandsnummer versehen wird, so wie es in der Action-Tabelle angegeben ist.

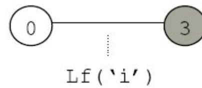
- reduce(v): Kanten, die in einer Distanz von n von Vorgängern abgehen, wobei n die Anzahl der Symbole auf der rechten Regelseite ist, werden aus dem Graph entfernt, ebenso wie die nun überflüssigen Knoten. Ausgehend von jedem Ursprungsknoten wird eine Kante angelegt, welche eine neue Analyse umfasst, die wiederum aus einem 'Forest'-Knoten besteht. Dieser ist beschriftet mit der linken Regelseite derjenigen Regel, deren Nachfolger 'Forests' sind, welche an den Kanten hängen, die zu v führen.

'Subtree-sharing' ist in der Weise in den GSS eingebunden, als dass zwei Kanten, die identische Analysen von Teilen der Eingabe beherbergen, in einem 'Forest' gespeichert werden und die jeweiligen Kanten sich diesen teilen.
'Local ambiguity packing' wird dann benutzt, wenn mehr als eine Kante zwischen zwei Knoten existiert. Dies stellt eine Stelle dar, in der der Parser sich wegen eines Konfliktes in der Parse-Tabelle aufgeteilt hat und anschliessend die verzweigten Analysen in einen oberen Knoten reduziert. Sollte dies der Fall sein, dann können die Analysen in einen Knoten gepackt werden und die einzelnen Kanten mit einer Kante ausgetauscht werden, die den gepackten 'Forest' enthält.

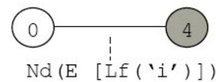
Der Parser wird mit einem einzigen Knoten initialisiert, der den Startzustand darstellt; normalerweise '0'. Der Parse-Vorgang wird dann entsprechend der Parse-Tabelle fortgeführt. Tritt ein Konflikt auf, werden alle Aktionen aus der Tabelle auf dem Knoten ausgeführt, von der Konzeption her geschieht dies gleichzeitig. Treffen die auseinander gedriftteten Analysen wieder zur gleichen Zeit (nach Abarbeitung der selben Anzahl von Symbolen) zusammen, so werden sie wieder in einem Knoten vereint.
Zur Verdeutlichung wollen wir ein Beispiel aus [Med 02] angeben, ausgehend von unserer simplen Grammatik 'E $\rightarrow$ E +E | i' und der schon zuvor betrachteten Eingabe 'i + i + i'. Siehe hierzu auch die zugehörige Parse-Tabelle auf Seite 4.

Der GSS wird mit dem Startzustand initialisiert (die Knoten, die im Stack oben liegen, sind grau markiert).

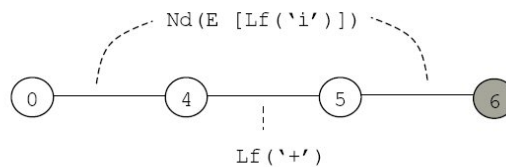
Das erste Symbol der Eingabe ist 'i' und die zugehörige Aktion 'shift3'. Dementsprechend wird eine neue Kante erstellt, die den Startknoten mit dem neuen Knoten des Folgezustands, hier 3, verbindet und an der ein Blatt mit dem Label des gelesenen Symbols hängt.




Das nächste Symbol ist '+' und die zugehörige Aktion im Zustand 3 lautet 'reduce2'. Die Reduktion wird ausgeführt und der GSS wird wie folgt gebildet:

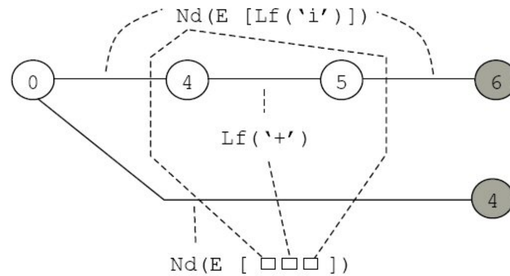



Der Parse läuft bis zum vierten Symbol deterministsich fort. Hier sieht der GSS nun so aus:

