Diplomarbeit

Entwurf und Implementierung eines Systems zur Erzeugung syntaxgesteuerter Editoren

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik
Institut für Theoretische Informatik
Grundlagen der Programmierung

von

Enrico Bormann
geb. am 19. Juli 1974
in Freital / Sachsen

Betreuender Hochschullehrer: Prof. Dr.-Ing. habil. Heiko Vogler
Betreuer: Dr. Armin Kühnemann

Dresden, im März 2000

Danksagung

Ich möchte mich an dieser Stelle bei Gott und all den Menschen bedanken, die mir die Erstellung dieser Diplomarbeit ermöglicht haben. Folgenden Personen gilt dabei mein ganz besonderer Dank:
Meinem betreuenden Hochschullehrer Prof. Dr.-Ing. habil. Heiko Vogler danke ich für das interessante und meinen Geist herausvordernde Thema dieser Arbeit. Bei Dr. Armin Kühnemann möchte ich mich besonders für die wertvolle fachliche und menschliche Unterstützung bedanken, welche er mir als mein Betreuer geleistet hat. Herrn Sean Eikenberg danke ich für seine hilfreiche Mitarbeit (im Rahmen seiner Tätigkeit als studentische Hilfskraft) bei der Auswahl geeigneter Softwaretools für die Erstellung einer graphischen Oberfläche des zu entwerfenden Systems.
Meinen Freunden Rajko Rebs, Andrea Sackmann, Stefan Zschammer, Heike Al-Roussan, Christian Sylkin, Sven Leibiger, Michael Hupfer und Gabi Veit möchte ich sehr dafür danken, daß sie mit mir nicht nur die Freude, sondern auch das Leid geteilt haben, welches eine Diplomarbeit mit sich bringt. Zum Schluß (aber alles andere als zuletzt) gilt mein Dank Rebekka Homilius, dem Engel, der dafür Sorge trug, daß sich mein Herz stets mit Liebe gefüllt der Arbeit widmen konnte.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die hier vorliegende Arbeit beschäftigt sich mit einem System zur Erzeugung syntaxgesteuerter Editoren.

Es handelt sich dabei um einen Generator für Editoren, welche speziell auf eine gewünschte Sprache zugeschnitten sind und sich somit in der Lage befinden, ihrem Nutzer bei seiner Arbeit Unterstützung zu leisten. Daher nennen wir solche Editoren in dieser Arbeit auch sprachbasierte Editoren. Sprachbasierte Editoren können durch ihre besonderen Fähigkeiten beträchtlich dazu beitragen, die heutzutage aktuelle Aufgabe zu verwirklichen, große Software-Systeme zu entwickeln und auch zu warten.
Als Beispiel seien hier sprachbasierte Editoren für die Programmiersprachen C++ und Java genannt. Weiterhin können mit einem derartigen Generator auch WYSIWYG-Editoren für Text oder auch mathematische Formeln erstellt werden, die sich als große Hilfe im Feld der Dokumenterstellung erweisen. Als drittes Anwendungsfeld sei die Generierung von Editoren genannt, welche Beweise in Sprachen der Logik führen können.

Das Wissen über die Sprache, auf welcher ein solcher Editor basiert, wird von ihm verwendet, um seinem Nutzer in vielfacher Hinsicht die Arbeit zu erleichtern. Hierbei kommen Konzepte zum Einsatz, auf die wir nun kurz eingehen wollen.
Zuerst sei das Konzept des strukturellen Editierens genannt. Hierbei nutzt der Editor die Kenntnis der kontextfreien Grammatik der Sprache, für die er konzipiert wurde. Das mit dem Editor editierte Konstrukt wird als Hierarchie geschachtelter Komponenten betrachtet und intern als Syntaxbaum repräsentiert. Die nach außen hin sichtbaren Komponenten sind Schablonen und Platzhalter. Der Nutzer des Editors kann nun Schablonen in passende Platzhalter einfügen oder geeignete Schablonen durch bestimmte andere ersetzen. Dadurch ist eine Top-Down-Entwicklung des zu editierenden Konstrukts, z.B. eines C-Programmes bei einem C-basiertem Editor, möglich. Diese sehr disziplinierte Art der Programmierung bewahrt den Nutzer vor dem Begehen syntaktischer Fehler und läßt ihn auf einem hohen Abstraktionslevel arbeiten. Die Abbildung 1.1 zeigt eine strukturelle Transformation bei einem Editor für eine beispielhafte imperative Programmiersprache.
Strukturelles und textuelles Editieren schließen sich nicht gegenseitig aus, sondern ihre Vorteile ergänzen sich. Daher ist es sinnvoll und auch möglich, beide Herangehensweisen in den zu generierenden Editoren zu vereinen. Das nächste zu nennende Konzept ist

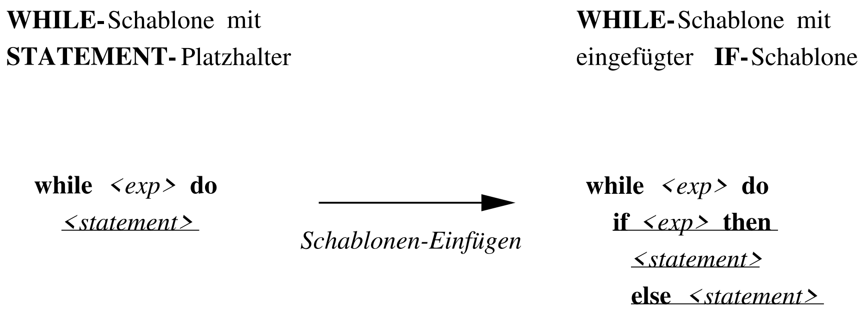

Abbildung 1.1: Strukturelles Editieren

das der sofortigen Berechnung. Berechnung steht hier u.a. für das Abtesten von semantischen Nebenbedingungen der Sprache, auf welcher der entsprechende Editor basiert. Jede Änderung des editierten Konstrukts hat einen sofortigen Effekt, z.B. daß der Nutzer umgehend Informationen über aufgetretene Fehler erhält. Dies erlaubt Testen und Fehlersuche noch während des Editierens, wie auch in Abbildung 1.2 dargestellt. Es fallen also die sonst notwendige Compilationsphase und die damit verbundenden Verzögerungen weg. Zwischen den Programmodifikationen des Programmierers (Nutzer des sprachbasierten Editors) sind aber nicht nur semantische Analysen für den Fehlerreport denkbar, sondern auch Formatierungen oder Codegenerierungen z.B. bei WYSIWYG-Editoren. Zur

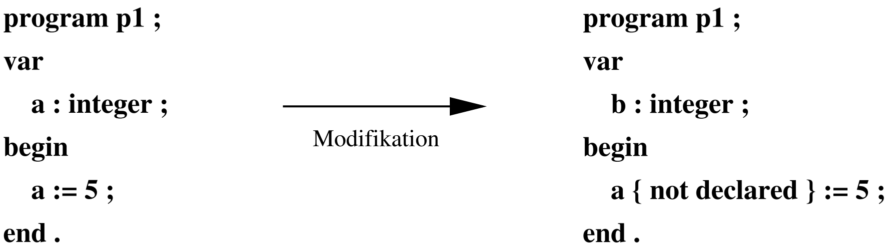

Abbildung 1.2: Sofortige Fehlerausgabe nach Änderung der Variablen in der Deklaration

Realisierung der notwendigen semantischen Analysen wird das Konzept der Makroattributgrammatiken verwendet. Diese Grammatiken eignen sich hervorragend als Beschreibungsmittel im Bereich der formalen Sprachen der Informatik, u.a. auch zur Beschreibung von kontextabhängigen Nebenbedingungen. Hervorgegangen sind sie aus den Attribut-Grammatiken [Knu68] durch Hinzunahme von Kontextparametern in den Attributen. Der interessierte Leser sei auf [KV94] verwiesen. Mit Hilfe der Konzepte der Attribut- bzw. Makroattributgrammatiken sind die von uns erzeugten Editoren in der Lage, eine selbstständige Attributauswertung auszuführen. Dabei liegt das editierte Konstrukt intern als attributierter Syntaxbaum vor. Somit können durch Auswertung der Attribut- bzw. Funktionswerte Fehlermeldungen erzeugt werden, um den Editornutzer über semantische Fehler zu informieren. Makroattributgrammatiken können bei unseren Editoren aber auch

für Übersetzungsaufgaben eingesetzt werden.
Durch die Verbindung des Konzeptes der Makroattributgrammatiken mit dem Konzept der sofortigen Berechnung kommt es relativ häufig zu Attributauswertungen. Diese sind um so aufwendiger, je größer das mit einem sprachbasierten Editor editierte Konstrukt und damit auch der interne attributierte Syntaxbaum ist. Daher ist es sinnvoll, das Konzept der inkrementellen Berechnung zur Anwendung zu bringen (s. [Kas80] u. [DRT81]). Dieses Konzept findet sich u.a. auch in [AJKV98] als inkrementeller Attributauswerter wieder. Dabei wird der Fakt berücksichtigt, daß sich aus den Attributänderungen selten Änderungen aller anderen Attribute ergeben, sondern sich der betroffene Bereich, in dem Attribute neu berechnet werden müssen, meistens relativ klein hält. Durch die Abbildung 1.3 sei das Prinzip der inkrementellen Berechnung kurz verdeutlicht. Mit diesen Fähigkeiten aus-

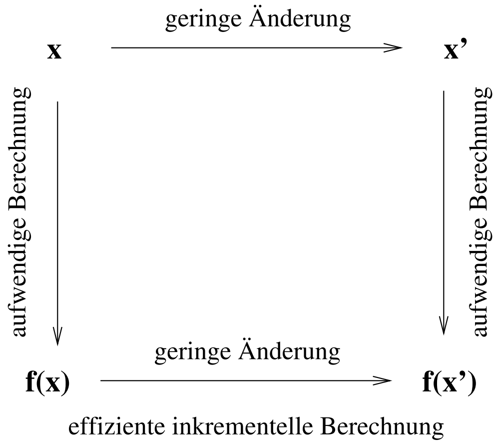

Abbildung 1.3: Der Wert f(x') läßt sich in vielen Fällen effizienter inkrementell aus f(x) berechnen als durch Anwendung von f auf x'

gestattet, können die besprochenen Editoren bei den verschiedenen Herangehensweisen der Softwareentwicklung nützlich sein. Für die stufenweise Programmentwicklung, bei der zuerst ein funktionsfähiges Programm erstellt und dieses danach durch korrektheitserhaltende Transformationen in eine effiziente Programmierung umgewandelt wird, bietet sich ein sprachbasierter Editor an, solche strukturelle Programmtransformationen selbst vorzunehmen. Bei der Herangehensweise der Softwareentwicklung, daß ein Programm Hand in Hand mit dem Beweis seiner Spezifikationstreue erstellt werden soll, kommen das Konzept der sofortigen Berechnung und die Anwendung der semantischen Analyse durch das Konzept der Makroattributgrammatiken zum Tragen. Hierdurch sind Korrektheitsbeweise des erstellten Programms schon während der Programmierung möglich.

Sprachbasierte Editoren finden heute schon weltweit ihren Einsatz. Die meisten sind aber 'per Hand' für ihre spezielle Anwendung erstellt worden. Neben den speziellen Anforderungen solcher Editoren, in denen sie sich untereinander unterscheiden, existieren aber auch Eigenschaften der Editoren, welche unabhängig vom konkreten Einsatzgebiet sind. Solche Anforderungen sind z.B. Kommandos zum Navigieren durch das zu editierende Konstrukt oder das Speichern und Laden von Konstrukten unter Verwendung eines Dateisystems.

Weiterhin ist stets eine Nutzerschnittstelle, also eine geeignete Oberfläche, notwendig. Hierbei müssen Editierfunktionen angeboten werden, wie das Kopieren, Einfügen und Löschen von Text und weiterhin die Verbindung vom puren Texteditieren zur Anpassung der internen Repräsentation des editierten Konstrukts. Zusammenfassend sei gesagt, daß eine Vielzahl von sprachunabhängigen Editorfunktionen existieren und es sich daher anbietet, einen Generator für sprachbasierte Editoren zu konzipieren, welcher die Erzeugung solcher in jedem Fall benötigter Teile eines Editors automatisiert. Mit Hilfe eines derartigen Generators ist der Entwickler eines sprachbasierten Editors in der Position, sich nur um die sprachabhängigen Eigenschaften seines Editors kümmern zu müssen. Das System Synthesizer Generator [RT89] ist ein solcher, sich schon im Einsatz befindender Generator.

Der in dieser Arbeit vorzustellende Generator bedient sich als Eingabesprache der Sprache Haskell$^+$. Diese baut auf der funktionalen Sprache Haskell ([FHP97],[HP97]) auf und erweitert sie um das Konzept der Makroattributgrammatiken. Die Sprache Haskell$^+$ ist eine Obermenge der Sprache Haskell, d.h. jedes Haskell-Programm ist auch ein Haskell$^+$-Programm. Ein Hauptziel der Entwicklung von Haskell$^+$ [Les99] war die Erstellung einer Eingabesprache für ein System zur Erzeugung von sprachbasierten Editoren. Die Beschreibung der abstrakten Syntax der Sprache, auf welcher der zu erzeugende Editor basieren soll, erfolgt wie üblich in einer funktionalen Sprache durch algebraische Datentypen. Im Gegensatz zu Haskell enthält Haskell$^+$ zusätzlich Sprachkonstrukte zur Beschreibung eines Displayformats, welches für die textuelle Darstellung des mit dem Editor editierten Konstrukts notwendig ist. Ebenfalls existieren Sprachkonstrukte zur Festlegung von strukturellen Transformationsregeln, welche später bei der Anwendung des Editors zum Tragen kommen sollen. Die schon erwähnte Möglichkeit, Makroattributgrammatiken zu spezifizieren, wird zur Beschreibung von semantischen Nebenbedingungen verwendet sowie zur späteren Analyse derselben.

Ziel dieser Arbeit ist zum einem die Erstellung eines Konzeptes für einen Generator für sprachbasierte Editoren, welcher aus einer Haskell$^+$-Spezifikation einer Sprache L einen sprachbasierten Editor für L erzeugt. Dabei ist zu prüfen, ob die Sprachkonstrukte, welche Haskell$^+$ anbietet, geeignet und ausreichend dafür sind, um Haskell$^+$ als Eingabesprache für den besprochenen Generator zu verwenden. Gegebenenfalls sind Vorschläge zur Verbesserung von Interesse. Der zu konzipierende Generator soll an das System Synthesizer Generator angelehnt sein. Außerdem sind verschiedene Anwendungsszenarien des Generators und der durch ihn erzeugten Editoren zu beleuchten, d.h. es ist die Frage zu beantworten, welche Personen welche Möglichkeiten haben, Nutzen aus dem System zu ziehen.
Weiterhin ist eine erste Version des Generators zu implementieren und die dabei gemachten Einschränkungen zu beschreiben bzw. zu begründen. Hauptaugenmerk liegt hierbei auf der Verwendung der abstrakten Syntax, der Umsetzung des Displayformats, der strukturellen Transformationsregeln, sowie einer auf Zahlen und Bezeichner reduzierten textuellen Eingabemöglichkeit. Bei den erzeugten Editoren soll eine fensterorientierte Bedienoberfläche dem späteren Nutzer zur Verfügung stehen.
Es wird auf die zukünftige Erweiterbarkeit des Systems Wert gelegt. Auf die Kompatibilität zu bereits existierenden und auf diesem Bereich nützlichen Implementierungs-, Analyse- und Transformationsprogrammen für Haskell$^+$ ist ebenfalls zu achten. Natürlich folgt auch eine konkrete Anleitung zur Arbeit mit dieser ersten implementierten Version.

Nicht Inhalt der Arbeit ist eine Einführung der Sprachen Haskell$^+$ oder Haskell. Dazu wird auf [Les99] bzw. auf ([FHP97],[HP97]) verwiesen. Auf das System Synthesizer Generator wird ebenfalls nicht konkret eingegangen. Weiterhin wird auf eine Einbindung des Konzeptes der inkrementellen Attributauswertung bei der Implementation verzichtet.

Die vorliegende Arbeit ist wie folgt aufgebaut:
In Kapitel 1 geben wir eine Einführung in die Thematik dieser Arbeit. Das Kapitel 2 beschreibt die formalen Grundlagen, welche wir in späteren Kapiteln benötigen (u.a. die formale Einführung von Makroattributgrammatiken). In Kapitel 3 erläutern wir am Beispiel einer Editiersitzung die Benutzung eines auf einer einfachen imperativen Sprache P basierenden Editors, welcher mit dem in dieser Arbeit behandelten System generiert worden ist. Kapitel 4 beschäftigt sich mit dem Entwurf des Systems zur Erzeugung von sprachbasierten Editoren. In Kapitel 5 beschreiben wir, wie ein sprachbasierter Editor spezifiziert werden muß, damit er von diesem System generiert werden kann. Dabei erläutern wir auch konkret die Spezifikation des auf der Beispielsprache P basierenden Editors aus Kapitel 3. Auf die Implementierung des Systems zur Erzeugung von sprachbasierten Editoren wird in Kapitel 6 eingegangen. Das Kapitel 7 enthält eine kurze Diskussion der Ergebnisse der Arbeit sowie einen Ausblick auf mögliche Erweiterungen des Systems. In Anhang A finden sich Hinweise zur Installation und Bedienung des Systems zur Erzeugung von sprachbasierten Editoren sowie der Editoren selbst. Den Abschluß der Arbeit bildet der Anhang B, der eine Legende von häufig in der Arbeit verwendeten Abkürzungen und graphischen Symbolen bereitstellt.

Kapitel 2

Präliminarien

In diesem Kapitel werden die in der Arbeit benötigten grundlegenden Begriffe, Notationen und Definitionen eingeführt. Damit geben wir u.a. dem Leser eine formale Einführung der Makroattributgrammatiken, wobei wir weitgehend den Formalismus aus [Les99] übernehmen.

2.1 Allgemeine Begriffe

Wir bezeichnen die leere Menge mit $\emptyset$. Für eine beliebige Menge M bezeichnet $\mathcal{P}(M)$ die Potenzmenge von M. Für zwei beliebige Mengen M und N bezeichnen $M \cup N$ die Vereinigungsmenge, $M \cap N$ den Durchschnitt und $M \setminus N$ die Differenzmenge von M und N. Die Menge der natürlichen Zahlen bezeichnen wir mit $I\!N$ und $I\!N_+$ steht für die Menge $I\!N - \{0\}$. Für alle $k \geq 0$ und die Mengen $M_1, \ldots, M_k$ bezeichnen wir das kartesische Produkt von $M_1, \ldots, M_k$ mit $M_1 \times \ldots \times M_k = \{(m_1, \ldots, m_k) \,|\, (1 \leq i \leq k) \wedge (m_i \in M_i)\}$. Die Konkatenation der zwei Wörter w_1 und w_2 wird mit $w_1 \cdot w_2$ oder einfach mit $w_1 w_2$ bezeichnet. Das leere Wort wird mit ε bezeichnet. Für zwei beliebige Mengen M und N von Zeichenreihen gilt $M \otimes N = \{mn \,|\, m \in M, n \in N\}$. Für eine beliebige Menge M von Zeichenreihen gilt $M^* = \bigcup_{n \geq 0} M^n$ mit $M^0 = \{\varepsilon\}$ und $M^n = M^{n-1} \otimes M$ für $n \geq 1$. Es gilt $M^+ = M^* \otimes M = \bigcup_{n \geq 1} M^n$. Für beliebige Mengen M und N bezeichnet $M - \rightarrow N$ die partielle Abbildung von M nach N.

Im folgenden sei $\Rightarrow$ eine binäre Relation über einer beliebigen Menge M. Es bezeichnet $\Rightarrow^0$ die Identität über M und $\Rightarrow^{n+1}$ für $n \in I\!N$ die $(n+1)$-fache Komposition von $\Rightarrow$. Wir bezeichnen mit $\Rightarrow^*$ und $\Rightarrow^+$ die transitive, reflexive Hülle von $\Rightarrow$ bzw. die transitive Hülle von $\Rightarrow$. Ein Element $m \in M$ heißt *maximal (bezüglich $\Rightarrow$)*, falls es kein $m' \in M$ mit $m \Rightarrow m'$ gibt. Falls $m \Rightarrow^* m'$ für $m, m' \in M$ und m' maximal bezüglich $\Rightarrow$ ist, so wird m' *Normalform von m (bezüglich $\Rightarrow$)* genannt. Im allgemeinen kann ein Element $m \in M$ keine oder mehr als eine Normalform besitzen. Falls die Normalform von m bezüglich $\Rightarrow$ existiert und eindeutig ist, so wird sie mit *nf*$(\Rightarrow, m)$ bezeichnet. Eine Relation $\Rightarrow$ heißt *konfluent*, falls es für alle $m, m_1, m_2 \in M$ mit $m \Rightarrow^* m_1$ und $m \Rightarrow^* m_2$ ein $m' \in M$ mit $m_1 \Rightarrow^* m'$ und $m_2^* \Rightarrow m'$ gibt. Eine Relation heißt *terminierend*, falls es keine unendliche Sequenz $m_1 \Rightarrow m_2 \Rightarrow m_3 \Rightarrow \ldots$ gibt. Für alle $m \in M$ existiert eine eindeutige Normalform von m bezüglich $\Rightarrow$, falls $\Rightarrow$ terminierend und konfluent ist.

2.2 Kontextfreie Grammatiken und Ableitungsbäume

In diesem Abschnitt führen wir die kontextfreien Grammatiken ein, auf welchen die später einzuführenden Makroattributgrammatiken basieren.

Definition 1 (Kontextfreie Grammatik) Eine *kontextfreie Grammatik* ist ein Tupel $G_0 = (N, \Sigma, Z, P)$, wobei

- N die endliche Menge der *Nichtterminalsymbole*,
- Σ die endliche Menge der *Terminalsymbole*,
- $Z \in N$ das *Startsymbol* sowie
- P eine endliche Menge von *Produktionen* der Form $X_0 \to w_0 X_1 w_1 \ldots X_n w_n$ mit $n \geq 0, X_0, \ldots, X_n \in N$ und $w_0, \ldots, w_n \in \Sigma^*$ ist.

Dabei gilt: $N \cap \Sigma = \emptyset$ □

Nun geben wir in einer Definition an, wie die Ableitungsrelation einer kontextfreien Grammatik bestimmt ist.

Definition 2 (Ableitungsrelation) Sei $G_0 = (N, \Sigma, Z, P)$ eine kontextfreie Grammatik. Die *Ableitungsrelation* von G_0, bezeichnet durch $\Rightarrow_{G_0}$, ist die kleinste binäre Relation $\Rightarrow \subseteq (\Sigma \cup N)^* \times (\Sigma \cup N)^*$, für die gilt: Für $\xi_1, \xi_2 \in (\Sigma \cup N)^*$ gilt $\xi_1 \Rightarrow \xi_2$, falls zwei Wörter $\xi, \xi' \in (\Sigma \cup N)^*$ und eine Produktion $X \to \zeta$ in P existieren, sodaß $\xi_1 = \xi X \xi'$ und $\xi_2 = \xi \zeta \xi'$.
Ist G_0 aus dem Kontext bekannt, so wird $\Rightarrow_{G_0}$ mit $\Rightarrow$ abgekürzt.
Eine Sequenz $Z \Rightarrow \xi_1 \Rightarrow \ldots \Rightarrow \xi_n$ mit $\xi_n \in \Sigma^*$ heißt *Ableitung*. □

Anhand der Ableitungsrelation können wir die von einer kontextfreien Grammatik erzeugte Sprache definieren.

Definition 3 (Erzeugte Sprache) Sei $G_0 = (N, \Sigma, Z, P)$ eine kontextfreie Grammatik. Die *von G_0 erzeugte Sprache* ist die Menge $L(G_0) = \{w \in \Sigma^* \mid Z \Rightarrow^*_{G_0} w\}$. □

Abschließend führen wir nun noch *X-Ableitungsbäume* und einige damit verbundene Notationen ein.

Definition 4 (X-Ableitungsbäume) Sei $G_0 = (N, \Sigma, Z, P)$ eine kontextfreie Grammatik. Für alle $X \in N$ ist die *Menge aller X-Ableitungsbäume von G_0*, bezeichnet durch $DT_{G_0}(X)$, wie folgt induktiv definiert:

1. Wenn $X_0 \to a_{0,1} \ldots a_{0,r_0} X_1 a_{1,1} \ldots a_{1,r_1} \ldots X_n a_{n,1} \ldots a_{n,r_n} \in P$
   mit $n \geq 0$, $r_0 \ldots, r_n \geq 0, X_0, \ldots, X_n \in N, a_{0,1}, \ldots, a_{0,r_0}, \ldots, a_{n,1}, \ldots, a_{n,r_n} \in \Sigma$
   und $t_1 \in DT_{G_0}(X_1), \ldots, t_n \in DT_{G_0}(X_n)$, dann ist
   $X_0(a_{0,1}, \ldots, a_{0,r_0}, t_1, a_{1,1}, \ldots, a_{1,r_1}, \ldots, t_n, a_{n,1}, \ldots, a_{n,r_n}) \in DT_{G_0}(X_0)$.

2. Es gibt keine weiteren X-Ableitungsbäume von G_0.

Die *Menge aller Ableitungsbäume von* G_0, die wir mit DT_{G_0} bezeichnen, ist die Menge aller Z-Ableitungsbäume von G_0; es gilt: $DT_{G_0} = DT_{G_0}(Z)$.

Für jedes $X \in N$ und jedes $t \in DT_{G_0}(X)$ bezeichnet $\underline{\text{node}}(t)$ die *Menge aller Knoten von* t und $\underline{\text{root}}(t)$ den *Wurzelknoten von* t. Für jedes $X \in N$ und jedes $t \in DT_{G_0}(X)$ und jedes $x \in \underline{\text{node}}(t)$ bezeichnet $\underline{\text{label}}_t(x)$ die *Markierung von* x und $\underline{\text{prod}}_t(x)$ die *an* x *angewandte Produktion*.
Sei $x \in \underline{\text{node}}(t)$ und $j \in \mathbb{N}_+$, dann bezeichnet $x.-1$ den Vorgängerknoten von x (falls x nicht Wurzel ist), $x.0$ den Knoten selbst und $x.j$ den j-ten Nachfolgerknoten von x, der mit einem Nichtterminalsymbol beschriftet ist (sofern dieser existiert).
Für jedes $X \in N$, jedes $t \in DT_{G_0}(X)$ und jedes $x \in \underline{\text{node}}(t)$ wird x auch durch die Sequenz $j_1.j_2. \dots .j_k$ mit $k \geq 0, j_1, \dots, j_k \in \mathbb{N}_+$, die den Pfad von $\underline{\text{root}}(t)$ zum Knoten x darstellt, angegeben (*Dewey-Notation*). Die Wurzel von t, $\underline{\text{root}}(t)$, wird durch ε bezeichnet. □

2.3 K-sortierte Mengen und K-sortierte Bäume

Bei der Einführung von Makroattributgrammatiken benötigen wir weiterhin die nun zu definierenden K-sortierten Mengen und K-sortierten Bäume.