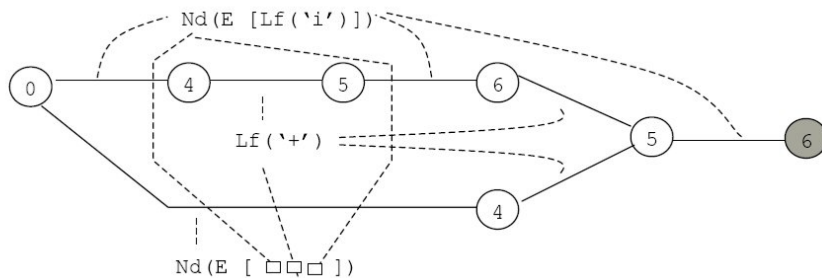



Es fällt auf, dass wir das erste 'i' nicht vom zweiten unterscheiden, denn es ist nur einmalig als 'shared subtree' zwischen den Kanten (0,4) und (5,6) gespeichert.

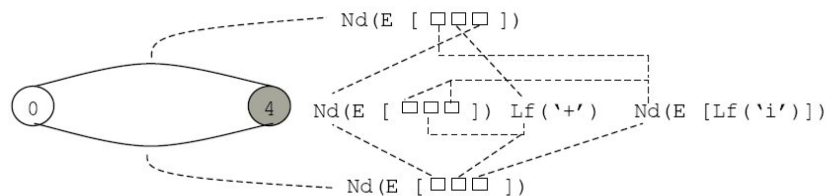
Die Position eines Lexems in der Eingabe wird nicht explizit gespeichert, sie kann jedoch bei Bedarf abgefragt werden.
Die Aktion für ein '+' im Zustand 6 ist 'shift5/reduce1'. Es besteht also ein Konflikt. Es werden nun zuerst alle shifts ausgeführt, dann die Reduktionen, wodurch der GSS die folgende Gestalt erhält:



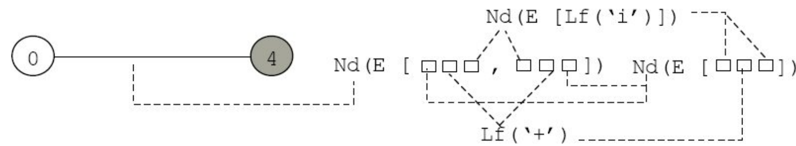

Es liegen uns nun zwei grau eingefärbte Zustände vor, die demnach auch zwei oberste Knoten auf dem Stack darstellen. Der 'Forest', der entlang der unteren Kante gebildet wird, nutzt hier bereits auch das 'subtree sharing'. Die shift Aktion auf beiden oberen Knoten ist 'shift5', was es erlaubt beide Analysen in einen Knoten zu überführen. Das letzte 'i' wird geshiftet und anschliessend zu 'E' reduziert um den GSS wie folgt zu gestalten:




Eine Reduktion auf dem oberen Knoten (Zustand 6) und anschliessend eine Reduktion auf dem neu erzeugten Knoten (wiederum Zustand 6) überführt den GSS in den nachstehenden Zustand:



Zu beachten ist hier, dass es nun zwei neue Kanten gibt, die beide vom gleichen Knoten abzweigen und in einem gemeinsamen Knoten wieder zusammenkommen, und dennoch separate Analysen für die bisherige Eingabe beinhalten. Diese Analysen können nun zu einer einzigen Analyse unter der Kategorie 'E' zusammengefasst werden. Die nächste Aktion ist letztendlich ein accept auf dem obersten Knoten und der Parse terminiert mit dem GSS in unten abgebildeter Beschaffenheit:




Ist der Parse erfolgreich, so wird der der GSS immer nur aus dem Startzustand, dem Endzustand und einer verbindenden Kante zwischen ihnen bestehen. An dieser Kante hängt nun der 'Parse-Forest', der alle gültigen Parse-Möglichkeiten (hier zwei Möglichkeiten) beinhaltet und als Resultat eines erfolgreichen Parse zurückgegeben wird.

# 3 Implementation

Wir wollen uns nun der Implementation des oben beschriebenen Verfahrens zuwenden und betrachten hierfür erstmal, wie das oben aufgeführte Beispiel in einer funktionalen Umgebung aussieht. Die nachfolgende Diskussion ist insbesondere durch [Med 02] und [Cal 05] motiviert.
Wie bereits besprochen, ist der 'Forest' technisch gesehen ein DAG. Eine solche Struktur kann als eine Liste von Knoten repräsentiert werden, wobei jeder Knoten auf einen eindeutigen Schlüsselwert abgebildet wird. Innerhalb jedes Knotens ist eine Liste der Schlüsselwerte seiner Nachfolger gespeichert.
In einem 'Parse-Forest' ist jeder Knoten mit einem Symbol aus der Grammatik versehen. Knoten mit einem oder mehr Nachfolgern sind mit Nonterminalen, Knoten ohne Nachfolger, also Blätter, mit Terminalen versehen. Ein gepackter Knoten enthält eine Liste von zwei oder mehr alternativen Analysen, jeder mit seiner eigenen Liste von Nachfolgerknoten. Der oberste Knoten im 'Parse-Forest' muss markiert sein, um ihn von den anderen zu unterscheiden.
Wir können nun den 'Parse-Forest' als eine injektive Abbildung von Schlüsselwerten zu Knoten auffassen. Es bleibt zu beachten, dass wenn der GSS erstmal erzeugt wurde, es möglich sein muss, einen Knoten wieder zu besuchen (vorallem für den Prozess des Packens) und dass Knoten im GSS erheblichen Veränderungen im Laufe des Parsens unterliegen. Würden wir nun eine solche Abbildung wie eben erläutert benutzen, würde dies viel Aufwand für das Auffinden, Entfernen und Wiederaufbauen erfordern. Wie auch oben gesehen, treten einige redundante Bereiche auf. Dies benötigt zusätzliche Berechnungen, denen wir mit einer funktionellen Sprache wie Haskell entgegen wirken möchten, da Haskell eine eigene 'garbage collection' besitzt.
Ljunglöf hat deshalb vorgeschlagen, den GSS als eine Sammlung von Bäumen zu präsentieren, wo jeder Baum einen Parse-Stack darstellt und einen eindeutigen oberen Knoten besitzt [Lju 02]. Wenn ein Konflikt auftritt, wird der Stack in zwei identische Stacks aufgeteilt und beide Operationen gleichzeitig ausgeführt. Auch wenn es so erscheint, als ob zwei Stacks erzeugt wurden, wird die identische Struktur beider Stacks intern als ein Objekt gespeichert, wie man der nachfolgenden Abbildung entnehmen kann:

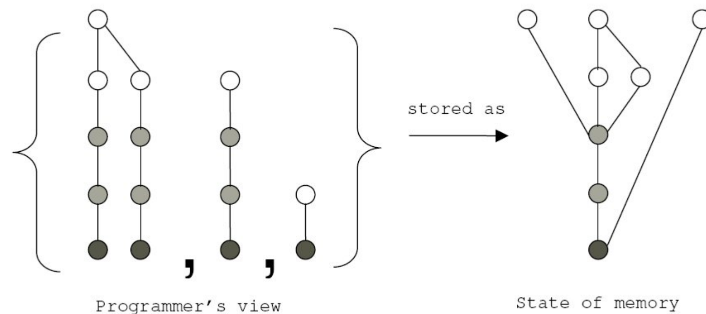



Abbildung 6: Äussere und innere Sicht auf den Stack

In Haskell werden intern alle Objekte mit Refernzen gespeichert, weshalb bei einer Duplikation eines Stack sein Zeiger kopiert wird, und der Stack selbst als einziges Objekt behalten wird. Bei dieser Art der Präsentation des GSS brauchen wir uns nun nicht mehr um die explizite Form des Graphen kümmern. Viele der Berechnungen können auf Bäumen aufsetzten, eine für die funktionale Programmierung optimale Datenstruktur. Desweiteren kann die Sammlung der Bäume in einer Liste erfolgen, was die Benutzung bewährter funktionaler Techniken auf Listen erlaubt.