Definition 5 (K-sortierte Menge) Sei K eine Menge von Sorten. Eine *K-sortierte Menge* ist ein Paar $(\Omega, sort)$, wobei Ω eine endliche Menge von Symbolen und $sort : \Omega \to K$ eine Abbildung, welche jedem Symbol $a \in \Omega$ eine Sorte $\kappa \in K$ zuordnet, ist.
Ein Symbol $a \in \Omega$ mit $sort(a) = \kappa$ wird auch durch a^κ bezeichnet. Für jede Sorte $\kappa \in K$ bezeichnet Ω^κ die Untermenge von Ω, die alle Symbole a^κ enthält: $\Omega^\kappa = \{a \,|\, (a \in \Omega) \wedge (sort(a) = \kappa)\}$. Ist $sort$ aus dem Kontext ersichtlich, so kürzen wir $(\Omega, sort)$ auch durch Ω ab. □

Wir definieren die K-sortierten Bäume entsprechend:

Definition 6 (K-sortierte Bäume) Sei Σ eine $(K^* \times K)$-sortierte Menge und Y eine K-sortierte Menge. Die *Menge der K-sortierten Bäume über Σ indiziert mit Y*, bezeichnet durch $T_\Sigma(Y)$, ist die kleinste K-sortierte Menge $\bigcup_{\kappa \in K} T^\kappa$, für die gilt:

1. $Y^\kappa \subseteq T^\kappa$ und

2. für jedes $\sigma \in \Sigma^{\kappa_1 \cdots \kappa_k, \kappa_0}$ mit $k \geq 0$ und $\kappa_0, \dots, \kappa_k \in K$ sowie für jedes $t_1 \in T^{\kappa_1}, \dots, t_k \in T^{\kappa_k}$ ist $\sigma(t_1, \dots, t_k) \in T^{\kappa_0}$

Mit $\underline{\text{sym}}(t)$ bezeichnen wir die Menge der in t vorkommenden Symbole aus Σ. Für einen Baum t und zwei Listen $v_1, \dots, v_l$ und $u_1, \dots, u_l$ von Bäumen mit $l \geq 0$ bezeichnet $t[v_1/u_1, \dots, v_l/u_l]$ den Baum, den man durch Ersetzung jedes Vorkommens von v_i in t durch u_i erhält, wobei angenommen wird, daß für jedes $i, j \in \{1, \dots, l\}$ die Bäume v_i und v_j nicht überlappend sind. □

2.4 Graphen

An dieser Stelle führen wir gerichtete und gerichtete azyklische Graphen ein, um später Abhängigkeitsgraphen von Ableitungsbäumen und die Nichtzirkularitäts-Eigenschaft von Makroattributgrammatiken definieren zu können.

Definition 7 (Gerichteter Graph) Ein *gerichteter Graph* ist ein Paar (V, E) bestehend aus einer *Knotenmenge* V und einer binären Relation $E \subseteq V \times V$, welche Kantenmenge genannt wird. □

Wir definieren nun die Eigenschaft eines Graphens, azyklisch zu sein:

Definition 8 (Gerichteter azyklischer Graph) Ein gerichteter Graph $G = (V, E)$ heißt *azyklisch*, wenn E^+ kein Paar (v, v) enthält, wobei $v \in V$ ist. □

2.5 Makroattributgrammatiken

Unter Verwendung der vorangegangenen Definitionen sind wir in der Lage, Makroattributgrammatiken zu definieren.

Definition 9 (Makroattributgrammatik) Eine *Makroattributgrammatik* ist ein Tupel $M = (G_0, D, B, R)$ bestehend aus

- einer *kontextfreien Grammatik* G_0,
- einem *semantischen Bereich* D,
- einer *Attributbeschreibung* B sowie
- einer *Beschreibung* R *semantischer Regeln.*

G_0 ist ein Tupel (N, Σ, Z, P) gemäß Definition 1.

D ist ein Tupel $(K, \Omega, \Phi, \varphi)$ bestehend aus

- einer endlichen Menge K von Sorten,
- einer K-sortierten Menge Ω (*Wertemenge*),
- einer endlichen $(K^* \times K)$-sortierten Menge Φ von *Funktionssymbolen* und
- einer *Interpretation* φ der Funktionssymbole, die jedem Symbol $f \in \Phi$ einer Funktion über Teilmengen von Ω zuordnet:

  Für jedes $k \in I\!N$ und für alle $\kappa_0, \ldots, \kappa_k \in K$ gilt: Für jedes $f \in \Phi^{(\kappa_1 \ldots \kappa_k, \kappa_0)}$ ist $\varphi(f) : \Omega^{\kappa_1} \times \ldots \times \Omega^{\kappa_k} \to \Omega^{\kappa_0}$ eine k-stellige berechenbare Funktion.

B ist ein Tupel $(S\text{-}Att, I\text{-}Att, A, \alpha_0)$ bestehend aus

- zwei disjunkten endlichen $(K^* \times K)$-sortierten Mengen $S\text{-}Att$ und $I\text{-}Att$, wobei die Elemente von $S\text{-}Att$ als *synthetische Attribute* und die Elemente von $I\text{-}Att$ als *inherite Attribute* bezeichnet werden,
- einer Abbildung $A : N \to \mathcal{P}(\mathcal{S}\text{-}Att \cup I\text{-}Att)$, die jedem Nichtterminalsymbol $X \in N$ eine Menge von Attributen zuordnet, wobei $A(Z) \cap I\text{-}Att = \emptyset$ gilt, sowie
- einem ausgezeichneten Attribut $\alpha_0 \in A(Z)$ mit $sort(\alpha_0) \in (\{\varepsilon\} \times K)$, welches *Bedeutungsattribut* heißt.

Sei $p = X_0 \to w_0 X_1 w_1 \dots X_n w_n \in P$. Die $(K^* \times K)$-sortierte Menge $A(p)$ der *Attributvorkommen von* p ist die Menge $A(p) = \{\langle a, j\rangle \mid (0 \leq j \leq n) \wedge (a \in A(X_j))\}$,
wobei $sort(\langle a, j\rangle) = sort(a)$ für alle $\langle a, j\rangle \in A(p)$.
Die Menge $A(p)$ läßt sich in zwei disjunkte Teilmengen zerlegen:

- die Menge der *Innenattributvorkommen von* p: $\underline{\text{innen}}(p) =$
$\{\langle a, j\rangle \mid a \in A(X_0) \cap S\text{-}Att\} \cup \{\langle a, j\rangle \mid a \in A(X_j) \cap I\text{-}Att, 1 \leq j \leq n\}$
- die Menge der *Außenattributvorkommen von* p: $\underline{\text{außen}}(p) =$
$\{\langle a, j\rangle \mid a \in A(X_0) \cap I\text{-}Att\} \cup \{\langle a, j\rangle \mid a \in A(X_j) \cap S\text{-}Att, 1 \leq j \leq n\}$

R ist ein Paar (Y, RP) bestehend aus

- einer endlichen K-sortierten Menge Y von *Kontextparametern* und
- einer Familie $RP = (R(p) \mid p \in P)$ von *semantischen Regeln.*

  Für jede Produktion p und jedes Innenattributvorkommen $\langle a, j\rangle \in \underline{\text{innen}}(p)$ mit $sort(\langle a, j\rangle) = (\kappa_1 \dots \kappa_k, \kappa_0), k \geq 0, \kappa_0, \dots, \kappa_k \in K$ enthält die Menge $R(p)$ genau eine semantische Regel der Form $\langle a, j\rangle(y_1^{\kappa_1}, \dots, y_k^{\kappa_k}) = \zeta_{\langle a,j\rangle}$, wobei $y_1, \dots, y_k \in Y$ und $\zeta_{\langle a,j\rangle} \in T_{\Phi \cup \underline{au\ss en}(p)}(\{y_1^{\kappa_1}, \dots, y_k^{\kappa_k}\})^{\kappa_0}$ gilt.

Im Sinne der funktionalen Programmierung sprechen wir statt von synthetischen und inheriten Attributen auch von *synthetischen Funktionen* und *inheriten Funktionen.* □

Die Sortierungen werden im folgenden manchmal weggelassen.

Wir definieren nun die Attributinstanzen, die wir im weiteren noch benötigen werden.

Definition 10 (Attributinstanzen) Sei $M = (G_0, D, B, R)$ eine Makroattributgrammatik und $t \in DT_{G_0}$ ein Ableitungsbaum von G_0. Sei $x \in \underline{\text{node}}(t)$ mit $\underline{\text{label}}(x) \in N$ ein Knoten von t. Die *Menge der Attributinstanzen von* x, bezeichnet durch $A(x)$, ist die $(K^* \times K)$-sortierte Menge $A(x) = \{\langle a, x\rangle \mid a \in A(\underline{label}(x))\}$, wobei für jedes $\langle a, x\rangle \in A(x)$ gilt: $sort(\langle a, x\rangle) = sort(a)$. Die *Menge der Attributinstanzen von* t, bezeichnet durch $A(t)$, ist die $(K^* \times K)$-sortierte Menge $A(t) = \bigcup_{x \in \underline{node}(t), \underline{label}_t(x) \in N} A(x)$. □

Nun definieren wir die Einbettung semantischer Regeln.

Definition 11 (Einbettung) Sei $M = (G_0, D, B, R)$ eine Makroattributgrammatik, $t \in DT_{G_0}$ ein Ableitungsbaum von G_0, $x \in \underline{node}(t)$ und $p = \underline{prod}_t(x)$ die am Knoten x angewandte Produktion. Sei $\zeta \in T_{\Phi \cup \underline{außen}(p)}(Y)^{\kappa_0}$ mit $\kappa_0 \in K$. Die *Einbettung von* ζ *an* x, bezeichnet durch $[\zeta]_x$, ist der wie folgt definierte Term aus $T_{\Phi \cup A(t)}(Y)^{\kappa^0}$:

1. $[\zeta]_x = \zeta$, falls $\zeta \in Y$
2. $[\zeta]_x = \phi([\zeta_1]_x, \ldots, [\zeta_k]_x)$, falls $\zeta = \phi(\zeta_1, \ldots, \zeta_k)$ und $\phi \in \Phi^{\kappa_1 \cdots \kappa_k, \kappa_0}$, wobei $k \geq 0$ und für jedes $i \in \{1, \ldots, k\}$ gilt: $\kappa_i \in K$ und $\zeta_i \in T_{\phi \cup \underline{außen(p)}}(Y)^{\kappa_i}$.
3. $[\zeta]_x = \langle b, x.j\rangle([\zeta_1]_x, \ldots, [\zeta_k]_x)$, falls $\zeta = \langle b, j\rangle(\zeta_1, \ldots, \zeta_k)$ und $\langle b, j\rangle \in \underline{außen}(p)^{(\kappa_1 \cdots \kappa_k, \kappa_0)}$, wobei $k \geq 0$ und für jedes $i \in 1, \ldots, k$ gilt: $\kappa_i \in K$ und $\zeta_i \in T_{\phi \cup \underline{außen(p)}}(Y)^{\kappa_i}$. □

Nun können wir semantische Gleichungen definieren.

Definition 12 (Semantische Gleichungen) Sei $M = (G_0, D, B, R)$ eine Makroattributgrammatik, $t \in DT_{G_0}$ ein Ableitungsbaum von G_0, $x \in \underline{node}(t)$ und $p = \underline{prod}_t(x)$. Sei $\langle a, i\rangle(y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}) = \zeta_{\langle a,i\rangle}$ eine semantische Regel von p. Dann heißt $\langle a, x.i\rangle(y_1^{\kappa_1}, \ldots, y_k^{\kappa_k}) = [\zeta_{\langle a,i\rangle}]_x$ *semantische Gleichung von* t. Die *Menge aller semantischen Gleichungen von* t bezeichnen wir mit $R(t)$. □

Jetzt definieren wir den Abhängigkeitsgraphen eines Ableitungsbaumes.

Definition 13 (Abhängigkeitsgraph eines Ableitungsbaumes)
Sei $M = (G_0, D, B, R)$ eine Makroattributgrammatik und $t \in DT_{G_0}$ ein Ableitungsbaum von G_0. Der *Abhängigkeitsgraph von* t, bezeichnet durch $D(t)$, ist der gerichtete Graph $D(t) = (A(t), E(t))$, wobei $E(t) = \{(\langle b, x.j\rangle, \langle a, x.i\rangle) \,|\, (x \in \underline{node}(t)) \wedge ((\langle a, x.i\rangle(y_1, \ldots, y_k) = [\zeta_{\langle a,i\rangle}]_x) \in R(t)) \wedge (\langle b, x.j\rangle \in (\ \underline{sym}([\zeta_{\langle a,i\rangle}]_x) \cap A(t)))\}$. □

Wir sind jetzt in der Lage, eine nichtzirkuläre Makroattributgrammatik zu definieren.

Definition 14 (Nichtzirkuläre Attributgrammatik) Eine Makroattributgrammatik $M = (G_0, D, B, R)$ heißt *nichtzirkulär*, wenn für jeden Ableitungsbaum $t \in DT_{G_0}$ gilt, daß der Abhängigkeitsgraph $D(t)$ keinen Zykel enthält. □

2.6 Reduktionsrelation und Reduktionssemantik

Wir definieren jetzt die Reduktionsrelation für Makroattributgrammatiken.

Definition 15 (Reduktionsrelation) Sei $M = (G_0, D, B, R)$ eine Makroattributgrammatik und $t \in DT_{G_0}$ ein Ableitungsbaum von G_0. Die *Reduktionsrelation von* M *und* t, bezeichnet durch $\Rightarrow_{M,t}$, ist die kleinste binäre Relation $\Rightarrow \subseteq T_{\Phi \cup A(t)}(\Omega) \times T_{\Phi \cup A(t)}(\Omega)$, für die gilt:

1. $\hat{t}\,[@/\langle a,x\rangle(\xi_1,\ldots,\xi_n)] \Rightarrow \hat{t}\,[@/[\zeta]_x[y_1/\xi_1,\ldots,y_n/\xi_n]]$,
   für alle $\hat{t} \in T_{\Phi\cup A(t)}(\Omega \cup \{@\})$ mit genau einem Vorkommen @ in $\hat{t}$, $n \geq 0$, $\xi_1,\ldots,\xi_n \in T_{\Phi\cup A(t)}(\Omega)$ und $x \in \underline{node}(t)$, falls $a \in S\text{-}Att$ und $(\langle a,0\rangle(y_1,\ldots,y_n) = \zeta) \in R(\underline{prod}_t(x))$.

2. $\hat{t}\,[@/\langle a,x.i\rangle(\xi_1,\ldots,\xi_n)] \Rightarrow \hat{t}\,[@/[\zeta]_x[y_1/\xi_1,\ldots,y_n/\xi_n]]$,
   für alle $\hat{t} \in T_{\Phi\cup A(t)}(\Omega \cup \{@\})$ mit genau einem Vorkommen @ in $\hat{t}$, $i \in I\!N_+$ sowie $n \geq 0$, $\xi_1,\ldots,\xi_n \in T_{\Phi\cup A(t)}(\Omega)$ und $x \in \underline{node}(t)$, falls $a \in I\text{-}Att$ und $(\langle a,i\rangle(y_1,\ldots,y_n) = \zeta) \in R(\underline{prod}_t(x))$.

3. $\hat{t}\,[@/\phi(\xi'_1,\ldots,\xi'_n)] \Rightarrow \hat{t}\,[@/\varphi(\phi)(\xi'_1,\ldots,\xi'_n)]$,
   für alle $\hat{t} \in T_{\Phi\cup A(t)}(\Omega \cup \{@\})$ mit genau einem Vorkommen @ in $\hat{t}$, $\phi \in \Phi$ sowie $n \geq 0$, $\xi'_1,\ldots,\xi'_n \in \Omega$. □

Wir definieren nun noch die Reduktionssemantik von Makroattributgrammatiken.

Definition 16 (Reduktionssemantik von Makroattributgrammatiken)
Sei $M = (G_0, D, B, R)$ eine Makroattributgrammatik und $t \in DT_{G_0}$ ein Ableitungsbaum von G_0. Die *Reduktionssemantik von M und t*, bezeichnet durch $\underline{sem}(M,t)$, ist definiert als die Normalform der Attributinstanz $\langle \alpha_0, \underline{root}(t)\rangle$ (Attributinstanz des Bedeutungsattributs α_0 an der Wurzel von t) bezüglich der Reduktionsrelation $\Rightarrow_{M,t}$, sofern die Normalform existiert: $\underline{sem}(M,t) = nf(\Rightarrow_{M,t} \langle \alpha_0, \varepsilon\rangle)$. □

Zum Abschluß sei noch folgende Vermutung angeführt:

Vermutung 1 Sei $M = (G_0, D, B, R)$ eine nichtzirkuläre Makroattributgrammatik und $t \in DT_{G_0}$. Dann ist $\Rightarrow_{M,t}$ terminierend und konfluent. □

Wenn die Vermutung 1 gilt, dann existiert für nichtzirkuläre Makroattributgrammatiken die Normalform im Sinne von Definition 16 mit Sicherheit.

Kapitel 3

Beispiel einer Editiersitzung

In diesem Kapitel demonstrieren wir anhand des Editors, welcher auf der formalen Beispielsprache $\underline{P}$ basiert, die Eigenschaften der von dem System $\underline{Gen_{SBE}}$ erzeugten sprachbasierten Editoren. Dabei bezeichnet $\underline{Gen_{SBE}}$ den in dieser Arbeit entworfenen Generator für sprachbasierte Editoren. Den Editor, der auf einer formalen Sprache basiert, welche wir mit $\underline{L}$ bezeichnen, kürzen wir im folgenden mit $\underline{SBE(L)}$ ab. Da wir uns in diesem Kapitel auf eine formale Sprache beziehen, welche wir mit $\underline{P}$ bezeichnen, beschreiben wir hier also die Nutzung des Editors $\underline{SBE(P)}$. Dabei beschreiben wir eine Editiersitzung aus Sicht der Person, die den $\underline{SBE(P)}$ benutzt, um ein Programm der Sprache $\underline{P}$ zu editieren. Diese Person nennen wir $\underline{Nutzer_{SBE(P)}}$.
Die Sprache $\underline{P}$ ist eine sehr einfach gehaltene imperative Sprache, in welcher nur vier Arten von Daten - `integer`, `boolean`, `char` und `string` - und vier Arten von Anweisungen - Zuweisung, bedingte Anweisung, While-Schleife und Block von Anweisungen - existieren. Die Sprache ist streng getypt und es kann statisch geprüft werden, ob eine Variable typgerecht verwendet wird.
Das Programm der Sprache $\underline{P}$, welches durch den $\underline{Nutzer_{SBE(P)}}$ mit Hilfe des Editors $\underline{SBE(P)}$ editiert wird, nennen wir auch editiertes Konstrukt der Sprache $\underline{P}$ und bezeichnen es mit der Abkürzung $\underline{EK_P}$. Durch die Editiersitzung beschreiben wir das strukturelle und textuelle Editieren dieses Programmes $\underline{EK_P}$, die Repräsentation des $\underline{EK_P}$ für den $\underline{Nutzer_{SBE(P)}}$ sowie die Handhabung von Verletzungen der kontextsensitiven Nebenbedingungen der Sprache $\underline{P}$ durch den $\underline{SBE(P)}$. In bezug auf den letzten Punkt sei gesagt, daß der Editor $\underline{SBE(P)}$ fortlaufend das editierte $\underline{P}$-Programm nach Verletzungen der folgenden drei kontextsensitiven Nebenbedingungen der Sprache $\underline{P}$ überprüft:

- Die in den Ausdrücken und in den linken Seiten der Zuweisungen verwendeteten Variablen müssen deklariert sein.
- Die Bestandteile der Ausdrücke und der Anweisungen müssen typgerecht auftreten.
- Variablen dürfen nicht mehrfach deklariert werden.

Für jeden auftretenen Fehler im $\underline{P}$-Programm zeigt der Editor $\underline{SBE(P)}$ einen Kommentar an, welcher den Fehler und dessen Position beschreibt.

Wir beschreiben hier die Verwendung des sehr einfach gehaltenen sprachbasierten Editors $\underline{SBE(P)}$, damit wir seine Spezifikation, abgekürzt durch $\underline{Spez_{SBE(P)}}$, in Kapitel 5 vollständig erläutern können. In den folgenden Kapiteln diskutieren wir aber auch Möglichkeiten der Entwicklung von sprachbasierten Editoren $\underline{SBE(L)}$, welche bei der Erstellung des $\underline{SBE(P)}$ nicht genutzt worden sind.

Wir beginnen mit der Beschreibung der Oberfläche des $\underline{SBE(P)}$, die dem $\underline{Nutzer_{SBE(P)}}$ präsentiert wird, wenn er eine Editiersitzung startet (siehe dazu Abbildung 3.1).
Die **Titel-Leiste** der Oberfläche zeigt den Namen der Datei, welche der $\underline{Nutzer_{SBE(P)}}$ für Editieren des Programmes der Sprache $\underline{P}$ verwendet. Unter der **Titel-Leiste** befindet sich

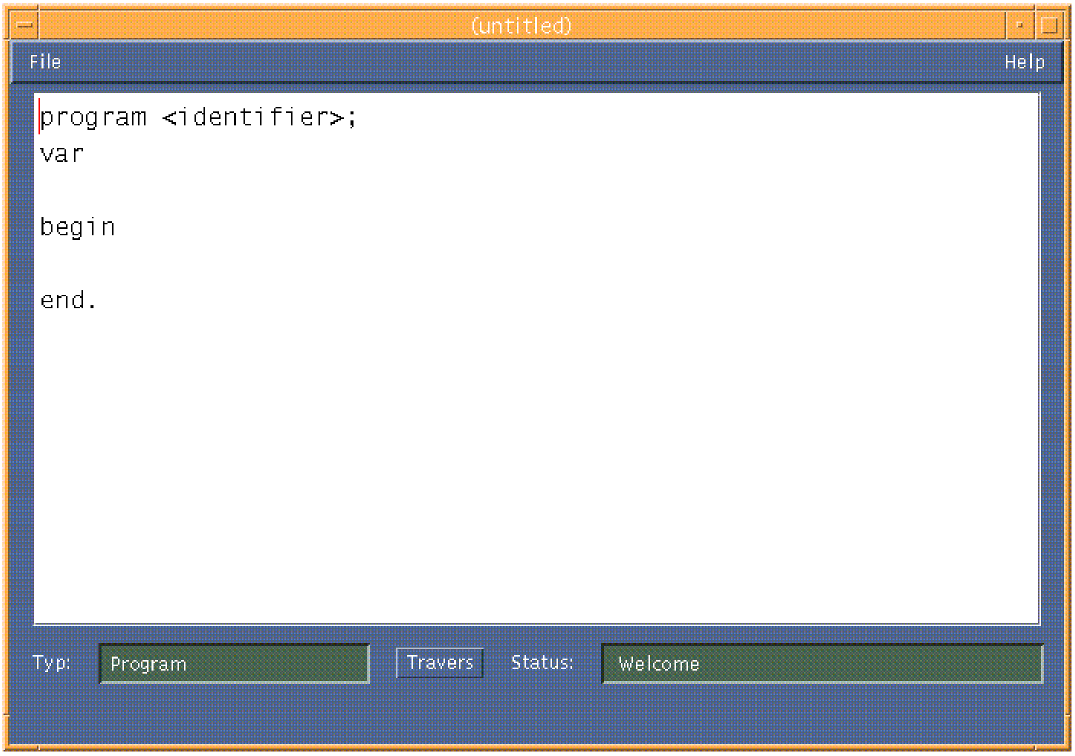

Abbildung 3.1: Screenshot

die Menü-Leiste, welche die zwei Menü-Punkte **File** und **Help** anbietet. Der Menü-Punkt **File** stellt die Funktionen zum Laden und Speichern von Dateien sowie dem Erzeugen neuer Dateien bereit (für den Beginn der Editation eines neuen $\underline{P}$-Programms). Weiterhin kann der $\underline{Nutzer_{SBE(P)}}$ hier den Editor beenden. **Help** bietet in dieser ersten Version nur die Anzeige der Versionsnummer an.
Darunter folgt das **Editierfenster**, welches das $\underline{EK_P}$ anzeigt. Weiterhin birgt es die Möglichkeit zur Auswahl eines Teils von $\underline{EK_P}$, welches es zur Unterscheidung vom restlichen Programmteil besonders hervorhebt. Auch das textuelle Editieren von $\underline{EK_P}$ erfolgt innerhalb dieses Editierfensters.
Im verbleibenden Bereich der Editoroberfläche befindet sich die Anzeige **Typ** (zur Infor-

mation über den ausgewählten Teil von $\underline{EK_P}$) und die Anzeige **Status** (zur Wiedergabe von Information über den zuletzt getätigten Editierbefehl des $\underline{Nutzer_{SBE(P)}}$). Dazwischen liegt der Kommandoknopf mit der Aufschrift **Travers**, dessen Funktion noch erklärt wird. Unterhalb dieser Komponenten befindet sich noch Platz für **Kommandoknöpfe**, die zur Auslösung von strukturellen Transformationen des $\underline{EK_P}$ genutzt werden können. Diese erscheinen aber nur dann, wenn die entsprechenden Transformationen vom $\underline{SBE(P)}$ im aktuellen Kontext erlaubt werden.

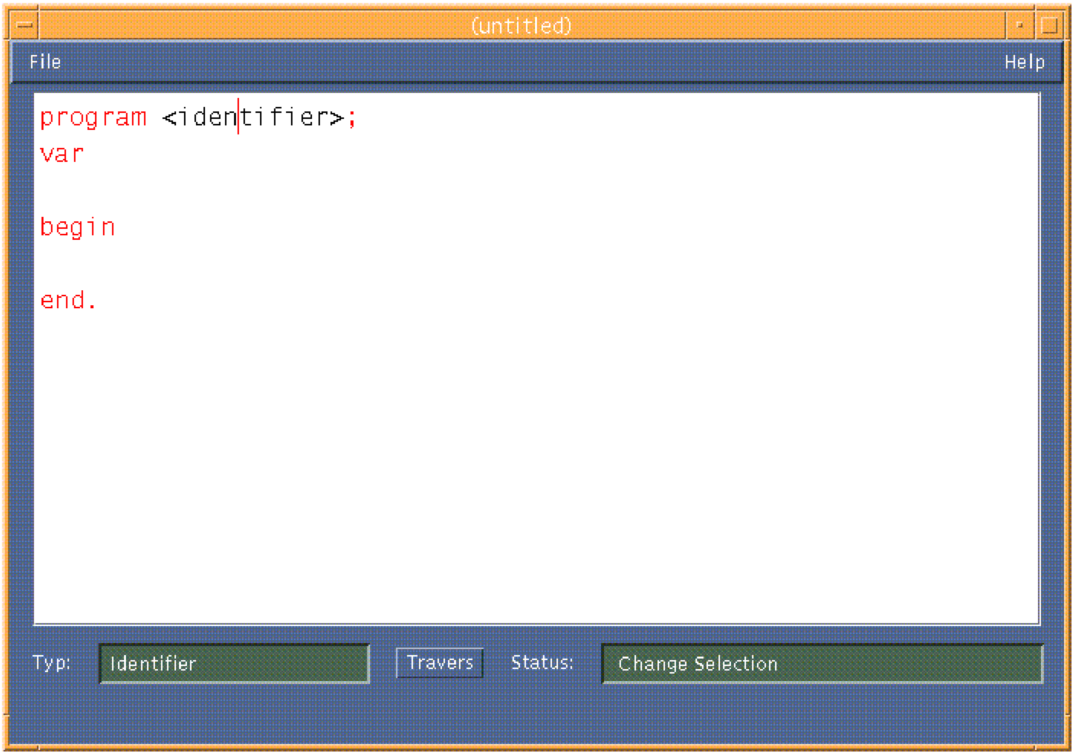

Abbildung 3.2: Screenshot

Mit dem fensterorientierten Editor $\underline{SBE(P)}$ wird das $\underline{EK_P}$ als hierarchische Struktur editiert, d.h. das editierte $\underline{P}$-Programm ist nicht nur eine Sequenz von Buchstaben und Zeilen. Der initiale Inhalt des **Editierfenster** besteht aus einer Schablone für ein Programm. Eine Schablone ist aus Schlüsselwörtern und Platzhaltern aufgebaut, wobei die Platzhalter die Möglichkeit zum Einsetzen weiterer Programmkomponenten bereitstellen. Somit kann eine Top-Down-Entwicklung der Programme $\underline{EK_P}$ stattfinden, bei welcher neue Anweisungen und Ausdrücke in das Skelett der vorher eingegebenen Schablonen eingefügt werden. Die möglichen Editierbefehle werden durch die Struktur von $\underline{EK_P}$ und der aktuellen Auswahl eines Teiles von $\underline{EK_P}$ bestimmt. Der ausgewählte Teil des Programms $\underline{EK_P}$ wird im **Editierfenster** besonders hervorgehoben (durch eine andere Farbe). In Abbildung 3.1 ist das gesammte $\underline{EK_P}$ ausgewählt, welches durch die dunkle Farbe der textuellen Darstellung von $\underline{EK_P}$ erkennbar ist. Es ist außerdem der Platzhalter `<identifier>` zu erkennen, welcher das Einfügen des Programmnamens ermöglicht. Platzhalter sind entweder durch solche spitzen Klammern begrenzte Zeichenketten oder sie werden nur durch einen leeren

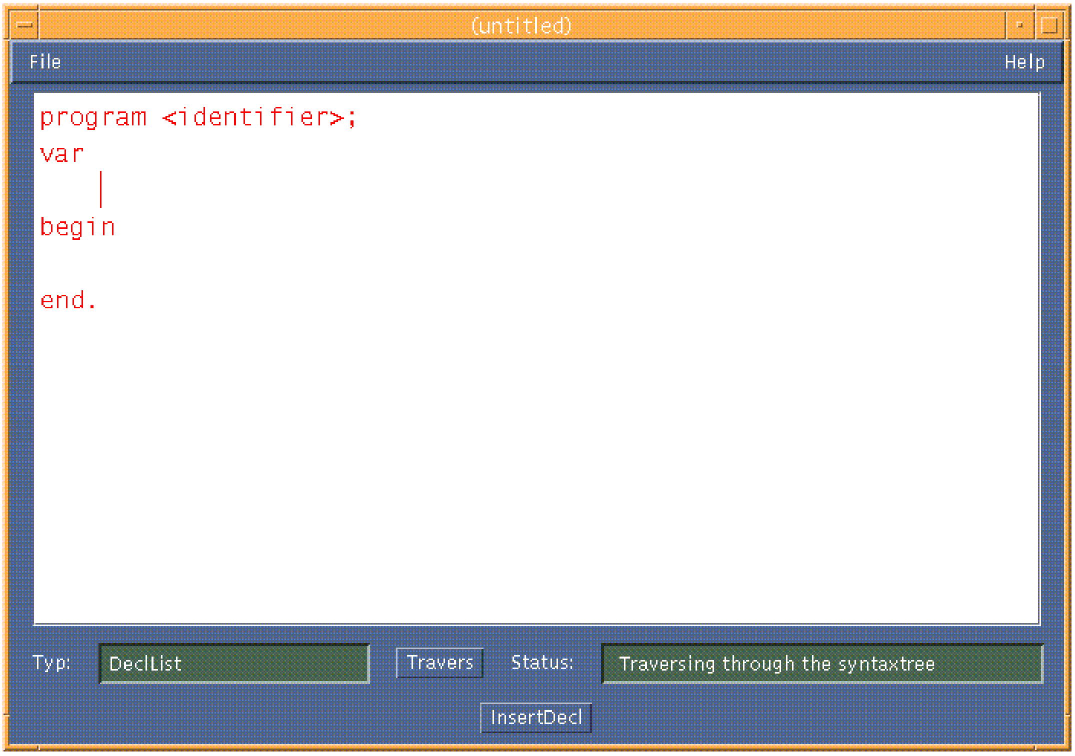

Abbildung 3.3: Screenshot

String repräsentiert. Wie ihre Existenz und Position im zweiten Fall vom $\underline{Nutzer_{SBE(P)}}$ erkannt werden kann, wird noch erklärt. Es sei hier nur darauf hingewiesen, daß sich in Abbildung 3.1 zwischen den Schlüsselwörtern `var` und `begin` bzw. zwischen `begin` und `end` je ein solcher unsichtbarer Platzhalter befindet, welcher zum Einfügen von Deklarationen bzw. von Anweisungen dient.

Der Editor $\underline{SBE(P)}$ ist aber nicht nur ein struktureller, er ermöglicht ebenfalls textuelle Modifikationen von $\underline{EK_P}$ im Sinne von Texteingaben und dem Löschen von Text. Während einer textuellen Modifikation wird durch den Cursor nicht nur ein Teil der Struktur des $\underline{EK_P}$ ausgewählt, sondern innerhalb dieser Selektion auch eine konkrete Position im einzugebenen Text.

Die strukturelle wie auch die textuelle Selektion wird durch den Cursor innerhalb des **Editierfensters** gesetzt. Dieses kann entweder durch die Maus (Anklicken der gewünschten Cursorposition) oder durch die Cursortasten und die Tasten Backspace, End und Home auf der Tastatur erfolgen. Der erste Editierbefehl in unserer Editiersitzung ist eine solche Plazierung des Cursors durch die Verwendung der Maus. In Abbildung 3.2 ist zu erkennen, daß der Cursor innerhalb des Platzhalters `<identifier>` gesetzt wurde. Die Änderung der Hervorhebungen in der textuellen Darstellung von $\underline{EK_P}$ zeigt die nun aktuelle Auswahl des Strukturteils des Programms, welcher durch den Platzhalter beschrieben wird, an. In der Anzeige *Typ* ist weiterhin zu erkennen, daß wir einen Identifier ausgewählt haben. Die Anzeige *Status* weist uns ferner auf die geänderte Auswahl hin.

Die Auswahl eines Teil der syntaxbasierten Struktur von $\underline{EK_P}$ kann aber auch noch auf eine andere Weise geändert werden. Dazu dient der Kommandoknopf `Travers`, der sich

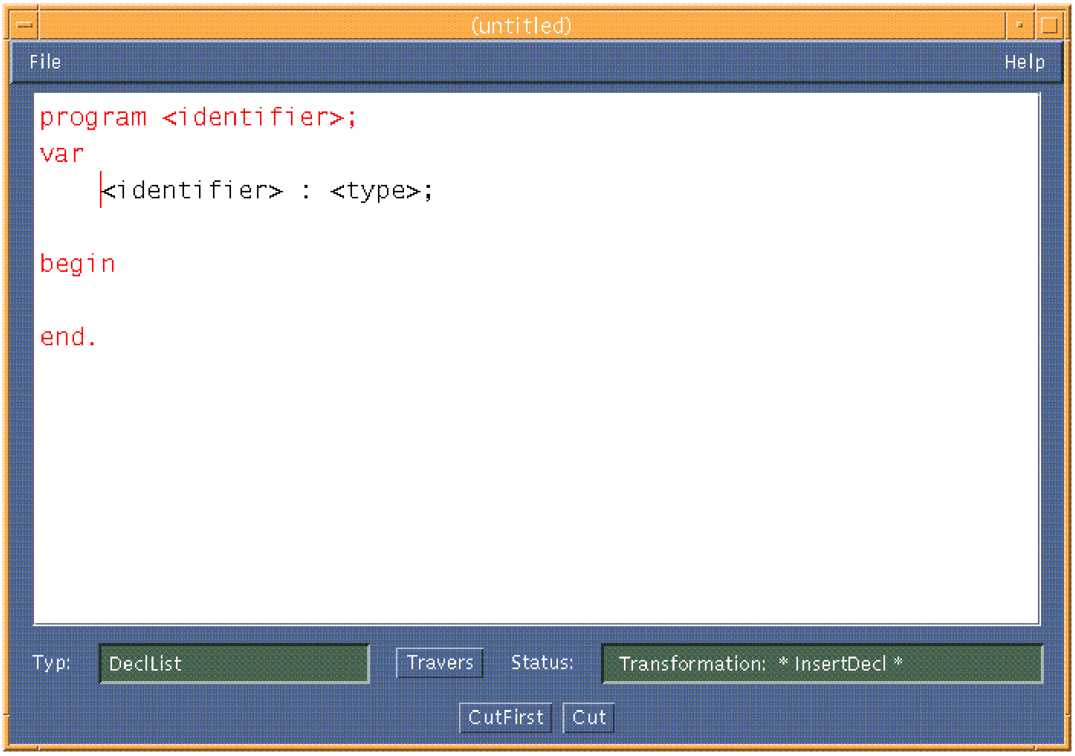

Abbildung 3.4: Screenshot

stets zwischen den Anzeigen `Typ` und `Status` befindet. Wird er durch Anklicken mit der Maus aktiviert, so wechselt die aktuelle Auswahl zum nächsten Knoten im abstrakten Syntaxbaum von $\underline{EK_P}$ (interne Darstellung) im Sinne eines Durchlaufes (top-down left-to-right). In Abbildung 3.3 ist das Ergebnis des Betätigens des Kommandoknopfes `Travers` dargestellt. Die Cursorposition hat gewechselt und befindet sich nun an der Position des unsichtbaren Platzhalters zwischen den Schlüsselwörtern **var** und **begin**. Somit ist der Typ der aktuellen Auswahl DeclList und wir können hier eine Deklaration in die noch leere Liste von Deklarationen einfügen. Dazu ist der Kommandoknopf mit der Aufschrift `InsertDecl` erschienen, welcher die gleichnamige strukturelle Transformation auslösen kann. Abbildung 3.4 zeigt den Editor $\underline{SBE(P)}$ nach Betätigen dieses Kommandoknopfes. Es wird der Platzhalter durch die Schablone ersetzt, die eine Deklaration repräsentiert. Diese enthält wiederum zwei Platzhalter, einen für den Namen der zu deklarierenden Variable, und einen für den Typ dieser Variable. In der Anzeige `Status` spiegelt sich der Name der ausgelösten strukturellen Transformation wider. Obwohl sich die strukturelle Selektion nicht geändert hat, erscheinen nun zwei andere Kommandoknöpfe und der Kommandoknopf mit der Aufschrift `InsertDecl` ist verschwunden. Dies bedeutet, daß am Anfang der Liste von Deklarationen keine neue Deklaration eingefügt, aber dafür das erste Element dieser Liste (mit `CutFirst`) bzw. die gesamte Liste (mit `Cut`) wieder entfernt werden kann. Wir wollen nun die Deklaration genauer spezifizieren. Da wir mit dem Editor $\underline{SBE(P)}$ keine zusammengesetzten Strukturen (wie z.B. eine Deklaration) textuell eingeben können, beschränken wir uns bei der textuellen Eingabe auf den Namen der zu deklarierenden Variable. Dazu müssen wir die aktuelle Auswahl auf den Platzhalter `<Identifier>` inner-

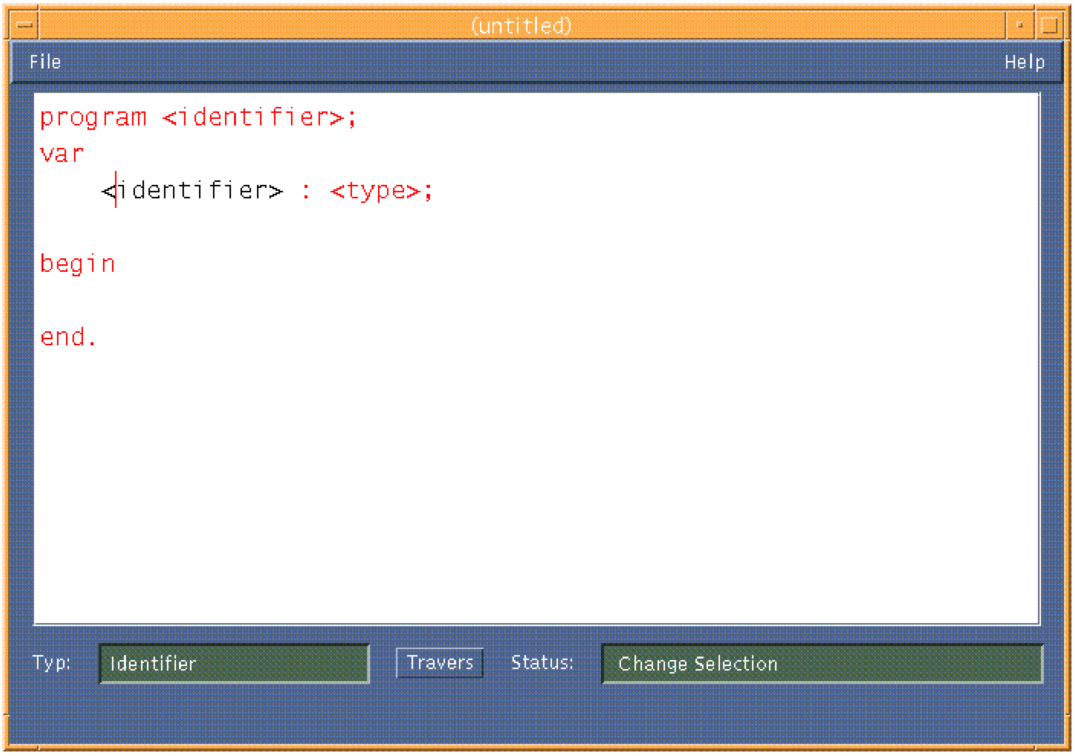

Abbildung 3.5: Screenshot

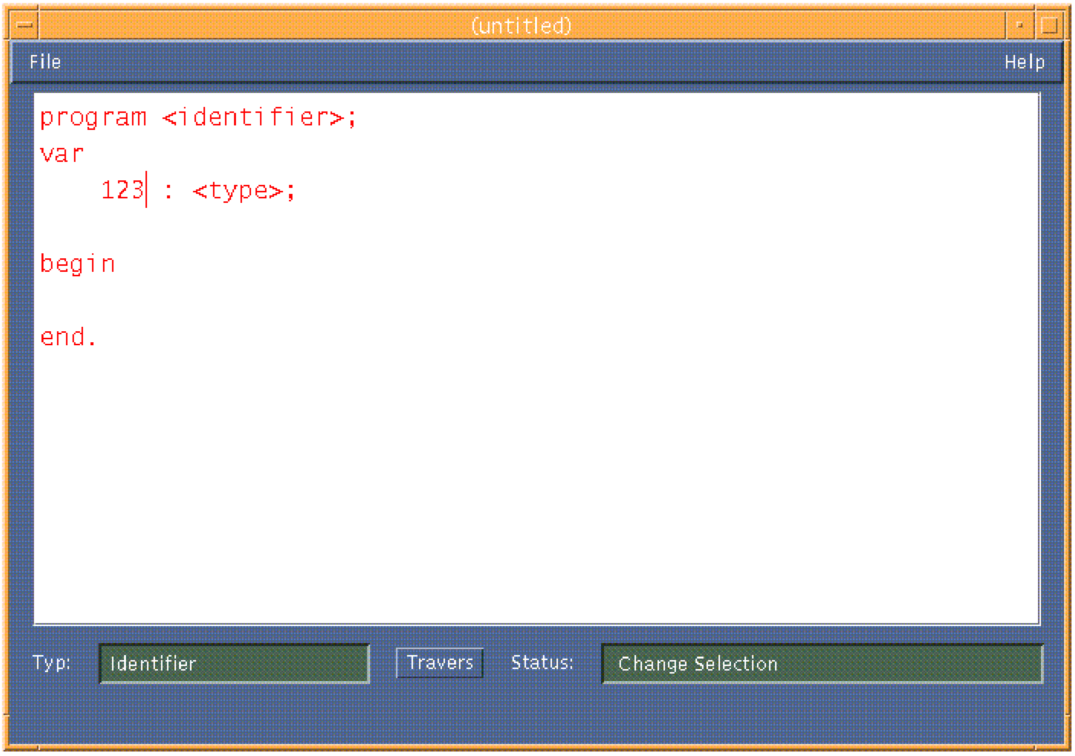

Abbildung 3.6: Screenshot

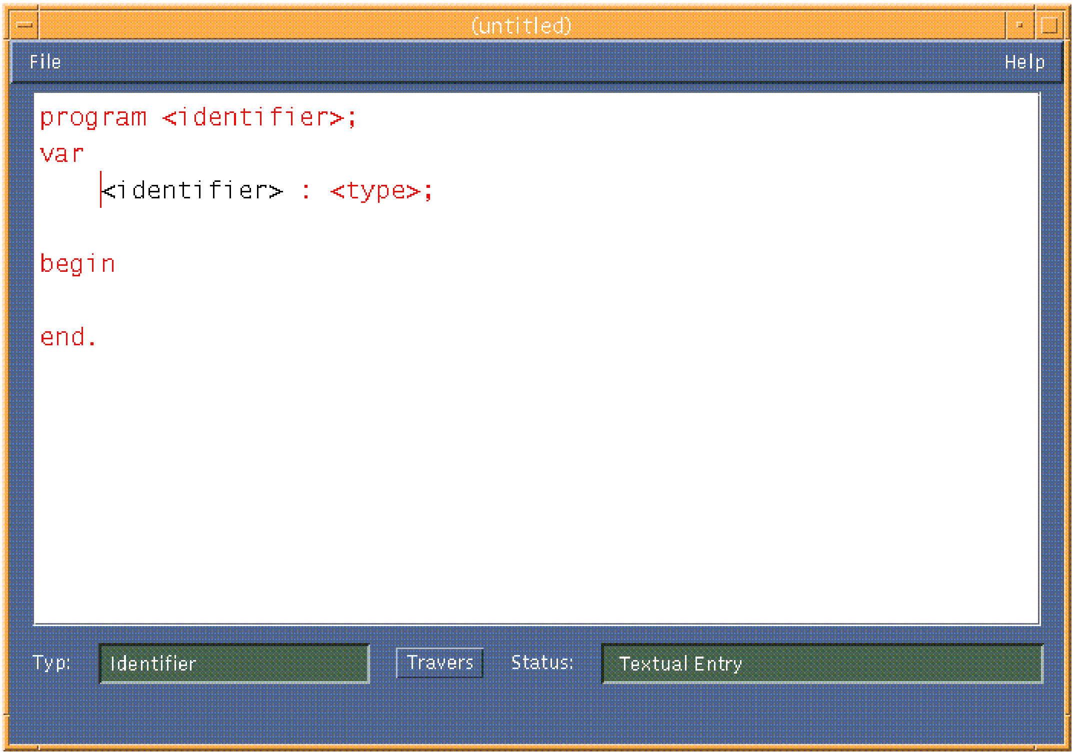

Abbildung 3.7: Screenshot

halb der Deklarations-Schablone setzen. Dies erreichen wir z.B. durch Betätigen der rechten Cursortaste, da so der Cursor seine Position innerhalb des Platzhalters `<Identifier>` ändert und dies zu der gewünschten Selektion führt. In Abbildung 3.5 ist das Resultat zu erkennen, wobei auffällt, daß die Kommandoknöpfe zur strukturellen Transformation verschwunden sind. Dies ist sinnvoll, da einerseits der Typ der Selektion jetzt Identifier ist und man andererseits einen Bezeichner nicht strukturell eingeben kann. Der Editor $\underline{SBE(P)}$ wechselt in den Modus zur textuellen Eingabe, sobald wir mit Hilfe der Tastatur ein Zeichen eingeben bzw. löschen. Beginnen wir mit der Eingabe eines Zeichens, so verschwindet der aktuell ausgewählte Text und schafft so Platz für die neue Eingabe. Dagegen bleibt der alte Text erhalten (bis auf das zu löschende Zeichen), wenn wir mit dem Löschen eines Zeichens beginnen. Somit haben wir die Möglichkeit den alten Text zu modifizieren. In unserer Editiersitzung geben wir nacheinander die Zeichen '1','2' und '3' als Zeichenkette "123" ein. Dies ist in Abbildung 3.6 zu betrachten. Der Eingabetext wird in einem Textbuffer gehalten und ist somit noch nicht Bestandteil des abstrakten Syntaxbaumes, welcher die interne Darstellung des Programmes $\underline{EK_P}$ repräsentiert. Verlassen wir nun mit dem Cursor den Bereich im *Editierfenster*, welcher durch die dunklere Farbe besonders hervorgehoben ist, so wird die textuelle Eingabe als abgeschlossen betrachtet und ein Scanner aufgerufen, der die lexikalische Richtigkeit des im Textbuffer gehaltenen Eingabetextes überprüft. Dies ist notwendig, da der $\underline{Nutzer_{SBE(P)}}$ durch die textuelle Eingabe auch Syntaxfehler in das Programm $\underline{EK_P}$ einbringen könnte. Um diese zu vermeiden, wird eine fehlerhafte textuelle Eingabe vom Editor $\underline{SBE(P)}$ abgewiesen. Auch in unserem Fall wird die Eingabe "123" abgewiesen, da diese Zeichenkette keinen

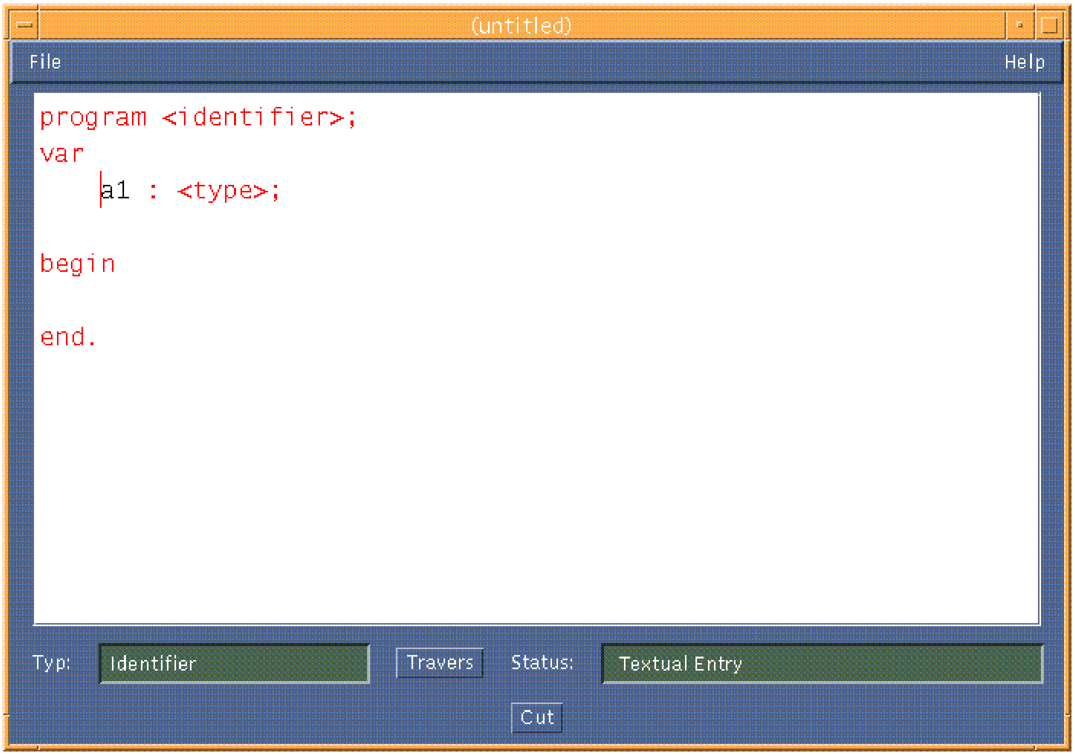

Abbildung 3.8: Screenshot

Bezeichner darstellt. Das Ergebnis dieses fehlgeschlagenen textuellen Eingabeversuches ist in Abbildung 3.7 zu sehen.
Wir versuchen nun erneut eine textuelle Eingabe, und zwar geben wir diesmal die Zeichenkette “a1” ein. Nachdem wir die Eingabe abschließen (durch Plazierung des Cursor außerhalb des ausgewählten Teils), erfolgt ein erfolgreiches Prüfen (Scannen) der Eingabe durch den $SBE(P)$. Anschließend wird die Eingabe in das EK_P eingepaßt. Das Ergebnis kann in Abbildung 3.8 betrachtet werden. Es ist zu beachten, daß der Cursor an den Anfang des Variablennamens, also zur alten Selection, zurückspringt. Der erschienene Kommandoknopf `Cut` kann zum Entfernen des Variablennamens und damit zur erneuten Verwendung des Platzhalters `<identifier>` verwendet werden.
Die Abbildung 3.9 zeigt das durch mehrere Editierbefehle erweiterte Programm, bei dem wir die Deklaration der Variablen `a1` vervollständigt haben. Außerdem haben wir durch strukturelle Transformation eine Anweisung eingefügt, und zwar die Schablone der bedingten Anweisung (*if-then-else*-Konstrukt). Diese Schablone besitzt drei Platzhalter: die Bedingung, den Anweisungsteil für die erfüllte sowie den Anweisungsteil für die nichterfüllte Bedingung. In den ersten Platzhalter haben wir textuell den Bezeichner `a1` eingetragen. Nachdem diese textuelle Eingabe ins Programm EK_P eingefügt worden ist, erscheint sofort die Fehlermeldung { `Boolean Expression Needed` } hinter dem Bedingungsteil. Der $Nutzer_{SBE(P)}$ erhält somit umgehend die Information, daß die als `integer` deklarierte Variable `a1` nicht als Bedingung in diesem Anweisungstyp benutzt werden darf. Allerdings bleibt es dem $Nutzer_{SBE(P)}$ überlassen ob er den Fehler sofort oder zu einem späteren