## 3.1 Der 'Parse-Forest'

Zuerst wollen wir uns nochmal mit der Modellierung der Operationen in einer funktionalen Umgebung des GSS beschäftigen. Der oberste Knoten eines jeden Baumes in der Sammlung korresponiert also zu einem oberen Knoten im Stack im traditionellen GSS Modell. Dementsprechend führen wir die shift und reduce Operationen auf den obersten Knoten der Bäume aus, so wie im traditionellen Ansatz von Tomita. Wird ein Zustand gleichzeitig von zwei Kanten, ausgehend von verschiedenen Knoten erreicht, so werden ihre obersten Knoten in einem vereinigt. Das 'subtree sharing' wird durch Knoten-Indices realisiert. Da der 'Parse-Forest' als eine injektive Abbildung von Schlüsselwerten auf Knoten im 'Forest' präsentiert wird, dienen die Indexwerte als Zeiger auf jede einzelne Subanalyse und repräsentieren somit das gemeinsame Nutzen verschiedener Sektionen im GSS.
'Local ambiguity' tritt auf, wenn der Parser einen Zustand erreicht, indem zwei Bäume bis auf ihre oberen Analysen identisch sind. Somit können die oberen Analysen zusammengepackt und die Bäume kombiniert werden:

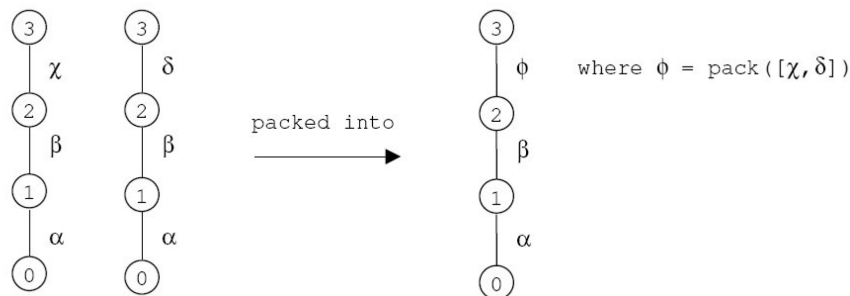



Abbildung 7: 'Local ambiguity packing'

Für die eigentliche Implementiereung unseres funktionalen Modells bedarf es nun zweier Komponenten:

- einer Datenstruktur die die 'Forest'-Knoten repräsentiert (gespeichert als Elemente in einer Map)
- einer Datenstruktur, die die Map selbst darstellt

Wir implementieren die Datenstruktur für den 'Forest' als einen algebraischen Typ in Haskell:

data ForestNode a = FNode a [[Int]]

Jeder Knoten innerhalb des Graphen erhält einen eindeutigen Bezeichner, die 'ForestID'. Die Graphstruktur des 'Forests' wird realisiert, indem eine Abbildung von 'ForestIDs' nach 'ForestNodes' aufgebaut wird. Eine 'ForestNode' v beinhaltet die Liste derjenigen Knoten, die über Kanten mit v verbunden sind. Diese werden über ihre 'ForestIDs' identifiziert. Die Technik des 'local ambiguity packing' macht es nun notwendig, die Möglichkeit vorzusehen, mehrere mögliche Analysen in einer Forest Node zu kodieren. Eine Node enthält somit nicht einfach eine Liste von verbundenen Knoten sondern eine Liste solcher Listen, deren Elemente jeweils einer möglichen Analyse zugehörig sind.
Ints dienen als eindeutige Bezeichner:

type FID = Int

Der eigentliche 'Forest' wird mit Hilfe der Haskell Datenstruktur 'FiniteMap' realisiert:

type Forest = FiniteMap FID (ForestNode GSymbol)

wobei es sich bei 'GSymbol' um ein Symbol der Grammatik, also ein Terminal oder ein Nonterminal handelt:

data GSymbol = Terminale und Nonterminale

## 3.2 Die Datenstruktur für den 'Graph-Structered-Graph'

Die Umsetzung der Kollektion von Bäumen geschieht nun mittels eines abstrakten Datentyps, genannt TStack, der folgenden Konstruktor besitzt:

data TStack a = TS Int [(a, TStack a)]

Der TStack besteht aus einem oberen Zustand und besitzt eine Reihe von Nachfolger-TStacks. Diese Söhne sind mit dem oberen Zustand über Kanten verbunden, an welchen der 'Forest' anhängt. Der Index des oberen Zustands stellt den ersten Konstruktorparameter dar, der zweite Parameter wird benutzt um die Nachfolger-TStacks zu charakterisieren, diese werden als eine Liste von Paaren übergeben. Die zweite Komponente des Paars gibt den entsprechenden Nachfolger-TStack an. In der ersten Komponente findet sich die Information, welcher 'Forest'-Knoten mit der korrespondierenden Kante verbunden ist.