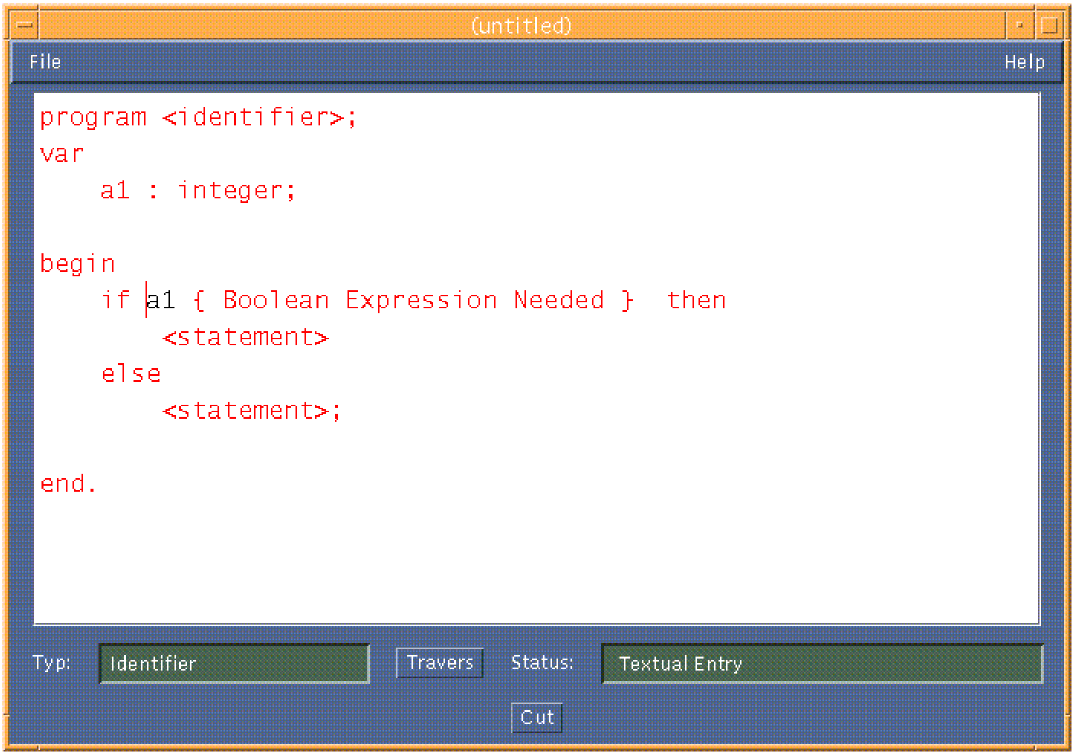

Abbildung 3.9: Screenshot

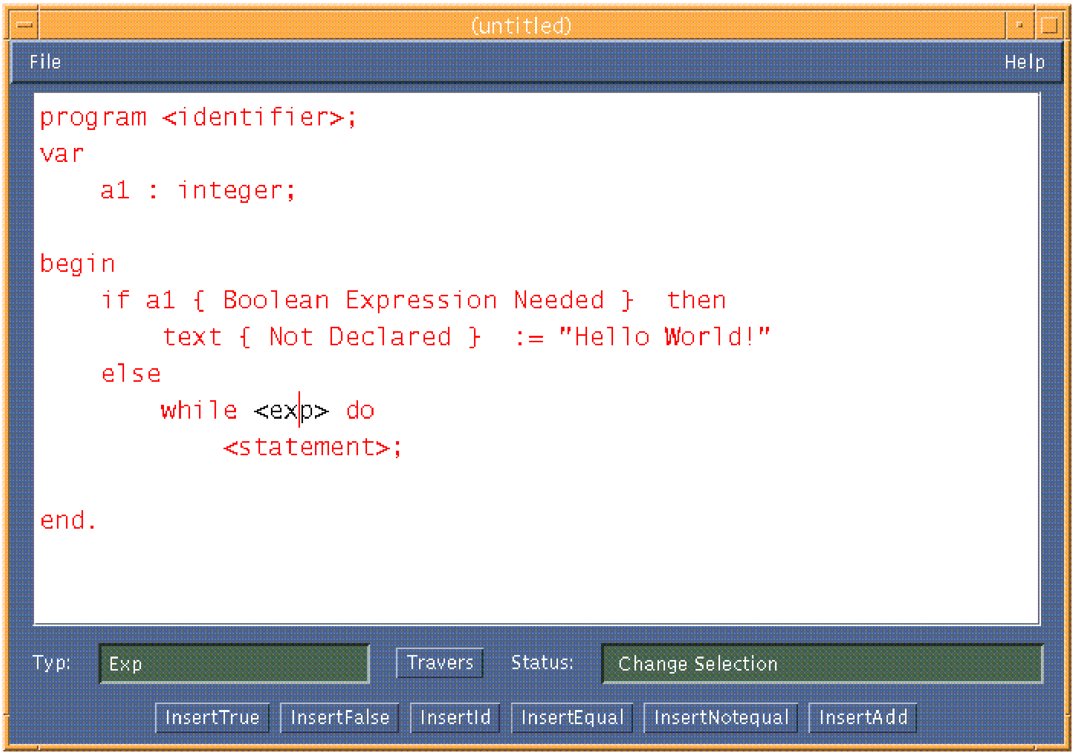

Abbildung 3.10: Screenshot

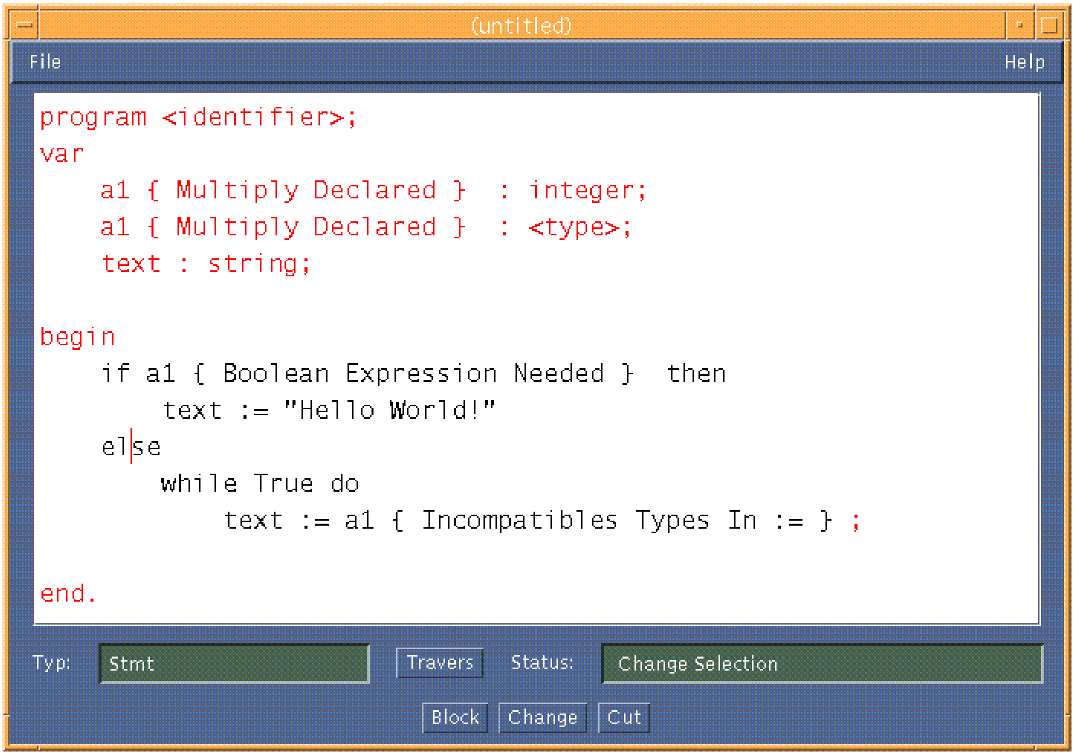

Abbildung 3.11: Screenshot

Zeitpunkt korrigiert.
Nachdem wir wiederum mehrere Editierbefehle an den $\underline{SBE(P)}$ gestellt haben, erhalten wir das Programm $\underline{EK_P}$, welches in Abbildung 3.10 gezeigt wird. Der zweite Platzhalter der bedingten Anweisung wurde von uns durch die Zuweisung instanziert, in der die undeklarierte Variable `text` als Wert einen String erhält. Der Editor $\underline{SBE(P)}$ weist durch die passende Fehlermeldung nach dem Erscheinen der Variablen `text` innerhalb der Zuweisung darauf hin, daß noch keine Deklaration für diese Variable in $\underline{EK_P}$ enthalten ist. Der dritte Platzhalter der bedingten Anweisung wurde von uns durch die Schablone einer While-Schleife ausgetauscht. Danach wurde der Platzhalter `<exp>`, der die noch unspezifizierte Abbruchbedingung in der While-Schleife darstellt, durch Plazierung des Cursors als aktuelle Auswahl im $\underline{EK_P}$ festgelegt. Man erkennt in Abbildung 3.10 eine Reihe von Kommandoknöpfen zur strukturellen Transformation des ausgewählten Platzhalters. Wenn wir nun die noch unspezifizierte Abbruchsbedingung in der While-Schleife mit dem Wert `True` instanzieren wollen, so haben wir sowohl die Möglichkeit, dies durch textuelle Eingabe zu erledigen, als auch durch Betätigen des Kommandoknopfes mit der Aufschrift `InsertTrue`. Hierbei sieht man, daß es auch möglich ist, dem $\underline{Nutzer_{SBE(P)}}$ gleichzeitig strukturelles und textuelles Editieren zu erlauben, um ihm mehr Freiheit beim Editieren von $\underline{EK_P}$ zu geben.
Die Abbildung 3.11 zeigt das $\underline{EK_P}$ nach einer weiteren Sequenz von Editierbefehlen. Zuerst wurde die Abbruchbedingung der While-Schleife mit dem Wert `True` instanziert. Es wurde danach eine weitere Deklaration der Variablen `a1` hinzugefügt, was zum Erscheinen der Fehlermeldungen über die Mehrfachdeklaration hinter den Variablenvorkommen

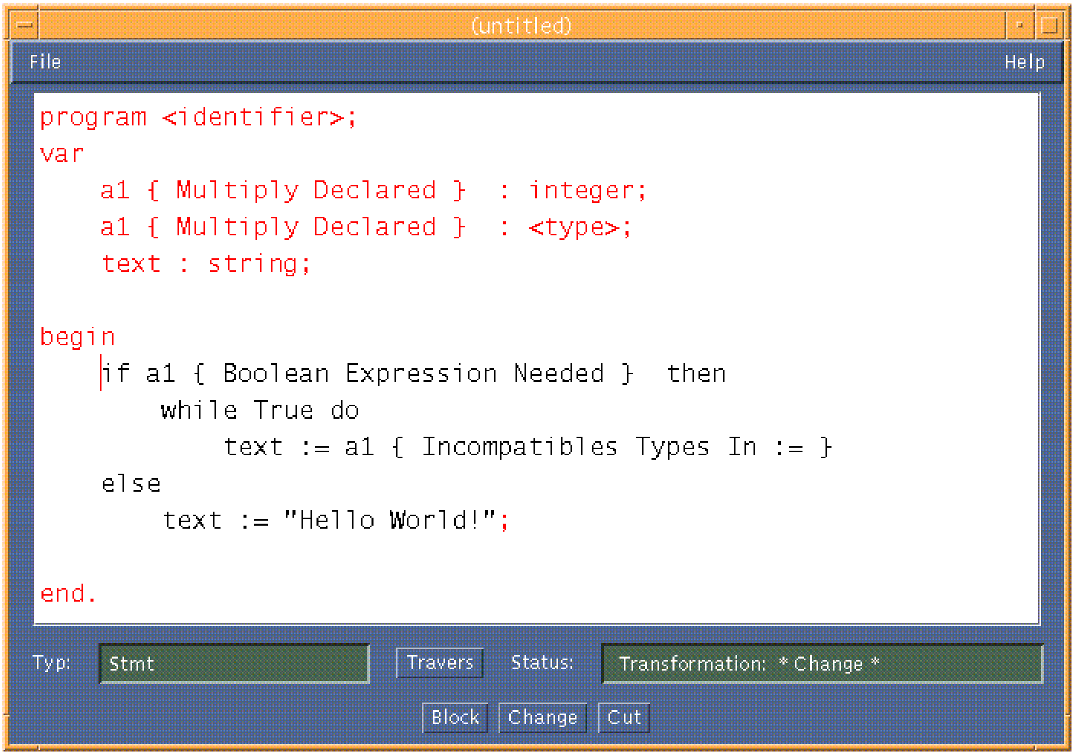

Abbildung 3.12: Screenshot

in den ersten zwei Deklarationen führte. Weiterhin wurde die Variable `text` als `string` deklariert, wonach die entsprechende Fehlermeldung über die Nichtdeklarierung im Anweisungsteil des $\underline{P}$-Programms $\underline{EK_P}$ verschwand. Im Anschluß daran wurde auch der Platzhalter im Anweisungsteil der While-Schleife durch eine unspezifizierte Anweisung ersetzt, welche dann textuell durch die Variablennamen `text` und `a1` instanziiert wurde. Da diese beiden Variablen nicht typkombatibel sind, gibt der $\underline{SBE(P)}$ eine entsprechende Fehlermeldung aus. Zum Schluß wurde durch Plazierung des Cursors mit der Maus die bedingte Anweisung (if-then-else-Konstrukt) zur weiteren Editation aktuell ausgewählt. Es werden uns nun drei Kommandoknöpfe zur strukturellen Transformation dieser Auswahl angeboten. Wir entscheiden uns für die Betätigung des mittleren, der die Aufschrift `Change` trägt. Durch ihn wird eine strukturelle Transformation ausgelöst, bei der nicht nur wie in den schon besprochenen Fällen ein Platzhalter durch eine Schablone ersetzt wird. Diesmal wird eine Schablone transformiert, wobei das Transformationsergebnis abhängig von der Belegung der Platzhalter in der Schablone ist. Es handelt sich hier also nicht nur um schlichtes Ersetzen. Konkret wird durch diese strukturelle Transformation die beiden Anweisungsteile (für die erfüllte bzw. nichterfüllte Bedingung) in der bedingten Anweisung vertauscht. Das Ergebnis dieser Transformation kann in Abbildung 3.12 betrachtet werden.
Damit soll auch unsere beispielhafte Editiersitzung enden.

Kapitel 4

Entwurf des Systems

In diesem Kapitel befassen wir uns mit dem Entwurf eines Systems zur Erzeugung von sprachbasierten Editoren. Dazu erstellen wir ein Konzept für den zu entwickelnden Generator $\underline{Gen_{SBE}}$ für sprachbasierte Editoren $\underline{SBE}$, indem wir uns an den Zielen orientieren, welche wir mit der Erstellung des $\underline{Gen_{SBE}}$ verfolgen. Daraus resultieren dann Überlegungen und Entscheidungen darüber, auf welche geeignete Weise diese Ziele erreicht werden können.

4.1 Zielstellung des Systems $\underline{Gen_{SBE}}$

Die Aufgabe des Generators $\underline{Gen_{SBE}}$ für sprachbasierte Editoren ist es, aus einer formalen Beschreibung $\underline{Spez_{SBE(L)}}$ eines sprachbasierten Editors der Sprache $\underline{L}$, einen auf dieser Sprache $\underline{L}$ basierenden Editor zu erzeugen.

Um ein Konzept zur Erzeugung eines solchen Generators zu erstellen, ist es notwendig, die gewünschte Funktionalität der zu generierenden Editoren $\underline{SBE(L)}$ zu untersuchen und daraus Anforderungen an die formale Beschreibung $\underline{Spez_{SBE(L)}}$ abzuleiten.

4.2 Komponenten eines sprachbasierten Editors $\underline{SBE(L)}$

Bei der Untersuchung der Funktionalität eines sprachbasierten Editors $\underline{SBE(L)}$ für die Sprache $\underline{L}$ wurde eine Trennung zwischen einer von der Sprache $\underline{L}$ abhängigen und einer von ihr unabhängigen Komponente vorgenommen.

4.2.1 Sprachunabhängige Komponente eines $\underline{SBE(L)}$

Zu den sprachabhängigen Editorfunktionen und Eigenschaften eines $\underline{SBE(L)}$ gehören in erster Linie das Vorhandensein einer geeigneten Oberfläche als Nutzerschnittstelle. Dem $\underline{Nutzer_{SBE(L)}}$ soll das von ihm mit dem $\underline{SBE(L)}$ editierte Konstrukt $\underline{EK_L}$ textuell dargeboten werden können. Er soll dabei die Möglichkeit haben, Teile aus $\underline{EK_L}$ zu selektieren, entweder durch Anklicken mit der Maus oder durch Kommandos zum Navigieren durch $\underline{EK_L}$. Dabei soll der aktuell selektierte Teil in der textuellen Darstellung von $\underline{EK_L}$ besonders hervorgehoben werden, damit der $\underline{Nutzer_{SBE(L)}}$ diesen erkennen kann. Das in $\underline{EK_L}$ aktuell selektierte Sprachkonstrukt aus $\underline{L}$ soll textuell oder strukturell editierbar sein. Bei einem textuellen Editiervorgang ist es notwendig, Text eingeben und löschen zu können. Für den strukturellen Editiervorgang sind geeignete Kommandoknöpfe zur Aktivierung einer strukturellen Transformation nötig (unabhängig von einer konkreten Sprache $\underline{L}$ ist hier das Ziel, daß ein Kommandoknopf mit einer bestimmten Aufschrift die dazugehörige strukturelle Transformation auslösen soll; von der konkreten Aufschrift und der konkreten Transformation wird hier abstrahiert). Natürlich werden auch Funktionen zum Speichern und Laden der editierten Konstrukte $\underline{EK_L}$ unter Verwendung eines Dateisystems notwendig sein. All diese Editorfunktionen sind unabhängig von der Sprache $\underline{L}$ und sollen vom $\underline{Gen_{SBE}}$ automatisch erzeugt werden. Die Person, welche den $\underline{SBE(L)}$ spezifiziert und den wir im folgenden mit $\underline{Entwickler_{SBE(L)}}$ abkürzen, soll sich also nicht um diese Aufgaben kümmern müssen, so daß er sich bei der Erstellung der $\underline{Spez_{SBE(L)}}$ nur auf die sprachabhängigen Eigenschaften von $\underline{SBE(L)}$ zu konzentrieren braucht.

Fazit:
Die sprachunabhängige Komponente eines mit dem $\underline{Gen_{SBE}}$ zu erzeugenden sprachbasierten Editors $\underline{SBE(L)}$ ist die Nutzerschnittstelle des $\underline{SBE(L)}$ in Form einer geeigneten Oberfläche. Sie ist nicht Inhalt der $\underline{Spez_{SBE(L)}}$.

4.2.2 Sprachabhängige Komponente eines $\underline{SBE(L)}$

Zur Realisierung der sprachabhängigen Editorfunktionen soll der sprachbasierte Editor $\underline{SBE(L)}$ über Wissen verfügen, welches sich auf die Sprache $\underline{L}$ bezieht. Dieses Wissen soll dem $\underline{Nutzer_{SBE(L)}}$ ein komfortables Editieren des $\underline{EK_L}$, also eines Satzes (Programmes) der Sprache $\underline{L}$, ermöglichen. Hier stellt sich die Frage, wie diese Kenntnisse des $\underline{SBE(L)}$ über die Sprache $\underline{L}$ zu beschreiben sind und in welche Spezifikationskomponenten die $\underline{Spez_{SBE(L)}}$ geeigneterweise modularisiert werden kann. Dazu betrachten wir noch einmal die gewünschten Eigenschaften des zu generierenden $\underline{SBE(L)}$:

Der $\underline{Nutzer_{SBE(L)}}$ soll in der Lage sein, strukturell zu editieren, also $\underline{EK_L}$ hierarchisch zu entwickeln, d.h. schon spezifizierte Sprachkonstrukte sukzessive zu verfeinern, und zwar durch schlichtes Betätigen eines dafür vorgesehenen Kommandoknopfes. So kann beispielsweise bei einer imperativen Programmiersprache $\underline{L}$ eine unspezifizierte Anweisung in eine While-Schleife verfeinert werden. Da hierbei dem $\underline{Nutzer_{SBE(L)}}$ die genaue Syntax der einzelnen Sprachkonstrukte, wie z.B. der While-Schleife, gar nicht bekannt sein muß, kann er so die Sprache $\underline{L}$ leicht bei der Erstellung erster eigener Programme erlernen.

Für die Realisierung der bis hier genannten Eigenschaften von $\underline{SBE(L)}$ bietet sich die Angabe der folgenden drei Spezifikationskomponenten an:

- **Abstrakte Syntax** Hiermit wird der syntaktische Aufbau der Sprache $\underline{L}$ beschrieben. Im Unterschied zur konkreten, kontextfreien Syntax wird in der abstrakten Syntax auf die Terminalsymbole verzichtet und die einzelnen Produktionen sind in der abstrakten Syntax benannt. Mit Hilfe der abstrakten Syntax läßt sich ein abstrakter Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ ($\underline{EK_L}$ als abstrakter Syntaxbaum der Sprache $\underline{L}$) konstruieren, welcher eine interne Darstellungsform des $\underline{EK_L}$ ist und sich geeignet bearbeiten läßt.

- **Ausgabesyntax** Durch die Ausgabesyntax wird festgelegt, auf welche Art und Weise der aktuell bearbeitete Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ textuell dem $\underline{Nutzer_{SBE(L)}}$ dargeboten werden soll. Hier erfolgt also die Beschreibung der Umwandlung von $\underline{EK_L < AbstrakteSyntax_L >}$ in $\underline{EK_L < Ausgabesyntax_L >}$ ($\underline{EK_L}$ als geeignete textuelle Darstellung in Relation zur Sprache $\underline{L}$).

- **Strukturelle Transformationen** Mit Hilfe von Regeln zur strukturellen Transformation des intern vorliegenden Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$ wird festgelegt, welche Sprachkonstrukte von $\underline{L}$ in welche anderen Sprachkonstrukte von $\underline{L}$ transformiert werden können und wie dies geschehen soll.

Außerdem soll, wie auch bei einfachen Texteditoren, eine textuelle Eingabemöglichkeit für ausgewählte Konstrukte der Sprache $\underline{L}$ existieren, bei denen sich eine strukturelle Editierung als nicht vorteilhaft oder gar unmöglich erweist, wie z.B. bei Zahlen und auch bei Bezeichnern für Variablen, Funktionsnamen etc..

Damit der Editor $\underline{SBE(L)}$ diese Fähigkeit besitzen kann, ist folgende Spezifikationskomponente erforderlich:

- **Eingabesyntax** Die Eingabesyntax dient zur Festlegung dafür, wie eine textuelle Eingabe des $\underline{Nutzer_{SBE(L)}}$ verarbeitet werden soll, um sie in den aktuell bearbeiteten $\underline{EK_L < AbstrakteSyntax_L >}$ korrekt einfügen zu können.

Weiterhin soll der $\underline{SBE(L)}$ nach jeder Modifikation von $\underline{EK_L}$ durch den $\underline{Nutzer_{SBE(L)}}$ eine selbständige Überprüfung aller kontextsensitiven Nebenbedingungen der Sprache $\underline{L}$ bei $\underline{EK_L}$ durchführen. Falls Verletzungen dieser Bedingungen aufgetreten sind, soll der $\underline{Nutzer_{SBE(L)}}$ geeignete Hinweise darüber erhalten. Die folgende Spezifikationskomponente eignet sich hierbei hervorragend zur Beschreibung der kontextsensitiven Nebenbedingungen von $\underline{L}$:

- **Attributierung** Hierbei erfolgt die Beschreibung der kontextsensitiven Nebenbedingungen der Sprache $\underline{L}$ durch die Festlegung einer geeigneten Attributierung des intern vorliegenden Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$.

Die fünf hier genannten Spezifikationskomponenten beschreiben die Sprache $\underline{L}$ hinreichend für den Zweck der Erzeugung eines auf dieser Sprache basierenden Editors $\underline{SBE(L)}$.

Fazit:
Die Aufgabe der sprachabhängigen Komponente eines mit dem $\underline{Gen_{SBE}}$ zu erzeugenden sprachbasierten Editors $\underline{SBE(L)}$ ist also die geeignete Verarbeitung bzw. Auswertung der genannten Spezifikationskomponenten, welche die Sprache $\underline{L}$ spezifizieren und die $\underline{Spez_{SBE(L)}}$ bilden.

Nachdem wir festgelegt haben, aus welchen Komponenten die Spezifikation $\underline{Spez_{SBE(L)}}$ eines sprachbasierten Editors $\underline{SBE(L)}$ geeigneterweise bestehen soll, stellen wir uns nun die Frage nach einer konkreten Sprache zur Beschreibung dieser Spezifikationskomponenten.

4.3 Spezifizierungssprache für $\underline{SBE(L)}$

Die von D. E. Knuth 1968 eingeführten Attributgrammatiken [Knu68] sind ein bewährtes Beschreibungsmittel im Bereich der formalen Sprachen. Sie können zur Lösung von vielfältigen Problemstellungen verwendet werden und kommen in der Praxis vor allem bei Übersetzungsaufgaben und zur Beschreibung der kontextsensitiven Nebenbedingungen von Programmiersprachen zum Einsatz.

Makroattributgrammatiken (siehe Kapitel 2) stellen eine Erweiterung von Attributgrammatiken dar, welche in der Hinzunahme von Kontextparametern besteht. Mit ihnen besitzt man ein Werkzeug zur Beschreibung der kontextfreien Syntax und der kontextsensitiven Abhängigkeiten einer formalen Sprache, wobei die kontextsensitiven Abhängigkeiten einer Sprache noch komfortabler spezifiziert werden können als bei Attributgrammatiken.

Um dieses Werkzeug nutzbar zu machen, wurde in einer in der Professur GdP (Grundlagen der Programmierung) stattfindenden Arbeitsgemeinschaft die Sprache Haskell$^+$ entworfen. Diese Sprache basiert auf der modernen, getypten funktionalen Sprache Haskell und erweitert diese um das Konzept der Makroattributgrammatiken. Dabei ist Haskell$^+$ eine Obermenge von Haskell, was bedeutet, daß jedes Haskell-Programm auch ein Haskell$^+$-Programm ist. Wie man mit Hilfe der Sprache Haskell$^+$ Makroattributgrammatiken spezifizieren kann, wird in [Les99] erklärt.

Ein Ziel in der Entwicklung der Sprache Haskell$^+$ lag darin, eine Möglichkeit zu erhalten, komfortabel Macro Attributed Tree Transducer [KV94] spezifizieren zu können. Macro Attributed Tree Transducer sind ein Spezialfall der Makroattributgrammatiken, da sie ausschließlich auf symbolischen Daten operieren. Ein weiteres Hauptziel war die Erzeugung einer Eingabesprache für ein System zur Erzeugung von sprachbasierten Editoren. Daher besitzt sie auch die notwendigen Möglichkeiten zur Festlegung der Spezifikationskomponenten für die **Abstrakte Syntax**, die **Attributierung**, die **Ausgabesyntax** und die **Strukturellen Transformationen**. Diese Spezifikationskomponenten werden mit Hilfe von Makroattributgrammatiken beschrieben. Es existiert jedoch zum jetzigen Zeitpunkt nur sehr eingeschränkt die Möglichkeit, eine **Eingabesyntax** zu spezifizieren. Dies resultiert in der Hauptsache aus dem Fakt, daß man in Haskell$^+$ keine regulären Ausdrücke beschreiben kann. Weiterhin fehlt auch die Möglichkeit einer Spezifizierung, ob ein Sprachkonstrukt der Sprache $\underline{L}$ mit dem $\underline{SBE(L)}$ textuell editiert werden darf oder nicht. Da sich Haskell$^+$ aber noch in der Entwicklung befindet, sind diesbezügliche Spracherweiterungen von Haskell$^+$ schon abzusehen.
In der vorliegenden Arbeit legen wir uns aus den genannten Gründen auf die Sprache Haskell$^+$ als Spezifizierungssprache für sprachbasierte Editoren fest. In Kapitel 5 beschreiben wir dann den Aufbau einer Spezifikation $\underline{Spez_{SBE(L)}}$ für einen $\underline{SBE(L)}$ und erklären

somit dem $\underline{Entwickler_{SBE(L)}}$, was er bei der Erstellung einer $\underline{Spez_{SBE(L)}}$ zu beachten hat. Dabei gehen wir auch ausführlich auf die $\underline{Spez_{SBE(P)}}$ des in Kapitel 3 vorgestellten sprachbasierten Editors $\underline{SBE(P)}$ ein.

4.4 Zwischenzusammenfassung

An dieser Stelle fassen wir zunächst einmal die Generierung und die Nutzung eines sprachbasierten Editors zusammen. In Abbildung 4.1 sind diese Abläufe schematisch dargestellt. Die Bedeutung der graphischen Elemente sowie die in dieser Abbildung verwendeten Abkürzungen werden alle nochmal in der Legende erläutert, die sich im Anhang B der Arbeit befindet.

Der $\underline{Entwickler_{SBE(L)}}$ (den wir in der Abbildung mit **II** bezeichnen) programmiert die $\underline{Spez_{SBE(L)} < Haskell^+ >}$ ($\underline{Spez_{SBE(L)}}$ in der Sprache Haskell$^+$ beschrieben), welche aus den einzelnen Komponenten zur Festlegung der **Abstrakten Syntax**, der **Attributierung**, der **Ausgabesyntax**, der **Strukturellen Transformationen** und der **Eingabesyntax** besteht.

Das System $\underline{Gen_{SBE}}$ wurde vorher vom $\underline{Entwickler_{Gen_{SBE}}}$ (die Person, welche das System $\underline{Gen_{SBE}}$ implemetiert hat und in der Abbildung mit **I** abgekürzt wird) programmiert. Das so vorhandene Generator $\underline{Gen_{SBE}}$ erhält die $\underline{Spez_{SBE(L)} < Haskell^+ >}$ als Eingabe und erzeugt daraus die sprachabhängige Komponente des $\underline{SBE(L)}$. Außerdem wird die sprachunabhängige Komponente des $\underline{SBE(L)}$, also die Oberfläche des $\underline{SBE(L)}$ als Schnittstelle zum $\underline{Nutzer_{SBE(L)}}$ (welcher in der Abbildung mit **III** abgekürzt wird), vom $\underline{Gen_{SBE}}$ automatisch generiert. Beide Komponenten werden so erzeugt, daß sie zusammen einen funktionstüchtigen $\underline{SBE(L)}$ ergeben.

Der so erzeugte $\underline{SBE(L)}$ steht nun dem $\underline{Nutzer_{SBE(L)}}$ zur Verfügung:

Der $\underline{Nutzer_{SBE(L)}}$ kann mit Hilfe des $\underline{SBE(L)}$ ein Konstrukt $\underline{EK_L}$ der Sprache $\underline{L}$ erzeugen. Dies erreicht er durch die Anwendung einer Folge von dazu geeigneten Editierbefehlen, mit denen er das editierte Konstrukt $\underline{EK_L}$ als $\underline{EK_L < Editbefehl^+ >}$ ($\underline{EK_L}$ betrachtet als Sequenz von getätigten Editierbefehlen) beschreibt. Gewisse Editierbefehle führen zu einer neuen aktuellen Auswahl eines Teils von $\underline{EK_L}$ und andere verändern den ausgewählten Teil textuell oder strukturell. Die Editierbefehle werden von der Oberfläche des $\underline{SBE(L)}$ entgegengenommen.

Während des Editierens von $\underline{EK_L}$ durch den $\underline{Nutzer_{SBE(L)}}$ wird das $\underline{EK_L}$ von dem $\underline{SBE(L)}$ intern stets als Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ repräsentiert. Nach jedem Editierbefehl der strukturellen oder textuellen Art vom $\underline{Nutzer_{SBE(L)}}$ muß daher der Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ wie gewünscht geändert werden.

Wie dieses bei einem textuellen Editierbefehl erfolgt, wird durch die Komponente der $\underline{Spez_{SBE(L)} < Haskell^+ >}$, welche die **Eingabesyntax** spezifiziert, beschrieben. (Es sei hier daran erinnert, daß in Haskell$^+$ die Möglichkeit der Beschreibung einer **Eingabesyntax** noch sehr beschränkt ist. Es können daher in der $\underline{Spez_{SBE(L)} < Haskell^+ >}$ Datentypen, die für die Sprache Haskell$^+$ vordefiniert sind, also nicht in $\underline{Spez_{SBE(L)} < Haskell^+ >}$ beschrieben werden müssen bzw. können, zur Realisierung von textuellen Eingabemöglichkeiten verwendet werden, worauf wir in Kapitel 4.5 noch genauer eingehen)

Die Realisierung von strukturellen Editierbefehlen wird in der Spezifikationskomponente für die **Strukturellen Transformationen** festgelegt.
Die Struktur des Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$ wird durch die Spezifikationskomponente von $\underline{Spez_{SBE(L)} < Haskell^+ >}$ zur Beschreibung der **Abstrakten Syntax** definiert.
Die Spezifikationskomponente zur Beschreibung der **Attributierung** definiert die synthetischen und inheriten Funktionen, die auf den Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ angewendet werden, um kontextsensitive Nebenbedingungen der Sprache $\underline{L}$ in $\underline{EK_L}$ abzutesten. Die Ergebniswerte der Berechnungen dieser syntaxbasierten Funktionen können dann in die Umwandlung der internen Darstellung $\underline{EK_L < AbstrakteSyntax_L >}$ in die externe Darstellung $\underline{EK_L < Ausgabesyntax_L >}$ miteinbezogen werden. Auf diese Weise kann der $\underline{Nutzer_{SBE(L)}}$ über Fehler in den kontextsensitiven Nebenbedingungen in $\underline{EK_L}$ informiert werden. Nach jeder Modifikation von $\underline{EK_L < AbstrakteSyntax_L >}$ erfolgt eine sofortige Umwandlung dieses Syntaxbaumes in die Darstellung $\underline{EK_L < Ausgabesyntax_L >}$. Hierbei müssen natürlich auch diejenigen synthetischen und inheriten Funktionen neu berechnet werden, deren Ergebniswerte Auswirkungen auf diese Umwandlung haben, damit ein vollständig aktualisiertes $\underline{EK_L < Ausgabesyntax_L >}$ dem $\underline{Nutzer_{SBE(L)}}$ dargeboten werden kann.
Wie die Umwandlung von $\underline{EK_L < AbstrakteSyntax_L >}$ in $\underline{EK_L < Ausgabesyntax_L >}$ konkret realisiert werden soll, wird in der Komponente von $\underline{Spez_{SBE(L)} < Haskell^+ >}$ festgelegt, welche die **Ausgabesyntax** definiert. Die Oberfläche des $\underline{SBE(L)}$ übernimmt dann die Darstellung von $\underline{EK_L < Ausgabesyntax_L >}$, wobei der aktuell ausgewählte Teil von $\underline{EK_L}$ besonders hervorgehoben wird.

Es sei nochmal darauf hingewiesen, daß beim Editieren von $\underline{EK_L}$ durch den $\underline{Nutzer_{SBE(L)}}$ nach jedem Auslösen eines Editierbefehls die aktuelle Auswahl in $\underline{EK_L}$ oder der Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ geändert wird und danach eine Umwandlung dieses Syntaxbaums in die Darstellungsform $\underline{EK_L < Ausgabesyntax_L >}$ nötig ist. Dies erfordert wiederum die Berechnung von synthetischen und inheriten Funktionen, welche ihre Ergebniswerte zu dieser Umwandlung beisteuern. In dieser Form kommt hier das Konzept der sofortigen Berechnung zum Einsatz.
Eine Kombination mit dem Konzept der inkrementellen Berechnung durch Verwendung von inkrementellen Auswertungsstrategien für die Berechnung der syntaxbasierten Funktionen wäre hier der Effizienz wegen anzuraten. In der vorliegenden Arbeit wird aber darauf verzichtet.

Der $\underline{Entwickler_{Gen_{SBE}}}$, der $\underline{Entwickler_{SBE(L)}}$ und der $\underline{Nutzer_{SBE(L)}}$ werden mit **I**, **II** bzw. **III** bezeichnet, da der $\underline{Entwickler_{Gen_{SBE}}}$ über das detaillierteste Wissen verfügen muß. Von ihm wird verlangt, daß er weiß, wie man eine beliebige formale Sprache geeignet beschreiben kann, um ein System zur Erzeugung eines auf dieser Sprache basierenden Editors zu konzipieren. Vom $\underline{Entwickler_{SBE(L)}}$ wird im Vergleich dazu nur genügend Wissen über eine konkrete formale Sprache $\underline{L}$ gefordert, damit er die Erstellung der $\underline{Spez_{SBE(L)}}$ als Eingabe des vom $\underline{Entwickler_{Gen_{SBE}}}$ entwickelten Systems realisieren kann. Der $\underline{Nutzer_{SBE(L)}}$ benötigt am wenigsten Wissen, er muß die konkrete Sprache $\underline{L}$ nur insoweit kennen, als daß er ein korrektes Programm der Sprache $\underline{L}$ mit Unterstützung des vom $\underline{Entwickler_{SBE(L)}}$ spezifizierten $\underline{SBE(L)}$ erstellen kann.

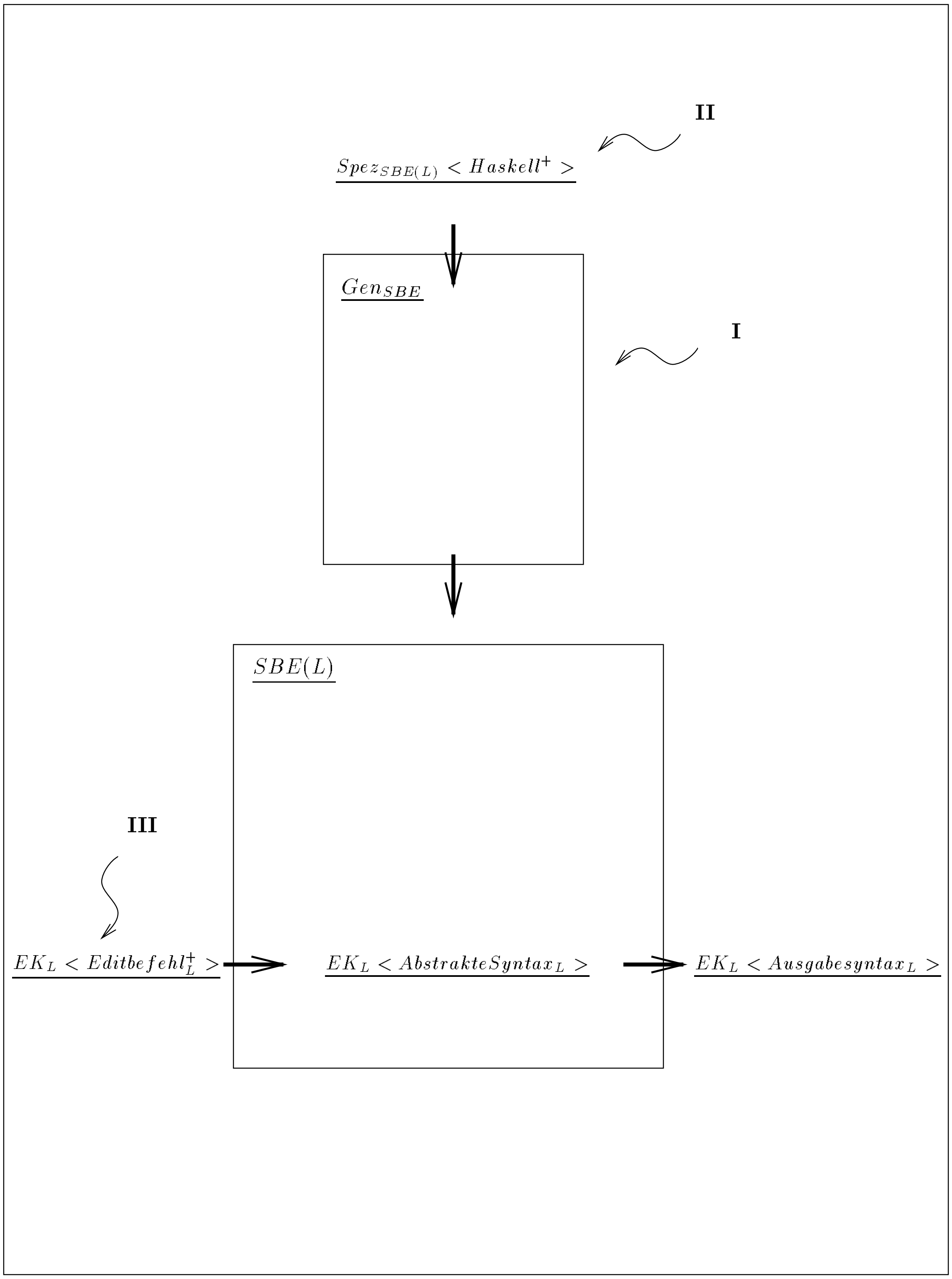

Abbildung 4.1: Schema der Generierung und Nutzung von $\underline{SBE(L)}$

4.5 Realisierung der Komponenten eines $\underline{SBE(L)}$

In diesem Abschnitt betrachten wir die Realisierung der Erzeugung eines $\underline{SBE(L)}$ in Hinsicht auf deren programmtechnische Umsetzung. Dies erfolgt wieder getrennt für die sprachabhängige (Auswerter) und die sprachunabhängige Komponente (Nutzerschnittstelle) des zu erzeugenden $\underline{SBE(L)}$.

4.5.1 Realisierung des Auswerters eines $\underline{SBE(L)}$

Die sprachabhängige Komponente eines $\underline{SBE(L)}$ ist der Teil, welcher durch die $\underline{Spez_{SBE(L)}}$ beschrieben wird. Die Spezifikation $\underline{Spez_{SBE(L)}}$ eines sprachbasierten Editors $\underline{SBE(L)}$ ist ein Haskell$^+$-Programm, welches aus einer oder mehreren Makroattributgrammatiken zur Beschreibung der Sprache $\underline{L}$ besteht. Derzeit existiert aber kein Compiler zur direkten Übersetzung eines solchen Haskell$^+$-Programmes in ein Maschinensprachprogramm. Da die Erstellung eines solchen Compilers im Rahmen dieser Diplomarbeit zu aufwendig wäre, haben wir uns für die Übersetzung in eine Programmierhochsprache entschieden, für die bereits ein Compiler existiert.

In der vorliegenden Arbeit wurde dafür aus folgenden Gründen die funktionale Sprache Haskell ausgewählt:

Unsere Spezifikationssprache Haskell$^+$ umfaßt die Sprache Haskell und ist an diese angelehnt. Daher müssen bei der Übersetzung eines Haskell$^+$-Programmes in ein Haskell-Programm nur die Programmteile übersetzt werden, die nicht schon Bestandteil der Sprache Haskell sind.
Ein weiterer Grund für die Entscheidung, die Haskell$^+$-Programme nach Haskell zu übersetzen, liegt darin, daß sich eine funktionale Sprache wie Haskell sehr gut zur Programmierung von abstrakten Problemen eignet, in denen auf Symbolen gerechnet wird. Im Gegensatz zu imperativen Programmiersprachen ist bei einer funktionalen Programmiersprache die funktionale Beschreibung eines Problems für dessen rechnergestützte Lösung ausreichend.
In [Les99] wird für Haskell$^+$-Programme ein Scanner und ein Parser zum Aufbau abstrakter Syntaxbäume vorgestellt, sowie deren Übersetzung nach Haskell. Diese Übersetzung kann jedoch die Anforderungen zur Erzeugung eines sprachbasierten Editors $\underline{SBE(L)}$ nicht vollständig erfüllen.
Ziel unserer Arbeit ist es also nun, die in [Les99] beschriebene Übersetzung so zu vervollständigen bzw. anzupassen, daß aus den Makroattributgrammatiken zur Beschreibung der Sprache $\underline{L}$ solche Haskell-Funktionen generiert werden können, die uns die sprachabhängige Komponente des zu erzeugenden sprachbasierten Editors $\underline{SBE(L)}$ realisieren.
Diese Haskell-Funktionen implementieren dann konkret einen geeigneten Mechanismus zur Berechnung der Instanzwerte der Attribute aus den Attributbeschreibungen der Makroattributgrammatiken.

4.5.2 Realisierung der Nutzerschnittstelle eines $\underline{SBE(L)}$

Wir besprechen in diesem Abschnitt die Generierung der sprachunabhängigen Komponente der zu erzeugenden sprachbasierten Editoren. Hierbei geht es, wie schon besprochen, um die Generierung einer geeigneten fensterorientierten Oberfläche als Schnittstelle zum $\underline{Nutzer_{SBE(L)}}$ des sprachbasierten Editors $\underline{SBE(L)}$. Die Oberfläche soll ein ereignisgesteuertes Verhalten aufweisen. Als Ereignis in diesem Sinn sind hier die Betätigung von Tastatur und Maus durch den $\underline{Nutzer_{SBE(L)}}$, in bezug auf die verschiedenen Oberflächenkomponenten Editierfenster, Kommmandoknöpfe etc. zu verstehen. In Abhängigkeit von diesen Ereignissen soll das editierte Konstrukt $\underline{EK_L}$ entsprechend verändert werden. Konkret erfordert dies eine Änderung der aktuellen Auswahl in $\underline{EK_L}$ oder die Transformation des Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$ als interne Darstellung des editierten Konstruktes $\underline{EK_L}$ und damit eine neue Auswertung dessen Attributinstanzen.

An dieser Stelle muß eine Implementierungssprache für das Problem der Generierung der Oberfläche eines sprachbasierten Editors $\underline{SBE(L)}$ ausgewählt werden. Folgende Varianten wurden dafür untersucht:

<u>Haskell und deren Aufsatzmodule Fudgets und Haggis :</u>

Wie die meisten heutzutage existierenden funktionalen Sprachen eignet sich Haskell nicht zur Programmierung einer fensterorientierten, ereignisgesteuerten Oberfläche. Allerdings existieren die Aufsatzmodule Fudgets und Haggis, die für diesen Zweck implementiert worden sind. Trotzdem bieten sie nicht genügend Unterstützung für die Erzeugung einer Oberfläche, die in ihrer Funktionalität unseren Anforderungen entspricht. Mit beiden Spracherweiterungen gelingt es z.B. nicht, ein mehrzeiliges Editierfenster zu generieren, das gleichzeitig mehrere Textformate zuläßt. Dies ist aber notwendig, um das $\underline{EK_L}$ so textuell darstellen zu können, daß der $\underline{Nutzer_{SBE(L)}}$ den aktuell ausgewählten Teil von $\underline{EK_L}$ von der restlichen textuellen Darstellung unterscheiden kann. Da die beiden Spracherweiterungen Fudgets und Haggis seit 1996 nicht mehr weiterentwickelt werden und auch kein Support existiert, kommt eine Lösung mit diesen Erweiterungen der Sprache Haskell nicht in Betracht.

Fazit: Die Sprache Haskell eignet sich nicht zur Generierung der sprachunabhängigen Editorkomponenten. Damit steht fest, daß diese Aufgabe in einer anderen Sprache implementiert werden muß. Damit muß man den Nachteil in Kauf nehmen, daß die Generierung der sprachabhängigen und die der sprachunabhängigen Komponente eines zu erzeugenden sprachbasierten Editors $\underline{SBE(L)}$ in zwei verschiedenen Implementierungssprachen programmiert werden muß.

<u>Java (Java Development Kit 1.2) :</u>

Das JDK 1.2 verfügt über die gewünschte Funktionalität zur Erzeugung einer ereignisgesteuerten fensterorientierten Oberfläche. Java ist eine sehr moderne und heutzutage in der Praxis vielfach verwendete Sprache, deren objektorientiertes Konzept sich sehr gut zur Programmierung von ereignisgesteuerten Anwendungen eignet. Außerdem hat sie den Vorteil, unabhängig von einer konkreten Rechnerplattform zu sein. Allerdings ist es nicht möglich, in einem Java-Programm externe Haskell-Funktionen aufzurufen. Dies erschwert die Integration der beiden Sprachen Java und Haskell.

C++ :

Die objektorientierte Sprache C++ wurde ebenfalls als Variante in Betracht gezogen. Es sind hier aber gegenüber Java in bezug auf unser Problem keine signifikanten Vorteile zu nennen.

Es konnte hier keine optimale Variante gefunden werden, so daß wir uns für Java als Implementierungssprache zur Erzeugung der Oberfläche und damit der sprachunabhängigen Komponente unserer sprachbasierten Editoren $\underline{SBE(L)}$ als bestmögliche Variante entschieden haben.

An dieser Stelle gilt mein besonderer Dank Herrn Eikenberg, der mich bei der Auswahl dieser Implementierungssprache im Rahmen seiner Tätigkeit als studentische Hilfskraft wesentlich unterstützt hat.

4.5.3 Integration der Komponenten eines $\underline{SBE(L)}$

In diesem Abschnitt behandeln wir das Problem der Integration des Haskell- und des Java-Codes zu einem sprachbasierten Editor $\underline{SBE(L)}$.

Wie schon erwähnt, existiert leider zur Zeit keine nutzbare Möglichkeit der Einbindung von Haskell-Code in ein Java-Programm. Ebenfalls läßt sich Java-Code nicht in ein Haskell-Programm einbinden. Die Möglichkeit, den Haskell-Code nach Java zu übersetzen, wurde aus Zeit- und Aufwandsgründen im Rahmen dieser Diplomarbeit verworfen, da kein verwendbarer Übersetzer existiert und dieser so erst erstellt werden müßte.

Die Lösung des Problems wurde darin gefunden, daß aus den entsprechenden Haskell- und Java-Programmcodes mit den dazugehörigen Compilern lauffähige Programme erzeugt werden. Diese Programme werden dann zur gleichen Zeit abgearbeitet. Dabei kommunizieren sie über das Dateisystem des Rechners, auf dem sie ausgeführt werden. Dadurch können sie in Zusammenarbeit die Funktionalität des sprachbasierten Editors $\underline{SBE(L)}$ sicherstellen, den sie implementieren.

Die Kommunikation der beiden Programme über das Dateisystem funktioniert wie folgt:

Jedes der beide Programme schreibt Informationen in dafür vorgesehene Dateien, während das jeweils andere Programm das Dateisystem auf die Aktualität dieser Dateien überprüft und in Abhängigkeit davon die darin enthaltenen Informationen ausliest. Das Java-Programm gibt so Informationen über die vom $\underline{Nutzer_{SBE(L)}}$ ausgelösten Editierbefehle an das Haskell-Programm, während dieses Informationen über die geforderte Darstellung der Oberfläche, in Abhängigkeit des von ihm bearbeiteten $\underline{EK_L}$, für das Java-Programm bereitstellt.

4.6 Beschreibung des Systems $\underline{Gen_{SBE}}$ in Phasen bezüglich der verwendeten Funktionsmodule

Wir gehen nun nocheinmal konkreter auf die Generierung eines sprachbasierten Editors $\underline{SBE(L)}$ aus seiner Spezifikation $\underline{Spez_{SBE(L)}}$ ein, wobei wir sie in Phasen aufteilen. Dabei beschreiben wir die einzelnen Übersetzungsphasen im Hinblick auf verwendete Programmodule und deren Funktionalität. Der in Phasen aufgeteilte Generierungsablauf ist dazu in Abbildung 4.2 zu betrachten, welche die Abbildung 4.1 spezialisiert. Die einzelnen Übersetzungsphasen sind in Abbildung 4.2 mit P1 bis P6 bezeichnet. Weiterhin ist in dieser Abbildung der auf der Sprache $\underline{L}$ basierende Editor $\underline{SBE(L)}$ in die Komponenten Auswerter und Nutzerschnittstelle unterteilt. Die Kommunikation der beiden Komponenten über das Dateisystem ist angedeutet.

Innerhalb der Phasen 1, 2 und 3 wird der Haskell$^+$-Code $\underline{Spez_{SBE(L)} < Haskell^+ >}$ in den Haskell-Code $\underline{Spez_{SBE(L)} < Haskell >}$ ($\underline{Spez_{SBE(L)}}$ als Haskell-Programm) übersetzt. Dies erfolgt über den Schritt der Generierung eines geeigneten Zwischenformates, und zwar eines Syntaxbaumes $\underline{Spez_{SBE(L)} < AbstrakteSyntax_{Haskell^+} >}$ ($\underline{Spez_{SBE(L)}}$ als abstrakter Syntaxbaum der Sprache Haskell$^+$) in Phase 1. Auf diesen Syntaxbaum werden dann in Phase 2 Analysen durchgeführt. In Phase 3 erfolgt dann die Übersetzung des nun vorliegenden Zwischenformates $\underline{Spez_{SBE(L)} < AbstrakteSyntax_{Haskell^+} >}$ nach Haskell. Die drei Phasen werden in Haskell realisiert. Die grobe Steuerung übernimmt dabei das Haskell-Modul **HPC**. Das Haskell-Modul **Syntaxtr** beschreibt die abstrakte Syntax von Haskell$^+$ für die Verwendung bei $\underline{Spez_{SBE(L)} < AbstrakteSyntax_{Haskell^+} >}$.

P1 : Begonnen wird mit dem Scannen des Haskell$^+$-Codes als String durch das Haskell-Modul **Scanner.** Dabei erfolgt die Erzeugung des letztgenannten Moduls durch das Haskell-Tool Alex, einem Scanner-Generator. Das Modul **Scanner** verwendet das Haskell-Modul **Tokens** zur Spezifizierung der beim Scannen zu verwendenden Token. Weiterhin wird das Haskell-Modul **Alex** zur Unterstützung des Scanvorganges benutzt. Danach erfolgt das Parsen durch das Haskell-Modul **Parser**, welches durch den Parser-Generator Happy, einem Haskell-Tool, erstellt wurde. Zusätzlich in dieser Phase verwendete Hilfsfunktionen sind im Haskell-Modul **Layout** zu finden.

Die für Phase 1 benötigten Haskell-Module wurden von Herrn Lescher in Rahmen seiner Arbeit [Les99] erstellt.

P2 : Auf dem vorliegenden Zwischenformat $\underline{Spez_{SBE(L)} < AbstrakteSyntax_{Haskell^+} >}$ erfolgt hier zuerst ein Testen der kontextsensitiven Nebenbedingungen der Spezifikation $\underline{Spez_{SBE(L)}}$ durch das Haskell-Modul **CsrTests**. Anschließend beginnt ein Testen der Nicht-Zirkularität der Makroattributgrammatiken, die sich in der Spezifikation $\underline{Spez_{SBE(L)}}$ befinden, durch das Haskell-Modul **Circtest**. Abschließend seien die Haskell-Module **Types**, welches hauptsächlich für die Generierung von Fehlermeldungen während der Testabläufe verantwortlich ist, sowie **AuxFuns** und **MoreTest** genannt. Die beiden letztgenannten Haskell-Module bieten Hilfsfunktionen zur Arbeit auf dem Syntaxbaum $\underline{Spez_{SBE(L)} < AbstrakteSyntax_{Haskell^+} >}$ an.

Die in Phase 2 verwendeten Haskell-Module stammen ebenfalls aus [Les99].

P3 : Hier wird die Generierung des Haskell-Codes $\underline{Spez_{SBE(L)} < Haskell >}$ aus dem Zwischenformat $\underline{Spez_{SBE(L)} < AbstrakteSyntax_{Haskell^+} >}$ vorgenommen. Die Steuerung dieser Generierung sowie Teile der Übersetzung selbst, welche die abstrakten Datentypen und die synthetischen bzw. inheriten Funktionen aus $\underline{Spez_{SBE(L)}}$ betreffen, übernimmt das Haskell-Modul **Translat**. Die Generierung der Funktionen zur Realisierung der Ausgabesyntax beim $\underline{SBE(L)}$, realisert das Haskell-Modul **GenUnparse**. Die Funktionen zur strukturellen und zur textuellen Transformation des $\underline{EK_L}$ werden durch das Haskell-Modul **GenTransform** generiert. Das Haskell-Modul **GenTravers** erzeugt die rekursiven Funktionen zum Durchlaufen des Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$, die benötigt werden, wenn der $\underline{Nutzer_{SBE(L)}}$ beim Editieren des $\underline{EK_L}$ einen konkreten Teil des $\underline{EK_L}$ auswählen möchte.

Die hier verwendeten Haskell-Module wurden im Rahmen der vorliegenden Diplomarbeit erstellt. Das Haskell-Modul **Translat** stammt zwar ursprünglich aus [Les99], wurde aber hier notwendigerweise angepaßt bzw. umprogrammiert. Die Implementierung dieser Haskell-Module (grundlegende Herangehensweise) wird in Abschnitt 6.1 beschrieben.

In den Phasen 4 und 5 wird jeweils Haskell-Code mit dem Glasgow-Haskell-Compiler in ausführbaren Code übersetzt. Die entstehenden Teile werden zusammengelinkt und ergeben so den Teil des zu generierenden sprachbasierten Editors $\underline{SBE(L)}$, welcher für die Auswertung der Makroattributgrammatiken zuständig ist, durch die die Sprache $\underline{L}$ beschrieben wurde.