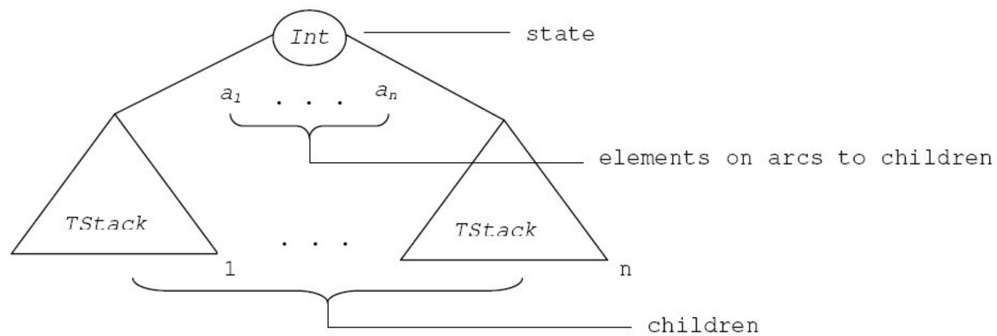



Abbildung 8: Aufbau des TStack

Eine TStack Instanz repräsentiert einen Ast des GSS. Der gesamte GSS wird durch eine Liste von TStacks kodiert. Während des Parse-Vorgangs werden die TStacks mittels einer Reihe von Operationen aufgebaut. Diese sind definiert wie folgt:

```
top :: TStack a -> Int
top (TS t _) = t
```

'top' liefert den obersten Zustand zurück.

```
push :: a -> Int -> TStack a -> TStack a
push f s tstk = TS s [(f, tstk)]
```

'push' fügt einen neuen obersten Zustand ein, der über eine Kante mit dem bisherigen TStack verbunden ist, an der sich der übergebene 'Forest'-Knoten befindet.

```
pop :: Int -> TStack a -> [([a],TStack a)]
pop 0 ts = [([],ts)]
pop n (TS _ ch) = [ (xs ++ [x] , stk')|
                  (x,stk) <- ch, let rec = pop (n-1) stk,
                  (xs,stk') <- rec ]
```

'pop' ermöglicht es in der TStack Struktur rekursiv bis zu einer bestimmten Tiefe abzusteigen und die innerhalb dieser Tiefe abgelegten Teil-TStacks zurückzuliefern. Hierbei werden auch die während des Abstiegs gesammelten 'Forest'-Knoten als Liste zurückgegeben. Man erhält somit eine Liste von Paaren, wobei sich in der ersten Komponente eine Liste von gesammelten 'Forest'-Knoten befindet, die zweite Komponente enthält den der vorgegebenen Tiefe zugehörigen TStack.

```
merge :: [TStack a] -> [TStack a]
merge stks
 = [ TS st h id ch
 | st <- nub (map top stks)
 , let ch = concat [ x | TS st' _ _ x <- stks , st==st' ]
 h = foldl1 max [ x | TS st' x _ _ <- stks , st==st' ]
 id = head [ x | TS st' _ x _ <- stks , st==st' ]
 ]
```

'merge' fasst diejenigen TStacks aus der Eingabeliste zusammen, welche den gleichen oberen Zustand aufweisen. Hierzu werden, ausgehend von diesem, Kanten angelegt, welche den oberen Zustand mit den jeweiligen verbleibenden unteren TStack Strukturen der zusammengefassten Stacks verbinden.