P4 : Der in den ersten drei Phasen erzeugte Haskell-Code $\underline{Spez_{SBE(L)} < Haskell >}$ liegt in diesem Moment als Haskell-Modul **Evaluator** vor. Dieses Modul wird nun dazu verwendet, einen Teil des zu erzeugenden sprachbasierten Editor $\underline{SBE(L)}$ zu erschaffen. Dieser Teil ist der Bereich der Auswerterkomponente von $\underline{SBE(L)}$, welcher abhängig von der Sprache $\underline{L}$ ist, auf welcher der Editor $\underline{SBE(L)}$ basiert.

P5 : In dieser Phase wird der universell verwendbare Bereich der Auswerterkomponente des zu erzeugenden sprachbasierten Editors $\underline{SBE(L)}$ erzeugt. Dieser ergibt sich aus der Übersetzung des Haskell-Codes $\underline{UC_{AW} < Haskell >}$ (universeller Code in Haskell für die Auswerterkomponente), welcher die folgenden Haskell-Module umfaßt. Die Haskell-Module **UserIntInputScanner**, **UserBoolInputScanner**, **UserCharInputScanner**, **UserQuotedStringInputScanner**, **UserIdentInputScanner** sowie **UserQuotedStringInputScanner** realisieren das Scannen der für Haskell$^+$ vordefinierten Datentypen *IntInput*, *BoolInput*, *CharInput*, *StringInput IdentInput* und *QuotedStringInput*. Diese Datentypen werden kurz in Abschnitt 5.5 und ausführlich in Abschnitt 6.2.2 beschrieben. Die Scanvorgänge sind im Rahmen der textuellen Editiermöglichkeiten des $\underline{Nutzer_{SBE(L)}}$ notwendig. Die Erzeugung der sechs letztgenannten Haskell-Module erfolgte mit dem Scanner-Generator Alex. Das Haskell-Modul **UserInput** definiert die beim Scannen verwendeten Token. Außerdem ist für das Scannen wieder das Haskell-Modul **Alex** notwendig.
Als Quellcode der Übersetzungsphase 5 dient weiterhin das Haskell-Modul **InterChange**. In diesem Modul finden sich die Funktionen zur Realisierung der Kommu-

nikation der Auswerterkomponente des $\underline{SBE(L)}$ mit der Nutzerschnittstellenkomponente des $\underline{SBE(L)}$. In einem weiteren Haskell-Modul **HelpFuns** sind universelle Hilfsfunktionen für die Auswerterkomponente des zu erzeugenden $\underline{SBE(L)}$ enthalten. Als letztes sei das Haskell-Modul **Main** genannt, welches die main-Funktion für die Auswerterkomponente enthält.

Die Haskell-Module, die als Quellcode für diese Übersetzungsphase dienen, wurden im Rahmen dieser Diplomarbeit implementiert und werden in Abschnitt 6.2 näher erklärt.

In Phase 6 erfolgt die Generierung einer geeigneten Oberfläche des zu erzeugenden $\underline{SBE(L)}$ als Schnittstelle zum $\underline{Nutzer_{SBE(L)}}$. Der für diese Übersetzungsphase notwendige Quell-Code wurde in Java implementiert. Der Java-Code wird mit dem Java-Compiler in Programmcode übersetzt, welcher mit dem Java-Interpreter interpretiert werden kann.

P6 : Der hier verwendete Java-Code $\underline{UC_{NS} < Java >}$ (universeller Code in Java für die Nutzerschnittstellenkomponente), der die Komponente der Nutzerschnittstelle des $\underline{SBE(L)}$ erzeugen soll, ist völlig unabhängig von der Sprache $\underline{L}$. Folgende Java-Klassen werden hauptsächlich verwendet:
Zum einen die Klasse **Editor**, welche die Darstellung der fensterorientierten Oberfläche und deren ereignisbedingte Steuerung übernimmt. Zum zweiten die **Exposition**, welche die Kommunikation mit der Auswerterkomponente des zu erzeugenden $\underline{SBE(L)}$ realisiert.

Der hier besprochene Java-Code, der als Quelle der Übersetzungsphase 6 dient, wurde ebenfalls im Rahmen dieser Diplomarbeit implementiert. Er wird kurz in Abschnitt 6.3 beschrieben.

Man erkennt in Abbildung 4.2, daß die Übersetzungsphasen 1 bis 6 sowie die von der Sprache $\underline{L}$ unabhängigen und in Phase 5 bzw. 6 als Quellcode dienenden Programmcodes $\underline{UC_{AW}}$ und $\underline{UC_{NS}}$ vom $\underline{Entwickler_{Gen_{SBE}}}$ (abgekürzt mit **I**) implementiert werden. Diese Implementierungen realisieren so den Generator $\underline{Gen_{SBE}}$, der sich in der Lage befindet, aus einer $\underline{Spez_{SBE(L)}}$ den $\underline{SBE(L)}$ zu erzeugen. Die Spezifizierung $\underline{Spez_{SBE(L)}}$ wird dabei vom $\underline{Entwickler_{SBE(L)}}$ (abgekürzt mit **II**) erstellt. Nach der Erzeugung des $\underline{SBE(L)}$ durch den $\underline{Gen_{SBE}}$ kann der $\underline{SBE(L)}$ vom $\underline{Nutzer_{SBE(L)}}$ (abgekürzt mit **III**) verwendet werden, um komfortabel einen Satz bzw. ein Programm $\underline{EK_L}$ der Sprache $\underline{L}$ zu editieren. Die Bedeutung der graphischen Elemente sowie die in dieser Abbildung verwendeten Abkürzungen sind (wie schon bei Abbildung 4.1) in der Legende erklärt, welche sich im Anhang B der Arbeit befindet.

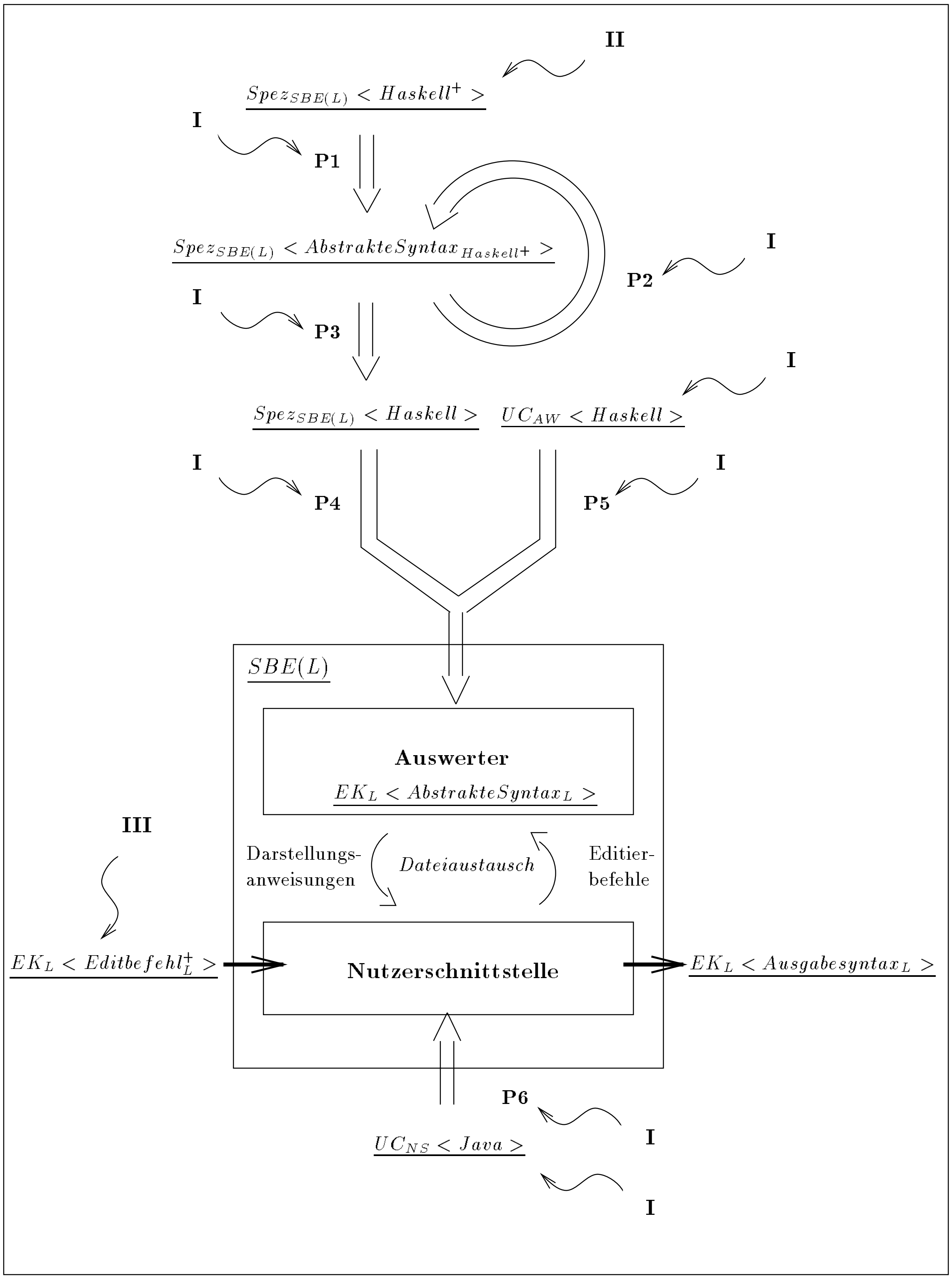

Abbildung 4.2: Schema der Generierung und Nutzung von $\underline{SBE(L)}$

4.7 Vergleich mit dem System SGen

In diesem Abschnitt vergleichen wir das von uns entworfene System $\underline{Gen_{SBE}}$ mit dem System Synthesizer Generator [RT89] (kurz SGen), das ebenfalls zur Erzeugung von sprachbasierten Editoren dient. Das System SGen ist seit 1981 unter der Leitung von T. W. Reps und T. Teitelbaum in der Entwicklung und wurde von der amerikanischen Firma GrammaTech, Inc. schon mehrere hundertmal vertrieben. Das System wurde in C implementiert und läuft unter dem Betriebssystem UNIX. Der Synthesizer Generator diente als Vorlage für das in dieser Diplomarbeit vorgestellte System $\underline{Gen_{SBE}}$.

Die Spezifizierung eines sprachbasierten Editors, die als Eingabe für das System SGen dient, ist wie bei unserem System $\underline{Gen_{SBE}}$ in die Komponenten **Abstrakte Syntax**, **Ausgabesyntax**, **Attributierung**, **Strukturelle Transformationen** und **Eingabesyntax** modularisiert. Als Spezifizierungssprache wird die Synthesizer Specification Language (kurz SSL) verwendet, die speziell für diesen Zweck entwickelt worden ist. Ihre Syntax ist daher stark an die Möglichkeit zur Beschreibung der einzelnen Spezifikationskomponenten angepaßt worden. Die Syntax unserer Spezifizierungssprache Haskell$^+$ ist nicht in diesem Maße spezialisiert und unterscheidet sich daher nicht so sehr von der Syntax einer funktionalen Sprache. Dies macht Haskell$^+$ für Programmierer mit Erfahrungen in der funktionalen Programmierung leichter zugänglich. Der Programmierer kann sogar rein funktional, also ohne Verwendung von inheriten Attributen bzw. Funktionen programmieren. Es sei in diesem Zusammenhang aber nochmal der Nachteil von Haskell$^+$ genannt, welcher in der derzeit sehr eingeschränkten Möglichkeit zur Beschreibung der Spezifikationskomponente **Eingabesyntax** liegt. Die Beseitigung dieses Nachteiles ist aber schon geplant.
Ein Vorteil von Haskell$^+$ gegenüber SSL ist es, Makroattributgrammatiken beschreiben zu können. Bei der Verwendung von SSL befindet man sich nur in der Lage, Attributgrammatiken zu spezifizieren, die ein Spezialfall von Makroattributgrammatiken sind.
Zur Auswertung der Attributinstanzen aus den Attributgrammatiken, die in SSL spezifiziert werden, wird im System SGen ein inkrementeller Attributauswerter für geordnete Attributgrammatiken [Kas80] verwendet. Das System $\underline{Gen_{SBE}}$ verwendet keinen inkrementellen Attributauswerter. Es ist aber möglich und auch angedacht, den $\underline{Gen_{SBE}}$ durch zukünftige Erweiterungen mit einem inkrementellen Attributauswerter auszustatten.
Beide Systeme zur Erzeugung von sprachbasierten Editoren generieren Editoren, die fensterorientierte, ereignisgesteuerte Oberflächen besitzen. Die mit dem $\underline{Gen_{SBE}}$ erzeugten Oberflächen sind in ihrer Funktionalität an die Oberflächen angelehnt, die mit dem System SGen erzeugt werden. Dabei ist der Umfang der bereitgestellten Funktionalität bei den erstgenannten Oberflächen noch deutlich geringer als bei den letztgenannten. Späteren Erweiterungen des $\underline{Gen_{SBE}}$ in Hinsicht auf die Menge der Oberflächenfunktionen bei den erzeugten Editoren steht aber nichts im Wege.

Kapitel 5

Spezifikation

Die Sprache Haskell$^+$ wurde in [Les99] durch eine EBNF-Definition und durch die Angabe deren kontextsensitiver Nebenbedingungen eingeführt, der uninformierte Leser sei daher auf diese Literatur verwiesen. Für die Verwendung von Haskell$^+$ als Eingabesprache des Systems $\underline{Gen_{SBE}}$ mußten allerdings bestimmte kontextsensitive Nebenbedingungen der Sprache abgeändert bzw. neue eingeführt werden.
Im folgenden beschreiben wir informell, wie eine Haskell$^+$-Spezifikation für eine formale Sprache $\underline{L}$ auszusehen hat (unter besonderer Beachtung der neuen kontextsensitiven Nebenbedingungen), um als Eingabe für den Generator zur Erzeugung eines sprachbasierten Editors für die Sprache $\underline{L}$ fungieren zu können. Dabei gehen wir konkret auf die einzelnen Spezifikationskomponenten ein und geben als Beispiel die Konkretisierungen für die $\underline{Spez_{SBE(P)}}$ des in Kapitel 3 vorgestellten sprachbasierten Editors $\underline{SBE(P)}$ unserer Beispielsprache $\underline{P}$ an.
Am Ende dieses Abschnitts finden sich dann die geänderten und die neuen kontextsensitiven Nebenbedingungen der Sprache Haskell$^+$, zur Aktualisierung und Vervollständigung der in [Les99] beschriebenen.

5.1 Abstrakte Syntax

Mit Hilfe der Syntax einer Sprache beschreiben wir deren unterliegende Struktur. Dies geschieht durch eine kontextfreie Grammatik. Ein editiertes Konstrukt $\underline{EK_L}$ wird intern immer als Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ repräsentiert. Jede Änderung dieses Konstruktes, hervorgerufen durch eine strukturelle Transformation oder durch eine textuelle Eingabe, verändert den unterliegenden Syntaxbaum.

Statt der kontextfreien Syntax verwenden wir hier die abstrakte Syntax. Sei z.B. die folgende Produktion einer kontextfreien Grammatik entnommen:

Program → “program” Identifier “;var” DeclList “begin” StmtList “end.”

In einer abstrakten Syntaxbeschreibung könnte die Produktion dann folgendermaßen aussehen:

Program = PROG Identifier DeclList StmtList

Die Terminalsymbole wurden entfernt und das Symbol PROG ist hinzugekommen. Dieses Symbol ist der Name der Produktion und damit zur Unterscheidung zu anderen Produktionen verwendbar. In Haskell^{+} können wir eine abstrakte Syntax mit Hilfe des Sprachkonstruktes *Data-Block* spezifizieren. Ein *Data-Block* ist eine Menge von *Data-Definitionen*, gekapselt durch die Schlüsselwörter *begindata* und *enddata*, wobei nach dem Schlüsselwort *begindata* der Name des *Data-Blockes* folgt.

In einer *Data-Definition* wird ein konstruktorbasierter Datentyp definiert. Wir bezeichnen einen Datentyp als konstruktorbasiert, wenn er durch (beliebigstellige) Konstruktoren definiert wird.

Die Zeile `Program = PROG Identifier DeclList StmtList`

wäre z.B. so eine *Data-Definition*. In dieser Definition wird der konstruktorbasierte Datentyp `Program` definiert. `PROG` ist hier der einzig zugehörige dreistellige Konstruktor, wobei `Identifier`, `DeclList` und `StmtList` die Typen seiner Nachfolger darstellen.

Wir beschreiben nun, wie ein *Data-Block* gestaltet sein darf, wenn wir mit ihm innerhalb der $\underline{Spez_{SBE(L)}}$ die abstrakte Syntax einer Sprache $\underline{L}$ beschreiben wollen.

Als Beispiel ist in Abbildung 5.1 der *Data-Block* abgebildet, welcher die abstrakte Syntax unserer Beispielsprache $\underline{P}$ festlegt.

Eine *Data-Definition* besteht im allgemeinem aus dem Namen des zu definierenden Typs und mindestens einem Konstruktor für diesen Typ. Dabei können die Konstruktoren eine Stelligkeit größer oder gleich Null aufweisen. Diese Konstruktoren finden sich dann in dem bearbeiteten Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ als Knotenbezeichner wieder.

Zusätzlich zu den selbstdefinierten konstruktorbasierten Typen ist die Verwendung der für Haskell^{+} vordefinierten nichtkonstruktorbasierten Typen *IntInput*, *BoolInput*, *CharInput*, *StringInput*, *IdentInput* und *QuotedStringInput* in den Argumentpositionen der Konstruktoren möglich (wir nennen eine Datentyp nichtkonstruktorbasiert, wenn er nicht unter Verwendung von Konstruktoren definiert wird). Die regulären Ausdrücke, auf denen die genannten Typen basieren, sind vordefiniert. Es ist nicht möglich, selbst Typen durch reguläre Ausdrücke zu spezifizieren. Die genannten vordefinierten Typen sind für das textuelle Editieren von $\underline{EK_L}$ gedacht und daher ausschließlich als Nachfolger eines einstelligen Konstruktors in einer Typdefinition eines konstruktorbasierten Datentyps zu verwenden. Die Notwendigkeit dieser Einschränkung hängt mit der Methodik der Umwandlung einer textuellen Eingabe in die abstrakte Syntax der Sprache $\underline{L}$ zusammen und wird in Abschnitt 5.5 genauer beschrieben.

Die Verwendung von strukturierten Typen wie Listen und Tupel sind in *Data-Definitionen* nicht zulässig, da sonst Datenstrukturen gebildet werden könnten, welche in dieser Form keine Bäume sind. Außerdem werden keine polymorphen *Data-Definitionen* unterstützt.

```
begindata P_Data

  data Program
    = PROG Identifier DeclList StmtList

  data DeclList
    = DECLLISTNIL
    | DECLLISTPAIR Decl DeclList

  data Decl
    = DECLARATION Identifier TypExp

  data TypExp
    = EMPTYTYPEXP
    | INTTYPEXP
    | BOOLTYPEXP
    | CHARTYPEXP
    | STRINGTYPEXP

  data Identifier
    = IDENTIFIERNULL
    | IDENTIFIER IdentInput

  data StmtList
    = STMTLISTNIL
    | STMTLISTPAIR Stmt StmtList

  data Stmt
    = EMPTYSTAT
    | ASSIGN Identifier Exp
    | IFTHENELSE Exp Stmt Stmt
    | WHILE Exp Stmt
    | COMPOUND StmtList

  data Exp
    = EMPTYEXP
    | INTCONST IntInput
    | BOOLCONST BoolInput
    | CHARCONST CharInput
    | STRINGCONST QuotedStringInput
    | ID Identifier
    | EQUAL Exp Exp
    | NOTEQUAL Exp Exp
    | ADD Exp Exp

enddata
```

Abbildung 5.1: *Data-Block* zur Beschreibung der abstrakten Syntax von $\underline{P}$

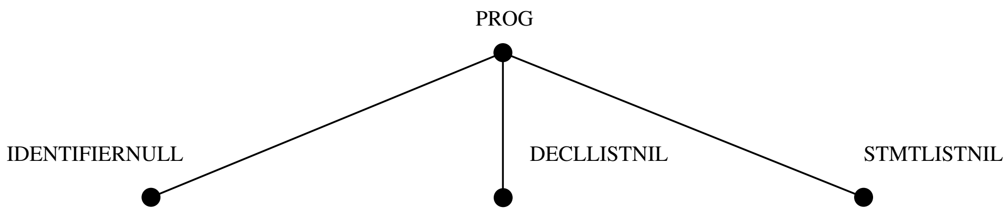

Abbildung 5.2: $\underline{EK[Init]_P < AbstrakteSyntax_P >}$ unserer Beispielsprache $\underline{P}$

In einem *Data-Block* ist es nicht erlaubt, in einer rechten Seite einer Typdefinition auf Typen zurückzugreifen, welche in anderen *Data-Blöcken* definiert sind. Es können natürlich mehrere *Data-Blöcke* vom $\underline{Entwickler_{SBE(L)}}$ innnerhalb einer $\underline{Spez_{SBE(L)}}$ definiert werden, aber nur einer beschreibt dann die abstrakte Syntax der Sprache $\underline{L}$.

In einem *Data-Block*, der uns die abstrakte Syntax der Sprache $\underline{L}$ beschreibt, muß stets ein spezieller konstruktorbasierter Datentyp vorhanden sein. Dieser Typ besitzt nur einen Konstruktor, welcher später als *Root-Symbol* verwendet wird. Dieser Konstruktor wird stets als Wurzelknoten des Syntaxbaums $\underline{EK_L < AbstrakteSyntax_L >}$ verwendet. Der Typ des Konstruktors darf nicht als Nachfolgertyp eines anderen Konstruktors aus dem *Data-Block* erscheinen.

Bei jedem konstruktorbasierten Datentyp, der in einem *Data-Block* definiert ist, welcher die abstrakte Syntax der Sprache $\underline{L}$ beschreibt, wird der in der Typdefinition erstangegebene Konstruktor zum sogenannten Defaultkonstruktor. Durch Verwendung dieser Defaultkonstruktoren und dem als *Root-Symbol* verwendeten Konstruktor ist es möglich, einen initialen Syntaxbaum, den wir mit $\underline{EK[Init]_L < AbstrakteSyntax_L >}$ bezeichnen, zu generieren, wenn der $\underline{Nutzer_{SBE(L)}}$ mit dem Editieren eines neuen $\underline{EK_L}$ beginnen möchte. Damit ein endlich großer Syntaxbaum $\underline{EK[Init]_L < AbstrakteSyntax_L >}$ aus den Defaultkonstruktoren erzeugt werden kann, ist es für den $\underline{Entwickler_{SBE(L)}}$ notwendig, bei der Wahl der Defaultkonstruktoren darauf zu achten, daß es einerseits beim automatischen Aufbau von $\underline{EK[Init]_L < AbstrakteSyntax_L >}$ nicht zu Zyklen kommt und das andererseits nur Konstruktoren als Defaultkonstruktor in Frage kommen, deren Nachfolger einen konstruktorbasierte Typen besitzen.
In Abbildung 5.2 ist z.B. der initiale Syntaxbaum $\underline{EK[Init]_P < AbstrakteSyntax_P >}$ unserer Beispielsprache $\underline{P}$ dargestellt.

5.2 Attributierung

In dieser Spezifikationskomponente wird der kontextsensitive Teil der Sprache $\underline{L}$ beschrieben, für die der sprachbasierte Editor $\underline{SBE(L)}$ erzeugt werden soll. Dies geschieht mit Hilfe einer bzw. mehrerer Makroattributgrammatiken.
Mit dem Haskell$^+$-Sprachkonstrukt *Mag-Block* ist es möglich, eine solche Grammatik zu spezifizieren. Dazu kann man innerhalb eines *Mag-Blocks* synthetische und inherite Funktionen deklarieren und definieren. Diese syntaxbasierten Funktionen arbeiten dann auf den konstruktorbasierten Datentypen, welche in einem bestimmten *Data-Block* definiert worden sind, dem *Input-Data-Block* für diesen *Mag-Block*. Dieser wird durch die *Input-Definition* des *Mag-Blocks* bestimmt. Wurde in diesem *Input-Data-Block* die abstrakte Syntax für die Sprache $\underline{L}$ festgelegt, so kann mit Hilfe der synthetischen und inheriten Funktionen ein Informationstransfer auf dem Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ realisiert werden.
Durch die Spezifizierung dieser Funktionen ist man in der Lage, die kontextsensitiven Nebenbedingungen der Sprache $\underline{L}$ zu beschreiben. Mit deren Hilfe kann der $\underline{SBE(L)}$ die kontextsensitiven Nebenbedingungen überprüfen. Dabei erzeugt er aus den Ergebniswerten der Funktionen Fehlermeldungen, welche dann dem $\underline{Nutzer_{SBE(L)}}$ dargeboten werden.

Wir geben hier noch einige allgemeine Informationen zu dem *Mag-Block*-Konstrukt an. Als erstes muß in einem *Mag-Block* durch die *Input-Definition* der zugehörige *Input-Data-Block* angegeben werden, welcher die Typen für die Rekursionsargumente der Funktionen festlegt, die in dem *Mag-Block* definiert werden. Danach ist immer das *Root-Symbol* durch einen Konstruktor aus dem *Input-Data-Block* zu spezifizieren. Nun folgen die konkreten Funktionsdeklarationen und -definitionen. Es ist stets erforderlich, den Typ einer Funktion (synthetisch, inherit etc.) anzugeben, die Typsignatur hingegen ist optional. Da es sich meistens anbietet, dieselbe Funktion für mehrere Typen aus dem *Input-Data-Block* zu definieren, kann hier das Haskell$^+$-Konstrukt der *Where-From-Typrestriktion* zur Realisierung eines eingeschränkten polymorphen Rekursionsargumentes verwendet werden.

Es sei darauf hingewiesen, daß es möglich ist, die Funktionsdefinitionen zur Spezifizierung von $\underline{SBE(L)}$ auf mehrere *Mag-Blöcke* zu verteilen. Dadurch ist man in der Lage, modular zu arbeiten, wodurch man eine übersichtliche Programmierung erreicht. Diese *Mag-Blöcke* müssen dabei auf den gleichen *(Input-)Data-Block* verweisen und dasselbe *Root-Symbol* spezifizieren. Es ist z.B. der Übersichtlichkeit halber sinnvoll, je einen eigenen *Mag-Block* für die in diesem Abschnitt besprochenen Funktionen zur Attributierung, für die Unparse-Funktionen und auch für die Transform-Funktion anzulegen. Die zwei letztgenannten Funktionstypen werden ebenfalls mit Hilfe des Sprachkonstruktes *Mag-Block* spezifiziert. Auf sie wird noch in den folgenden Abschnitten konkret eingegangen.