# 4 Integration in Happy

Nach der Implementation des Algorithmus von Tomita in Haskell bedarf es noch der Integration in Happy wie beschrieben in [Cal 05]. Zu dem bisherigen Code von Happy wurde ein neues Modul hinzugefügt: ProduceGLRCode. Dieses exportiert nur eine Methode: produceGLRParser. Diese wird in dem modifizierten Main Modul aufgerufen, sollte ein entsprechendes Flag bei der Ausführung gesetzt sein. Auch bei Verwendung des GLR-Parsers werden die von Happy erzeugten LALR-Parse-Tabellen unverändert genutzt und lediglich die daraus resultierenden Aktionen abgeändert, wie oben bereits besprochen. Durch ein hinzugefügtes Flag kann der Benutzer auswählen, welchen Parser-Generator er benutzen möchte; den Standard-LR-Parser oder durch die Angabe von dem Flag --glr den Parser-Generator für mehrdeutige Grammatiken. Die restliche Funktionalität von Happy bleibt wie gehabt erhalten.

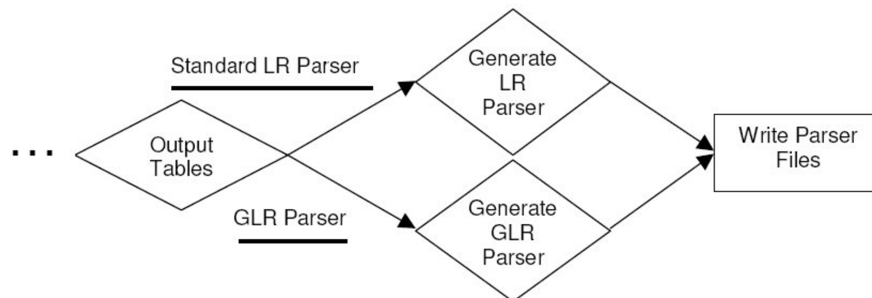



Abbildung 9: Integration der GLR Funktionalität in Happy

Weiterhin bleibt auch die Transparenz erhalten; der Benutzer kann sämtliche in Happy vorhandenen Optionen nutzen, wie zum Beispiel benutzredefinierte Tokens, Einbettung von eigenen Modulen und Ausgabe von Informationen über die Parse-Tabelle.

# 5 Schlußbemerkung

Wir haben uns mit der Problematik des Parsens mehrdeutiger Grammatiken beschäftigt und dabei Tomita's Algorithmus kennengelernt. Er basiert auf der Idee anstelle eines Stacks mehrere parallele Stacks für die verschiedenen möglichen Analysen aufzusetzen. Dieses Verfahren wurde von Medlock und Callaghan aufgegriffen und von ihnen in die funktionale Programmiersprache Haskell übernommen. Es gibt weitere Ansätze zur Lösung dieser Problematik dennoch entschieden sie sich für Tomita's Algorithmus aufgrund von Effizienzkriterien.
Das hier dargestellte Verfahren bezieht sich in erster Linie auf [Med 02]. Es wurde von Callaghan noch weiterentwickelt, vorallem in der Hinsicht auf das Hinzufügen semantischer Informationen um so komplexe mehrdeutige Grammatiken, wie sie beispielsweise zur Beschreibung natürlicher Sprache auftreten, sinnvoll zu parsen [Cal 05].
Die Integration in das Standardpaket des Parsergenerators Happy belegt die Praktikabilität dieses Ansatzes. Ab Happy Version 1.15 ist das eben genannte erweiterte Verfahren im Happy Code integriert und muss nicht als zusätzliches Modul eingebunden werden.

# 6 Literatur


[Cal 05] Callaghan, P. (2005): Ambiguous parsing with Happy, Journal of Functional Programming

[Cal 04] Callaghan, P. (2004): The Happy-GLR Website, http://www.dur.ac.uk/p.c.callaghan/happy-glr/

[Cal 97] Callaghan P. (1997): An Evaluation of LOLITA and related Natural Language Processing Systems, University of Durham

[Lju 02] Ljunglöf P. (2002): Chapter 6, Licentiate Thesis (final draft)

[Med 02] Medlock, B. (2002): A tool for generalised LR Parsing in Haskell, Single Honours CS Project Report, Department of Computer Science, University of Durham

[Tom 85] Tomita M. (1985): An efficient context-free parsing algorithm for natural languages, In proceedings of the 9th International Joint Conference on Artificial Intelligence, Los Angeles, CA.