In Abbildung 5.3 ist der *Mag-Block* zu erkennen, welcher die kontextsensitiven Nebenbedingungen unserer Beispielsprache $\underline{P}$ beschreibt. Es wird dort eine inherite Funktion *env* definiert, welche die Information über die Typdeklarationen der `Identifier` in einem $\underline{P}$-Programm $\underline{EK_P}$ in Teile des Syntaxbaumes $\underline{EK_P < AbstrakteSyntax_P >}$ transportiert, in denen sie benötigt werden. Der Wert dieser Funktion ist konkret der Teilbaum von $\underline{EK_P < AbstrakteSyntax_P >}$, welcher als Wurzelknoten den zweiten Nachfolger des `PROG`-Knotens (*Root-Symbol*) hat, da er die Deklarationen des $\underline{P}$-Programms $\underline{EK_P}$ re-

```
beginmag P_SynInh_Mag

  input P_Data
  root PROG

  inh env :: a -> DeclList where a from
{StmtList,Stmt,Exp,DeclList,Decl,Identifier}

  t@(PROG t1 t2 t3) in
    env t2 = t2
    env t3 = t2

  t@(DECLLISTPAIR t1 t2) in
    env t1 = env t
    env t2 = env t

  t@(DECLARATION t1 t2) in
    env t1 = env t

  t@(STMTLISTPAIR t1 t2) in
    env t1 = env t
    env t2 = env t

  t@(ASSIGN t1 t2) in
    env t1 = env t
    env t2 = env t
  t@(IFTHENELSE t1 t2 t3) in
    env t1 = env t
    env t2 = env t
    env t3 = env t
  t@(WHILE t1 t2) in
    env t1 = env t
    env t2 = env t
  t@(COMPOUND t1) in
    env t1 = env t

  t@(ID t1) in
    env t1 = env t
  t@(EQUAL t1 t2) in
    env t1 = env t
    env t2 = env t
  t@(NOTEQUAL t1 t2) in
    env t1 = env t
    env t2 = env t
  t@(ADD t1 t2) in
    env t1 = env t
    env t2 = env t


  syn typ :: a -> TypExp where a from {Identifier,Exp}
```

```
  typ IDENTIFIERNULL = EMPTYTYPEXP
  t@(IDENTIFIER t1) in
    typ t = lookuptyp t (env t)

  typ EMPTYEXP = EMPTYTYPEXP
  typ (INTCONST t1) = INTTYPEXP
  typ (BOOLCONST t1) = BOOLTYPEXP
  typ (CHARCONST t1) = CHARTYPEXP
  typ (STRINGCONST t1) = STRINGTYPEXP
  typ (ID t1) = (typ t1)
  typ (EQUAL t1 t2) = BOOLTYPEXP
  typ (NOTEQUAL t1 t2) = BOOLTYPEXP
  typ (ADD t1 t2) = INTTYPEXP

endmag
```

Abbildung 5.3: *Mag-Block* zur Beschreibung der Attributierung von $\underline{P}$

präsentiert.
Außerdem wird eine synthetische Funktion *typ* definiert, welche die Information darüber bereitstellt, welchen Typ aus $\underline{P}$ ausgewählte Sprachkonstrukte in dem $\underline{P}$-Programm $\underline{EK_P}$ besitzen.
Die in Abbildung 5.3 verwendete Hilfsfunktion `lookuptyp` wird in Abschnitt 5.6 beschrieben.

In Abbildung 5.4 ist der attributierte Syntaxbaum für folgendes $\underline{P}$-Programm skizziert. Dabei ist die synthetische Funktion *typ* rechts und die inherite Funktion *env* links neben den entsprechenden Knoten des Syntaxbaumes eingezeichnet. Die Abhängigkeiten zwischen den Attributen sind mit Hilfe der durchgehenden Pfeile eingezeichnet. Durchgehende Pfeile ohne Beschreibung des Startpunktes verdeutlichen die Übergabe eines konstanten Wertes (vgl. dazu die Funktionsdefinitionsgleichungen in Abbildung 5.3). Es sei in diesem Zusammenhang auch darauf hingewiesen, daß das Attribut *typ* am ersten Nachfolger des Wurzelkonstrukors nie ausgewertet wird. Die gestrichelten Pfeile beschreiben die Wertübernahme der Funktion *env*. Die Strings "Bsp-Programm" und "a" werden nicht als Knoten im Syntaxbaum gekennzeichnet, da sie nicht von einem konstruktorbasierten Datentyp sind und daher nicht als Argumente von synthetischen oder inheriten Funktionen dienen können.

```
(PROG
        (IDENTIFIER  "Bsp-Programm")
        (DECLLISTPAIR
                (DECLARATION  (IDENDIFIER  "a")  (INTTYPEXP))
                (DECLLISTPAIR  ..  ..))
        (STMTLISTPAIR
                (WHILE
                        (EQUAL  (ID (IDENTIFIER "a"))  (TRUE))
                        (EMPTYSTAT))
                (STMTLISTPAIR  ..  ..)))
```

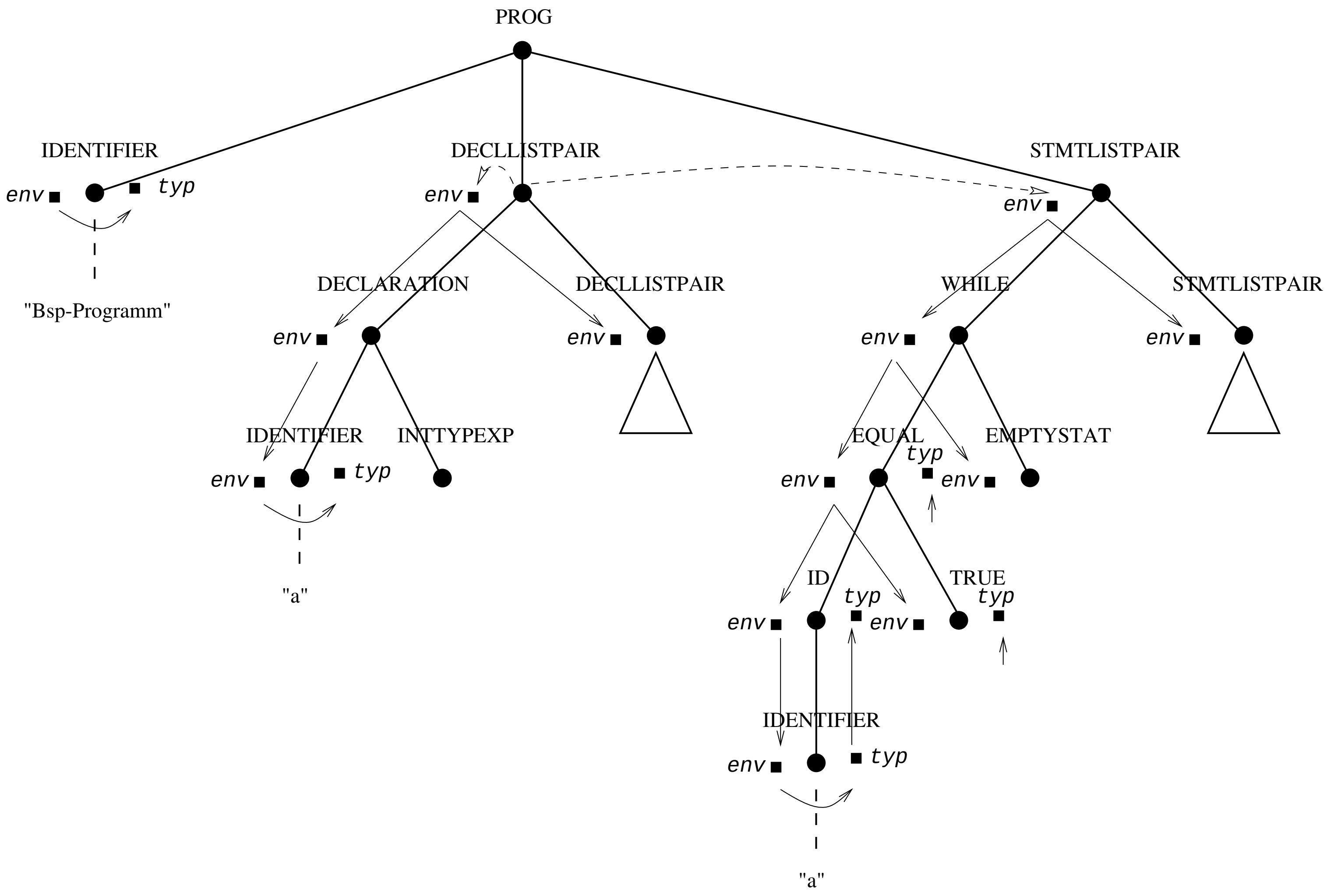

Abbildung 5.4: Attributierung

5.3 Ausgabesyntax

Durch die Ausgabesyntax wird festgelegt, wie dem $\underline{Nutzer_{SBE(L)}}$ das $\underline{EK_L}$ durch den $\underline{SBE(L)}$ textuell dargeboten werden soll. Da das $\underline{EK_L}$ während des Editierens intern als Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ vorliegt, muß es für die Anzeige geeignet in die textuelle Darstellungsform $\underline{EK_L < Ausgabesyntax_L >}$ umgewandelt werden. Diese Aufgabe übernimmt eine sogenannte Unparse-Funktion. Die Umwandlung der intern vorliegenden Repräsentation von $\underline{EK_L}$ in die externe Darstellung wird realisiert, indem die Unparse-Funktion auf den Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ angewendet wird und dabei als Ergebnis die Zeichenkette $\underline{EK_L < Ausgabesyntax_L >}$ berechnet. Das geschieht mit Hilfe eines Durchlaufes (top-down left-to-right) dieser rekursiven Funktion durch $\underline{EK_L < AbstrakteSyntax_L >}$.

Als Beispiel sei hier schon mal auf die Definition der Unparse-Funktion *up* für den Konstruktor `PROG` (*Root-Symbol*) bei unserer Beispielsprache $\underline{P}$ vorgegriffen:

```
up (PROG t1 t2 t3) = "program " ++ (up t1) ++ ";\n" ++
                     "var" ++ "\t\n" ++
                         (up t2) ++ "\b\n" ++
                     "begin" ++ "\t\n" ++
                         (up t3) ++ "\b\n" ++
                     "end."
```

Man erkennt hierbei neben der Rekursion auch die Terminalsymbole wieder, welche bei der Umwandlung von der konkreten kontextfreien Syntax in die abstrakte Syntax verloren gegangen waren (siehe Abschnitt 3.1). Die Steuerzeichen "\t", "\b" und "\n" können für eine bessere Lesbarkeit von $\underline{EK_L < Ausgabesyntax_L >}$ sinnvoll verwendet werden. Ihre Bedeutung wird später erklärt.

Es ist erlaubt, mehrere voneinander unabhängige Unparse-Funktionen zu definieren, um dem $\underline{Nutzer_{SBE(L)}}$ mehrere Sichten auf sein editiertes Konstrukt $\underline{EK_L}$ zu erlauben. Dabei sollte wenigstens eine Unparse-Funktion spezifiziert werden, welche ein ganzheitliches Abbild $\underline{EK_L < Ausgabesyntax_L >}$ der internen Repräsentation $\underline{EK_L < AbstrakteSyntax_L >}$ erzeugt, damit der $\underline{Nutzer_{SBE(L)}}$ stets vollständig über $\underline{EK_L}$ informiert ist. Diese Unparse-Funktion nennen wir in der hier vorliegenden schriftlichen Arbeit auch die primäre Unparse-Funktion, wobei es dem $\underline{Entwickler_{SBE(L)}}$ nicht möglich ist (was auch nicht von Vorteil wäre), in $\underline{Spez_{SBE(L)}}$ eine Unparse-Funktion als primär zu kennzeichnen.
In der ersten Implementierungsversion des Systems $\underline{Gen_{SBE}}$ hat der $\underline{Nutzer_{SBE(L)}}$ eines erzeugten $\underline{SBE(L)}$ auch noch nicht die Möglichkeit eine bestimmte Ausgabesyntax von $\underline{L}$ auszuwählen, es wird immer die vom $\underline{Entwickler_{SBE(L)}}$ in der $\underline{Spez_{SBE(L)}}$ zuerst definierte verwendet.

Die Unparse-Funktionen kann man auch als spezielle synthetische Funktionen auffassen. Daher sind sie auch wie eine synthetische Funktion innerhalb eines Mag-Blocks zu definieren. Dabei gelten aber folgende besondere Bedingungen. Eine Unparse-Funktion hat wie eine synthetische Funktion stets ein Rekursionsargument, welches von einem konstruktorbasierten Typ sein muß, der in dem *Input-Data-Block* für den Mag-Block definiert worden

ist, in dem diese Unparse-Funktion definiert wird. Im Unterschied zu einer synthetischen (wie einer inheriten) Funktion muß bei einer Unparse-Funktion dieser *Input-Data-Block* genau jener sein, in welchem die abstrakte Syntax der Sprache $\underline{L}$ definiert worden ist. Da der $\underline{Entwickler_{SBE(L)}}$ in der $\underline{Spez_{SBE(L)}}$ nicht anhand eines Schlüsselwortes festlegen kann, welcher *Data-Block* die abstrakte Syntax der Sprache $\underline{L}$ definiert, muß er es indirekt festlegen. Dies geschieht durch die Verwendung des für den genannten Zweck gewählten *Data-Block* in der *Input-Definition* eines *Mag-Blocks*, innerhalb dessen eine Unparse-Funktion definiert wird. In diesem Zusammenhang sei auch darauf hingewiesen, daß alle *Mag-Blocks*, in denen eine Unparse-Funktion definiert wird, dieselbe *Input-Definition* aufweisen und dasselbe Root-Symbol definieren müssen. Weiterhin muß mindestens eine Unparse-Funktion definiert werden.
Als Ergebnistyp besitzt eine Unparse-Funktion immer den Typ *String*. Kontextparameter sind nicht zulässig.
Da eine Unparse-Funktion in fast allen Fällen für mehrere Typen definiert werden muß, kann hier wie bei den synthetischen und inheriten Funktionen das Haskell$^+$-Konstrukt der *Where-From-Typrestriktion* in der Deklaration der Funktion zur Anwendung kommen.

Besondere Einschränkungen gibt es bei der Unparse-Funktion auch bei der Bildung der rechten Seite einer Definition:

Sei upfun der Name einer einstelligen Unparse-Funktion, angewendet auf den n-stelligen Konstruktor Kon, mit $0 \leq n$ und

$$
\begin{aligned}
&\mathrm{t@(Kon\ t_1\ ..\ t_n)\ in}\\
&\qquad \mathrm{upfun\ t = RS}
\end{aligned}
$$

eine Definitionsgleichung dieser Unparse-Funktion. Die rechte Seite RS dieser Funktionsdefinition darf nur aus folgenden Konstrukten aufgebaut werden:

- rekursiven Aufrufen (upfun t_j) der Unparse-Funktion upfun mit $1 \leq j \leq n$

  Falls es sich um die primäre Unparse-Funktion handelt, sollte folgendes beachtet werden. Für jeden Nachfolger tj des Konstruktors Kon soll genau ein rekursiver Aufruf erfolgen, damit ein rekursiver Abstieg durch den gesamten Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ möglich wird. Ausnahmen bilden dabei die Nachfolger, welche einen nichtkonstruktorbasierten Datentyp (*IntInput*, *BoolInput* etc.) besitzen. Diese Nachfolger sind ohne weitere Anwendung der Unparse-Funktion in die rechte Seite der Funktionsdefinition zu übernehmen, wobei aber vorher auf diese Nachfolger immer eine Anwendung der 'show'-Funktion stattfinden muß um Typkompatibilität zu erreichen.

- Stringkonstanten

  In den Stringkonstanten können folgende Kontrollzeichen (und nur diese) verwendet werden, um Texteinrückungen und Zeilenumbrüche zu erzielen.

  \t rückt den linken Zeilenrand um eine Tabulatoreinheit nach rechts

  \b rückt den linken Zeilenrand um eine Tabulatureinheit nach links

  \n erzeugt einen Zeilenumbruch

- Funktionen mit dem Ergebnistyp *String*

  In den Argumentpositionen dieser Funktionen sind keine rekursiven Aufrufe der Unparse-Funktion upfun erlaubt.
  Anwendung finden hier vor allem die schon besprochenen synthetischen und inheriten Funktionen, welche als Ergebniswert Meldungen über kontextsensitive Fehler in $\underline{EK_L}$ o.ä. für den $\underline{Nutzer_{SBE(L)}}$ erzeugen.

Die genannten Konstrukte können ausschließlich mit dem Infix-Operator ++ verknüpft werden.

Eine Unparse-Funktion darf nur in den rechten Seiten ihrer eigenen Definitionen aufgerufen werden. Das liegt an den speziellen Seiteneffekten dieser Funktionsart, worauf wir in Abschnitt 6.1.3 noch genauer eingehen werden.

In der Abbildung 5.5 ist der *Mag-Block* zu betrachten, welcher die primäre Unparse-Funktion zur Festlegung der Ausgabesyntax unserer Beispielsprache $\underline{P}$ enthält. Die dabei verwendeten Hilfsfunktionen (wie z.B. `numberofdecls`) werden in Abschnitt 5.6 erläutert.

```
beginmag P_Unparse_Mag

  input P_Data
  root PROG

  unparse up :: a -> String where a from {Program,DeclList,Decl,
                                          TypExp,Identifier,StmtList,Stmt,Exp}

  up (PROG t1 t2 t3) = "program " ++ (up t1) ++ ";\n" ++
                       "var" ++ "\t\n" ++
                           (up t2) ++ "\b\n" ++
                       "begin" ++ "\t\n" ++
                           (up t3) ++ "\b\n" ++
                       "end."

  up DECLLISTNIL = ""
  up (DECLLISTPAIR t1 t2) = (up t1) ++ ";\n" ++ (up t2)
```

```
t@(DECLARATION t1 t2) in
  up t = (up t1) ++
         (ifthenelse
              ((not (t1 == IDENTIFIERNULL)) && ((numberofdecls t1 (env t)) > 1))
           " { Multiply Declared } "
           "")
          ++ " : " ++ (up t2)

up EMPTYTYPEXP = "<type>"
up INTTYPEXP = "integer"
up BOOLTYPEXP = "boolean"
up CHARTYPEXP = "char"
up STRINGTYPEXP = "string"

up IDENTIFIERNULL = "<identifier>"
up (IDENTIFIER t1) = (show t1)

up STMTLISTNIL = ""
up (STMTLISTPAIR t1 t2) = (up t1) ++ ";\n" ++ (up t2)

up EMPTYSTAT = "<statement>"

t@(ASSIGN t1 t2) in
  up t = (up t1) ++
         (ifthenelse
              ((t1 == IDENTIFIERNULL) || (isdeclared t1 (env t)))
           ""
           " { Not Declared } ")
         ++ " := " ++ (up t2) ++
         (ifthenelse
              (incomptypes (typ t1) (typ t2))
           " { Incompatibles Types In := } "
           "")
up (IFTHENELSE t1 t2 t3) = "if " ++ (up t1) ++
                           (ifthenelse
                              (incomptypes (typ t1) BOOLTYPEXP)
                             " { Boolean Expression Needed } "
                             "")
                           ++ " then" ++ "\t\n" ++
                               (up t2) ++ "\b\n" ++
                           "else" ++ "\t\n" ++
                               (up t3) ++ "\b"

up (WHILE t1 t2) = "while " ++ (up t1) ++
                   (ifthenelse
                      (incomptypes (typ t1) BOOLTYPEXP)
                     " { Boolean Expression Needed } "
                     "")
                   ++ " do" ++ "\t\n" ++
                       (up t2) ++ "\b"
```

```
  up (COMPOUND t1) = "begin" ++ "\t\n" ++
                         (up t1) ++ "\b\n" ++
                     "end"

  up EMPTYEXP = "<exp>"
  up (INTCONST t1) = (show t1)
  up (BOOLCONST t1) = (show t1)
  up (CHARCONST t1) = (show t1)
  up (STRINGCONST t1) = (show t1)

  t@(ID t1) in
    up t = (up t1) ++ (ifthenelse
                        ((t1 == IDENTIFIERNULL) || (isdeclared t1 (env t)))
                        ""
                        " { Not Declared } ")

  up (EQUAL t1 t2) = "(" ++ (up t1) ++ " = " ++
                     (ifthenelse
                        (incomptypes (typ t1) (typ t2))
                       " { Incompatibles Types In = } "
                       "")
                     ++ (up t2) ++ ")"

  up (NOTEQUAL t1 t2) = "(" ++ (up t1) ++ " <> " ++
                        (ifthenelse
                                (incomptypes (typ t1) (typ t2))
                          " { Incompatibles Types In <> } "
                          "")
                        ++ (up t2) ++ ")"
  up (ADD t1 t2) = "(" ++ (up t1) ++
                   (ifthenelse
                        (incomptypes (typ t1) INTTYPEXP)
                     " { Int Expression Needed } "
                     "")
                   ++ " + " ++
                   (ifthenelse
                        (incomptypes (typ t2) INTTYPEXP)
                     " { Int Expression Needed } "
                     "")
                   ++ (up t2) ++ ")"

endmag
```

Abbildung 5.5: *Mag-Block* zur Beschreibung der Ausgabesyntax von <u>P</u>

5.4 Strukturelle Transformationen

Die Transform-Funktion ermöglicht dem $\underline{Nutzer_{SBE(L)}}$ strukturelles Editieren. Beim Vorgang des strukturellen Editierens von $\underline{EK_L}$ wird die abstrakte Syntax der Sprache $\underline{L}$ verwendet. Bei dieser Art des Editierens wird der Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ direkt verändert. Dies geschieht, indem ein Teilbaum dieses Syntaxbaumes durch einen anderen Teilbaum ersetzt wird. Um der abstrakten Syntax gerecht zu werden, müssen dabei beide Teilbäume, bzw. die Konstruktoren an deren Wurzel, vom gleichen Typ sein.

Welche Teilbäume durch welche Teilbäume ersetzt werden können und auf welche Weise dies geschehen soll, wird mit Hilfe der sogenannten Transform-Funktion spezifiziert.

Die Transform-Funktion kann man (wie schon die Unparse-Funktionen) auch als spezielle synthetische Funktion betrachten. Sie wird demnach ebenfalls innerhalb eines *Mag-Blockes* definiert, wobei wiederum besondere Bedingungen gelten. Die Transform-Funktion besitzt stets ein Rekursionsargument und einen Kontextparameter. Das Rekursionsargument ist von einem konstruktorbasierten Typ, der im *Input-Data-Block* des *Mag-Blockes* definiert worden ist. Der *Input-Data-Block* muß natürlich derjenige Data-Block sein, in dem die abstrakte Syntax der Sprache $\underline{L}$ festgelegt worden ist. Daher muß die *Input-Definition* und die Definition des *Root-Symbols* des *Mag-Blockes*, in welchem die Transform-Funktion definiert wird, mit der der *Mag-Blöcke* übereinstimmen, in denen eine Unparse-Funktion definiert wird.
Das Rekursionsargument einer Transform-Funktionsdefinition legt den Wurzel-Konstruktor des zu ersetzenden Teilbaumes fest. Das allein reicht jedoch nicht für die eindeutige Auswahl eines konkret zu ersetzenden Teilbaumes aus, da in dem bearbeiteten Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ mehrere Teilbäume mit dem gleichen Wurzel-Konstruktor existieren können. Die eindeutige Auswahl erfolgt dadurch, daß während des Editierens von $\underline{EK_L}$ durch den $\underline{Nutzer_{SBE(L)}}$ stets ein bestimmtes Sprachkonstrukt aus $\underline{EK_L}$ und somit ein konkreter Teilbaum der internen Darstellung $\underline{EK_L < AbstrakteSyntax_L >}$ als ausgewählt gilt. Dieser Teilbaum wird dann auch transformiert.
Der Kontextparameter der Transform-Funktion ist zur Auswahl einer konkreten Transformation notwendig, damit man mehrere verschiedene Transformationen für Teilbäume mit dem gleichem Wurzelkonstruktor definieren kann. Dieses zweite Argument besitzt einen, speziell dafür definierten, konstruktorbasierten Datentyp *TransSelection*, welcher nur nullstellige Konstruktoren besitzt. Die Namen dieser Konstruktoren finden sich im $\underline{SBE(L)}$ als Kommandos für die entsprechenden strukturellen Transformationen wieder. Der $\underline{Entwickler_{SBE(L)}}$ muß den Datentyp *TransSelection* mit Hilfe einer *Data-Defintion* der Sprache Haskell definieren, welche mit der Klausel *deriving (Show,Eq,Read)* endet.
Der Ergebnistyp der Transform-Funktion ist immer vom gleichen Typ wie das erste Argument.

Damit nun ein Teilbaum mit dem n-stelligen Wurzel-Konstruktor Kon in den Teilbaum TransErg strukturell transformiert werden kann, muß die Transform-Funktion für den Konstruktor Kon definiert werden. Dies würde bei der Transform-Funktion transfun folgendermaßen aussehen, wobei K ein Konstruktor vom Typ *TransSelection* ist und die Transformation benennt:

t@(Kon t_1 .. t_n) in
    transfun t K = TransErg

Auf der rechten Seite einer Definition der Transform-Funktion muß nicht unbedingt ein konstanter Teilbaum des passenden Types stehen. Hier können auch Funktionen (insbesondere auch synthetische und inherite, dies bietet die Möglichkeit der Anwendung eines Makro Attributed Tree Transducers) aufgerufen werden, welche einen Teilbaum mit dem geforderten Typ als Ergebnis liefern.

Durch das strukturelle Editieren von $\underline{Nutzer_{SBE(L)}}$ wird gewährleistet, daß $\underline{EK_L}$ stets syntaktisch korrekt, wenn auch nicht unbedingt vollständig spezifiziert ist. Damit $\underline{EK_L}$ nicht zu jedem Zeitpunkt vollständig spezifiziert sein muß, können sogenannte Platzhalter in $\underline{EK_L < Ausgabesyntax_L >}$ verwendet werden. Diesen Platzhaltern liegen nullstellige Konstruktoren zugrunde, geeigneterweise sind dies gleichzeitig die Defaultkonstruktoren des entsprechenden Datentyps aus dem *Input-Data-Block*. Diese Platzhalter stehen dann für ein noch unspezifiziertes Sprachkonstrukt von $\underline{L}$ in $\underline{EK_L}$. Soll dieses Sprachkonstrukt konkretisiert werden, kann es durch eine passende Schablone ersetzt werden. Diese Schablone spezifiziert ein Sprachkonstrukt, wobei in der Schablone wiederum Platzhalter enthalten sein können. Auf diese Weise kann eine Top-Down-Entwicklung von $\underline{EK_L}$ realisiert werden.

In unserer Beispielsprache $\underline{P}$ existiert so z.B. für den Datentyp `Stmt` der nullstellige (Default-)Konstruktor `EMPTYSTAT` als Platzhalter für das Sprachkonstrukt einer Anweisung. Die folgende Funktionsdefinition der Transform-Funktion *trans* beschreibt die Transformation dieses Platzhalters in eine Schablone, welche die unspezifizierte Anweisung mit einer Zuweisung instanziiert:

```
trans EMPTYSTAT InsertAssign = (ASSIGN IDENTIFIERNULL EMPTYEXP)
```

Es sei allgemein darauf hingewiesen, daß nicht nur Platzhalter durch Schablonen ersetzt werden können, vielmehr können auch die Schablonen transformiert werden, einerseits in einen entsprechenden Platzhalter und andererseits auch in andere Schablonen. Weiterhin ist es nicht unbedingt erforderlich, daß jeder Datentyp aus dem *Input-Data-Block* einen Konstruktor besitzt, der die Funktion eines Platzhalters übernimmt.

An den Stellen von $\underline{EK_L}$, an denen dem $\underline{Nutzer_{SBE(L)}}$ keine textuelle Eingabe bzw. Änderung erlaubt ist, ist das strukturelle Editieren die einzige Möglichkeit, $\underline{EK_L}$ zu editieren. Daher ist es dem $\underline{Entwickler_{SBE(L)}}$ nahezulegen, für jeden konstruktorbasierten Datentyp aus dem *Input-Data-Block*, Transformationen seiner Konstruktoren in diejenigen Konstruktoren dieses Datentyps durch die Transform-Funktion zu spezifizieren, welche nicht durch textuelle Modifikation in $\underline{EK_L < AbstrakeSyntax_L >}$ eingefügt werden können. Dabei ist es oft sinnvoll, zwischen dem Defaultkonstruktor für den besprochenen Datentyp und den übrigen Konstruktoren zu unterscheiden. Dann kann man die Transformationsmöglichkeiten übersichtlich halten, indem man je eine Transformation vom Defaultkonstruktor in einen der anderen Konstruktoren bereitstellt und zusätzlich dazu je eine Transformation von einem der letztgenannten Konstruktoren in den Defaultkonstruktor zurück.

Die im *Input-Data-Block* vorkommenden nichtkonstruktorbasierten Datentypen (*IntInput,BoolInput etc.*) können nicht strukturell transformiert werden, daher darf die Transform-Funktion auch nicht für diese Typen definiert werden.

Der $\underline{Entwickler_{SBE(L)}}$ darf in der $\underline{Spez_{SBE(L)}}$ höchstens eine Transform-Funktion definieren, da er durch die Verwendung des Kontextparameter in den Definitionen einer einzigen Transform-Funktion ohnehin in der Lage ist, verschiedene Transformationen der Teilbäume $\underline{EK_L < AbstrakteSyntax_L >}$ mit dem gleichem Wurzel-Konstruktor zu spezifizieren.

Die Verwendung von Aufrufen der Transform-Funktion ist in der gesamten $\underline{Spez_{SBE(L)}}$ untersagt. Denn ein Aufruf einer Transform-Funktion soll ausschließlich dann geschehen, wenn der $\underline{Nutzer_{SBE(L)}}$ während des Editierens von $\underline{EK_L}$ den Befehl dazu gibt.

```
data TransSelection
  = InsertDecl | InsertStat
  | InsertInteger | InsertBoolean | InsertChar | InsertString
  | InsertTrue | InsertFalse | InsertId
  | InsertEqual | InsertNotequal | InsertAdd
  | InsertAssign | InsertIf
  | InsertWhile | InsertBegin
  | Block | DisBlock
  | Cut | CutFirst
  | Change deriving (Show,Eq,Read)
```

Abbildung 5.6: Datentyp TransSelection

In Abbildung 5.7 ist der *Mag-Block* abgebildet, in welchem die Transform-Funktion zur Spezifizierung der strukturellen Transformationen unserer Beispielsprache $\underline{P}$ definiert ist. Die Hilfsfunktion `disblock` wird in Abschnitt 5.6 erläutert. Auf die Typumwandlungsfunktion `boolToBoolInput` gehen wir in Abschnitt 5.5 ein.

Abbildung 5.6 zeigt uns vorher die Definition des Datentyps *TransSelection*, welcher die nullstelligen Konstruktoren enthält, die als Kontextparameter der Transform-Funktion aus Abbildung 5.7 dienen.

```
beginmag P_transform_Mag

  input P_Data
  root PROG


  transform trans :: a -> TransSelection -> a where a from
                        {DeclList,TypExp,Identifier,StmtList,Stmt,Exp}

  trans DECLLISTNIL InsertDecl = DECLLISTPAIR
                                     (DECLARATION IDENTIFIERNULL EMPTYTYPEXP)
                                     DECLLISTNIL
```

```
t@(DECLLISTPAIR t1 t2) in
  trans t CutFirst = t2
  trans t Cut = DECLLISTNIL

t@EMPTYTYPEXP in
  trans t InsertInteger = INTTYPEXP
  trans t InsertBoolean = BOOLTYPEXP
  trans t InsertChar = CHARTYPEXP
  trans t InsertString = STRINGTYPEXP
trans INTTYPEXP Cut = EMPTYTYPEXP
trans BOOLTYPEXP Cut = EMPTYTYPEXP
trans CHARTYPEXP Cut = EMPTYTYPEXP
trans STRINGTYPEXP Cut = EMPTYTYPEXP

trans (IDENTIFIER t1) Cut = IDENTIFIERNULL

trans STMTLISTNIL InsertStat = STMTLISTPAIR EMPTYSTAT STMTLISTNIL
t@(STMTLISTPAIR t1 t2) in
  trans t CutFirst = t2
  trans t Cut = STMTLISTNIL

t@EMPTYSTAT in
  trans t InsertAssign = (ASSIGN IDENTIFIERNULL EMPTYEXP)
  trans t InsertIf = (IFTHENELSE EMPTYEXP EMPTYSTAT EMPTYSTAT)
  trans t InsertWhile = (WHILE EMPTYEXP EMPTYSTAT)
  trans t InsertBegin = (COMPOUND (STMTLISTPAIR EMPTYSTAT STMTLISTNIL))
t@(ASSIGN t1 t2) in
  trans t Block = (COMPOUND (STMTLISTPAIR t STMTLISTNIL))
  trans t Cut = EMPTYSTAT
t@(IFTHENELSE t1 t2 t3) in
  trans t Block = (COMPOUND (STMTLISTPAIR t STMTLISTNIL))
  trans t Change = (IFTHENELSE t1 t3 t2)
  trans t Cut = EMPTYSTAT
t@(WHILE t1 t2) in
  trans t Block = (COMPOUND (STMTLISTPAIR t STMTLISTNIL))
  trans t Cut = EMPTYSTAT
t@(COMPOUND t1) in
  trans t Block = (COMPOUND (STMTLISTPAIR t STMTLISTNIL))
  trans t DisBlock = disBlock t
  trans t Cut = EMPTYSTAT

t@EMPTYEXP in
  trans t InsertTrue = (BOOLCONST (boolToBoolInput True))
  trans t InsertFalse = (BOOLCONST (boolToBoolInput False))
  trans t InsertId = ID IDENTIFIERNULL
  trans t InsertEqual = EQUAL EMPTYEXP EMPTYEXP
  trans t InsertNotequal = NOTEQUAL EMPTYEXP EMPTYEXP
  trans t InsertAdd = ADD EMPTYEXP EMPTYEXP
```

```
  trans (INTCONST t1) Cut = EMPTYEXP
  trans (BOOLCONST t1) Cut = EMPTYEXP
  trans (CHARCONST t1) Cut = EMPTYEXP
  trans (STRINGCONST t1) Cut = EMPTYEXP
  trans (ID t1) Cut = EMPTYEXP
  trans (EQUAL t1 t2) Cut = EMPTYEXP
  trans (NOTEQUAL t1 t2) Cut = EMPTYEXP
  trans (ADD t1 t2) Cut = EMPTYEXP

endmag
```

Abbildung 5.7: *Mag-Block* zur Beschreibung der strukturellen Transformationen von $\underline{P}$

5.5 Eingabesyntax

Wie schon erwähnt, fehlt in Haskell$^+$ noch die Möglichkeit, innerhalb der $\underline{Spez_{SBE(L)}}$ eine Eingabesyntax so zu spezifizieren, daß die textuelle Editierung aller in der Sprache $\underline{L}$ enthaltenen Sprachkonstrukte mit dem generierten $\underline{SBE(L)}$ möglich wird. Dazu müßte der $\underline{Entwickler_{SBE(L)}}$ reguläre Ausdrücke beschreiben können, damit ein Scannen und somit auch ein Parsen der vom $\underline{Nutzer_{SBE(L)}}$ eingegebenen Zeichenkette realisiert werden kann, um sie in den $\underline{EK_L < AbstrakteSyntax_L >}$ geeignet einzufügen.
Eine Spezifizierung von regulären Ausdrücken in Haskell$^+$ ist derzeitig aber nicht möglich. Weiterhin wäre es notwendig, in Haskell$^+$ eine Möglichkeit zu schaffen, mit deren Hilfe der $\underline{Entwickler_{SBE(L)}}$ beschreiben könnte, wie die regulären Ausdrücke geparst werden sollen. Außerdem wäre es von Vorteil, wenn der $\underline{Entwickler_{SBE(L)}}$ festlegen könnte, welche Sprachkonstrukte aus der Sprache $\underline{L}$, vom $\underline{Nutzer_{SBE(L)}}$ textuell editiert werden dürfen und welche nicht. Diese Möglichkeiten sind ebenfalls zur Zeit nicht realisiert.
Das Problem wird in dieser Arbeit gelöst, indem ausschließlich die für die Sprache Haskell$^+$ vordefinierten Datentypen *IntInput*, *BoolInput*, *CharInput*, *StringInput*, *IdentInput* und *QuotedStringInput* vom $\underline{Nutzer_{SBE(L)}}$ textuell editiert werden dürfen. Die regulären Ausdrücke, welche den genannten vordefinierten Typen zugrundeliegen, werden nicht vom $\underline{Entwickler_{SBE(L)}}$ festgelegt, sondern vom $\underline{Entwickler_{Gen_{SBE}}}$ vorgegeben und sind somit in jedem erzeugten $\underline{SBE(L)}$ gleich (unabhängig von der konkreten Sprache $\underline{L}$). Daher reduziert sich die Angabe der Spezifikationskomponente für die Beschreibung der Eingabesyntax der Sprache $\underline{L}$ bei der Festlegung der $\underline{Spez_{SBE(L)}}$ durch den $\underline{Entwickler_{SBE(L)}}$ auf die Verwendung der für Haskell$^+$ vordefinierten Datentypen in der Beschreibung der abstrakten Syntax von $\underline{L}$.

Die vordefinierten Typen können zur textuellen Manipulation der folgenden Sprachelemente verwendet werden:

IntInput	: ganze Zahlen	z.B. -99
BoolInput	: boolsche Werte	z.B. True
CharInput	: Buchstaben	z.B. 'a'

StringInput	: beliebige Zeichenkette, ohne Zeilenwechsel	z.B. `3s.&"\`
IdentInput	: Bezeichner	z.B. a45
QuotedStringInput	: gequotete Zeichenkette	z.B. `"hallo Du"`

Wie schon in Abschnitt 5.1. beschrieben, dürfen die vordefinierten Datentypen in den *Data-Definitionen* nur als Nachfolger eines einstelligen Konstruktors verwendet werden. Besitzt ein konstruktorbasierter Datentyp, der zur Beschreibung der abstrakten Syntax der Sprache $\underline{L}$ definiert wurde, einen einstelligen Konstruktor mit einem vordefinierten nichtkonstruktorbasierten Nachfolgertyp, so ist dem $\underline{Nutzer_{SBE(L)}}$ die textuelle Eingabe einer Instanz dieses vordefinierten nichtkonstruktorbasierten Nachfolgertyps genau unter folgender Bedingung erlaubt: Die Wurzel des aktuell ausgewählten Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$ muß den genannten konstruktorbasierten Typ aufweisen. Kann nach der Eingabe des $\underline{Nutzer_{SBE(L)}}$ der Eingabetext korrekt als ein Element des vordefinierten Datentyps gescannt werden, so wird der aktuell ausgewählte Teilbaum von $\underline{EK_L < AbstrakteSyntax_L >}$ durch den einstelligen Konstruktor mit dem gescannten Eingabetext als Nachfolger ersetzt. Durch dieses Vorgehen ist es nicht notwendig, die textuelle Eingabe des $\underline{Nutzer_{SBE(L)}}$ zu parsen, wenn sie in die interne Darstellungsform $\underline{EK_L < AbstrakteSyntax_L >}$ überführt werden soll (wie diese Überführung konkret realisiert wird, beschreiben wir in Abschnitt 6.1.4). Somit wurde eine Lösung gefunden, in dieser ersten Version des $\underline{Gen_{SBE}}$ auf die Beschreibung von Parse-Vorgängen innerhalb der $\underline{Spez_{SBE(L)}}$ durch den $\underline{Entwickler_{SBE(L)}}$ verzichten zu können.
Zum Schluß verweisen wir noch einmal auf unsere Beispielsprache $\underline{P}$. Durch die Verwendung der vordefinierten Typen in der Beschreibung der abstrakten Syntax von $\underline{P}$ wurde festgelegt, daß die textuelle Eingabe einer Instanz des Typs *IdentInput* (also eines Bezeichners) genau dann möglich ist, wenn der aktuell ausgewählte Teil von $\underline{EK_P}$ intern durch einen Teilbaum in $\underline{EK_P < AbstrakteSyntax_P >}$ repräsentiert wird, dessen Wurzel-Konstruktor den Typ `Identifier` besitzt. Besitzt der ausgewählte Teilbaum in $\underline{EK_P < AbstrakteSyntax_P >}$ den Typ `Exp`, so ist dem $\underline{Nutzer_{SBE(L)}}$ die textuelle Eingabe von Instanzen der vordefinierten Typen *IntInput*, *BoolInput*, *CharInput* und *QuotedStringInput* erlaubt.
Die regulären Ausdrücke, welche den für Haskell$^+$ vordefinierten nichtkonstruktorbasierten Typen zugrundeliegen, werden in Abschnitt 6.2.2 besprochen.
Zur Typkonvertierung stehen dem $\underline{Entwickler_{SBE(L)}}$ folgende vordefinierte Funktionen zur Verfügung:

```
intInputToInt :: IntInput -> Int
boolInputToBool :: BoolInput -> Bool
charInputToChar :: CharInput -> Char
stringInputToString :: StringInput -> String
identInputToString :: IdentInput -> String
quotedStringInputToString :: QuotedStringInput -> String

intToIntInput :: Int -> IntInput
boolToBoolInput :: Bool -> BoolInput
charToCharInput :: Char -> CharInput
stringToStringInput :: String -> StringInput
stringToIdentInput :: String -> IdentInput
stringToQuotedStringInput :: String -> QuotedStringInput
```

5.6 Hilfsfunktionen

Der $\underline{Entwickler_{SBE(L)}}$ hat zusätzlich die Möglichkeit, verschiedene Hilfsfunktionen zu definieren, um sie in seiner $\underline{Spez_{SBE(L)}}$ zu verwenden. Zur Beschreibung solcher Funktionen kann sich der $\underline{Entwickler_{SBE(L)}}$ sämtlicher Sprachkonstrukte von Haskell bedienen, da diese auch in Haskell$^+$ enthalten sind, denn die Sprache Haskell$^+$ stellt wie schon erwähnt eine Obermenge der Sprache Haskell dar.
In Abbildung 5.8 sind die in $\underline{Spez_{SBE(P)}}$ verwendeten Hilfsfunktionen abgebildet. Hiermit ist auch die Spezifikation $\underline{Spez_{SBE(P)}}$ des sprachbasierten Editors $\underline{SBE(P)}$ für unsere Beispielsprache $\underline{P}$ abgeschlossen.

```
lookuptyp :: Identifier -> DeclList -> TypExp
lookuptyp _ DECLLISTNIL = EMPTYTYPEXP
lookuptyp i (DECLLISTPAIR (DECLARATION idf t) dl)
  = if (i == idf) then t else (lookuptyp i dl)

isdeclared :: Identifier -> DeclList -> Bool
isdeclared _ DECLLISTNIL = False
isdeclared i (DECLLISTPAIR (DECLARATION idf _) dl)
  = if (i == idf) then True else (isdeclared i dl)

numberofdecls :: Identifier -> DeclList -> Int
numberofdecls _ DECLLISTNIL = 0
numberofdecls i (DECLLISTPAIR (DECLARATION idf _) dl)
  = (if (i == idf) then 1 else 0) + (numberofdecls i dl)

incomptypes :: TypExp -> TypExp -> Bool
incomptypes t1 t2 = (t1 /= EMPTYTYPEXP) && (t2 /= EMPTYTYPEXP) &&
  (t1 /= t2)



disBlock :: Stmt -> Stmt
disBlock t@(COMPOUND (STMTLISTPAIR t11 STMTLISTNIL)) = t11
disBlock t = t

ifthenelse :: Bool -> a -> a -> a
ifthenelse pred ta ea = if pred then ta else ea
```

Abbildung 5.8: Benötigte Hilfsfunktionen für die Beschreibung von $\underline{P}$

5.7 Änderungen der kontextsensitiven Nebenbedingungen der Sprache Haskell$^+$

Folgende in [Les99] eingeführten kontextsensitiven Nebenbedingungen werden nun aktualisiert:

CSR3 Die Namen der in den *Data-Blöcken* definierten Typen müssen im gesamten Haskell$^+$-Programm eindeutig sein; dabei gelten die nichtkonstruktorbasierten Typen *IntInput*, *BoolInput*, *CharInput*, *StringInput*, *IdentInput* und *QuotedStringInput* als (für die Sprache Haskell$^+$) vordefiniert, wobei diese aber nur als Nachfolger von einstelligen Konstruktoren verwendet werden dürfen.

CSR11 Die Angabe des Root-Symbols ist obligatorisch.

CSR21 Der (erste) Kontextparameter einer Transform-Funktion muß vom Typ TransSelection sein. Dazu ist die Definition des Typs TransSelection erforderlich.

CSR35 In der linken Seite einer Definitionsgleichung der Transform-Funktion gibt es immer genau einen Kontextparameter, der einen Konstruktor vom Typ TransSelection darstellt.

CSR39 Der Aufruf einer synthetischen, inheriten oder einer Unparse-Funktion ist nur zulässig, wenn es sich dabei um ein Außenvorkommen handelt. Ein Aufruf einer Unparse-Funktion ist nur in der rechten Seite einer Definitionsgleichungs dieser Unparse-Funktion erlaubt. Aufrufe der Transform-Funktion sind generell verboten.

Außerdem werden nachstehende kontextsensitive Nebenbedingungen denen aus [Les99] hinzugefügt:

CSR46 Die rechte Seite einer Definitionsgleichung einer Unparse-Funktion darf nur aus rekursiven Aufrufen dieser Unparse-Funktion, Stringkonstanten oder Aufrufen von Funktionen mit dem Ergebnistyp *String* bestehen, welche mit dem Infix-Operator ++ verknüpft sind.

CSR47 Innerhalb der Funktionsaufrufe und den Stringkonstanten, welche in einer rechten Seite einer Definitionsgleichung einer Unparse-Funktion vorkommen, dürfen ausschließlich die Steuerzeichen `\t`, `\b` und `\n` benutzt werden.

CSR48 In den Argumentpositionen der Funktionsaufrufe innerhalb einer rechten Seite einer Definitionsgleichung einer Unparse-Funktion dürfen keine Aufrufe einer Unparse-Funktion verwendet werden.

CSR49 Die Defaultkonstruktoren müssen so gewählt werden, daß es einerseits beim Aufbau des initialen Syntaxbaumes aus den Defaultkonstruktoren nicht zu Zyklen kommt und das andererseits deren Nachfolger keinen nichtkonstruktorbasierten Typ besitzen.

CSR50 Die *Mag-Blöcke*, in denen Unparse-Funktionen oder eine Transform-Funktion definiert werden, müssen alle diesselbe *Input-Definition* besitzten und dasselbe *Root-Symbol* definieren.

CSR51 Es muß mindestens eine Unparse-Funktion definiert sein.

CSR52 Es darf höchstens eine Transform-Funktion definiert werden.

CSR53 In den rechten Seiten der *Data-Definitionen* dürfen als Nachfolger der Konstruktoren keine Listen- oder Tupeltypen verwendet werden.

Die kontextsensitive Nebenbedingung CSR42 aus [Les99] wird nicht mehr benötigt und daher als obsolet betrachtet.

Zum Abschluß dieses Kapitels sei der $\underline{Entwickler_{SBE(L)}}$ darauf aufmerksam gemacht, daß die geänderten und neuen kontextsensitiven Nebenbedingungen der Sprache Haskell$^+$ (in der vorliegenden ersten Implementierungsversion des System $\underline{Gen_{SBE}}$) bei der Generierung eines $\underline{SBE(L)}$ aus der $\underline{Spez_{SBE(L)}}$ **nicht** abgetestet werden. Daher ist besondere Sorgfalt bei der Einhaltung dieser kontextsensitiven Nebenbedingungen notwendig, da sonst fehlerhafte Editoren erzeugt werden.

Kapitel 6

Implementation

In diesem Kapitel gehen wir konkret auf die Implementierung des $\underline{Gen_{SBE}}$ ein. Dabei konzentrieren wir uns auf die Erzeugung der Auswerterkomponente eines zu generierenden $\underline{SBE(L)}$.
Wir beschreiben dabei zuerst die bei der Generierung eines $\underline{SBE(L)}$ stattfindende Übersetzung der $\underline{Spez_{SBE(L)}}$ in den Haskell-Code $\underline{Spez_{SBE(L)} < Haskell >}$, welcher nach der Übersetzung als Haskell-Modul **Evaluator** vorliegt (Phase 3 der Generierung).
Danach gehen wir auf den Haskell-Code $\underline{UC_{AW} < Haskell >}$ ein, der den Teil der Auswerterkomponente des zu generierenden $\underline{SBE(L)}$ darstellt, welcher unabhängig von $\underline{Spez_{SBE(L)}}$ ist (Quell-Code für die Phase 5 in der Generierung des $\underline{SBE(L)}$).
Wir erläutern außerdem die Kommunikation zwischen der Auswerter- und der Nutzerschnittstellenkomponente des generierten $\underline{SBE(L)}$.
Im letzten Abschnitt dieses Kapitels beschreiben wir kurz die Implementierung des Java-Codes $\underline{UC_{NS} < Java >}$ (der als Quell-Code der Phase 6 in der Generierung des $\underline{SBE(L)}$ dient), welcher die Nutzerschnittstellenkomponente des $\underline{SBE(L)}$ implementiert.

6.1 Phase 3 - Übersetzung von Haskell$^+$ nach Haskell

Wir beschreiben hier die in Phase 3 stattfindende Übersetzung der $\underline{Spez_{SBE(L)}}$ in das Haskell-Modul **Evaluator**. Zu Beginn von Phase 3 liegt $\underline{Spez_{SBE(L)}}$ zwar schon in dem Format $\underline{Spez_{SBE(L)} < AbstrakteSyntax_{Haskell^+} >}$ vor, wir beziehen uns aber sowohl informell als auch formal bei der Beschreibung der Übersetzung auf das Ursprungsformat $\underline{Spez_{SBE(L)} < Haskell^+ >}$. Diese Entscheidung wurde getroffen, um dem Leser leichter die Grundideen der Erzeugung des Haskell-Moduls **Evaluator** vermitteln zu können, so daß er sich nicht mit den Einzelheiten der Implementierung für die Verarbeitung des Zwischenformats $\underline{Spez_{SBE(L)} < AbstrakteSyntax_{Haskell^+} >}$ befassen muß. Es sei hier nur erwähnt, daß die Verarbeitung des Formats $\underline{Spez_{SBE(L)} < AbstrakteSyntax_{Haskell^+} >}$ stark an die in [Les99] ausführlich beschriebene Handhabung dieses Programmformates angelehnt ist und der interessierte Leser die Möglichkeit hat, den entsprechenden Programmcode nachzulesen (er befindet sich auf der beigefügten Diskette).

6.1.1 Abstrakte Syntax

Die **Abstrakte Syntax** der Sprache $\underline{L}$ wird durch einen *Data-Block* beschrieben.

Übersetzung eines *Data-Blocks*

Die *Data-Definitionen* innerhalb eines *Data-Blocks* entsprechen der Haskell-Syntax und können daher unverändert übernommen werden. Allerdings wird zu jeder *Data-Definition* die Klausel *deriving (Show,Eq,Read)* hinzugefügt, damit die definierten Typen untereinander verglichen, als *String* aus- und von einem *String* eingelesen werden können.

Zur Realisierung der eingeschränkten Polymorphie in Haskell$^+$ wird ein neuer konstruktorbasierter Datentyp angelegt, der Vereinigungstyp. Dieser Datentyp verfügt ausschließlich über einstellige Konstruktoren, deren Nachfolger einen im *Data-Block* definierten Typ besitzen. Es wird für jeden Typ aus dem *Data-Block* genau ein solcher Konstruktor erzeugt. Durch Voranstellen des entsprechenden Konstruktors kann so ein Typ aus dem *Data-Block* in den Vereinigungstyp umgewandelt werden. Der Name des Vereinigungstyps setzt sich aus der Bezeichnung UnionType_ und dem Namen des *Data-Blocks* zusammen. Die einstelligen Konstruktoren aus dem Vereinigungstyp werden mit der Zusammensetzung von UT und dem Namen des Typs ihres Nachfolgers bezeichnet.

Zur Verdeutlichung sei hier ein Fragment des Data-Blocks der Beispielsprache $\underline{P}$ aus Kapitel 4, sowie deren Übersetzung nach Haskell angegeben.

Haskell$^+$-Code:

```
begindata P_Data
  ...
  data Identifier = IDENTIFIERNULL
                  | IDENTIFIER IdentInput
  ...
enddata
```

Übersetzung:

```
...
data Identifier =
  IDENTIFIERNULL | IDENTIFIER IdentInput deriving (Show,Eq,Read)
...
data UnionType_P_Data =
  ... | UTIdentifier Identifier | ...deriving (Show,Eq,Read)
```

Da die für Haskell$^+$ vordefinierten Typen (*IdentInput* etc.) in Haskell nicht definiert sind, ist die Generierung folgender Typdeklarationen notwendig:

```
type IntInput = String
type BoolInput = String
type CharInput = String
```

```
type StringInput = String
type IdentInput = String
type QuotedStringInput = String
```

Wir erläutern in Abschnitt 6.2.2, warum alle diese Typen im zu erzeugenden Haskell-Programm als Typ-Synonyme für den Typ String verwendet werden.

6.1.2 Attributierung

Übersetzung der synthetischen und inheriten Funktionen

Die Methodik bei der Übersetzung der synthetischen und inheriten Funktionen wurde schon in [Les99] ausführlich beschrieben. Da wir hier dieselbe Übersetzungsmethode verwenden, sei sie nur kurz skizziert.

Die in einem *Mag-Block* definierten synthetischen und inheriten Funktionen besitzen als Rekursionsargument einen Teilbaum des sogenannten Eingabebaums. Ein solcher Teilbaum besitzt immer einen konstruktorbasierten Typ, der im *Input-Data-Block* des *Mag-Blocks* definiert worden ist. Ist der *Input-Data-Block* jener, in dem die abstrakte Syntax der Sprache L definiert wurde, so ist der Eingabebaum unser $\underline{EK_L < AbstrakteSyntax_L >}$.

Mit den synthetischen und inheriten Funktionen erfolgt ein Informationstransfer auf dem Eingabebaum. Bei Anwendung von synthetischen Funktionen wird im Eingabebaum "hinabgestiegen", während bei der Anwendung von inheriten Funktionen im Eingabebaum "hinaufgestiegen" wird. Die Problematik des "Hinaufsteigens" im Eingabebaum wird durch eine geeignete Übersetzung des Rekursionsargumentes der synthetischen und inheriten Funktionen gelöst.

In der Haskell-Übersetzung der synthetischen und inheriten Funktionen besitzen die Funktionen als Rekursionsargument ein Tupel aus einer Vereinigungstyp-Liste und einer Integer-Liste. In der Vereinigungstyp-Liste werden alle Knoten (bzw. die Teilbäume, welche als Wurzel diese Knoten besitzen) des aktuellen Pfades gehalten. Die Integer-Liste repräsentiert diesen aktuellen Pfad. Aus Gründen der Effizienz bei der Listenmanipulation in Haskell wird der Pfad dabei rückwärts gehalten.

Die Funktionsdefinition einer synthetischen, wie auch einer inheriten Funktion besteht aus einer linken und einer rechten Seite. Bei der Betrachtung beider Seiten ist auf die Festlegungen bei Makroattributgrammatiken im Hinblick auf Innen- und Außenvorkommen von synthetischen und inheriten Funktionen zu achten.

Weiterhin sei auf die notwendige Einbindung des Rekursionsargumentes der Funktionen in den passenden Vereinigungstyp hingewiesen. In den nun folgenden formalen Beschreibungen bezeichne dazu $\langle VTK\ von\ \mathrm{t}\rangle$ den Konstruktor aus dem Vereinigungstyp, dessen Nachfolgertyp mit dem Typ von t übereinstimmt.

<u>Übersetzung der linken Seiten</u>

Sei synfun der Name einer $(m+1)$-stelligen synthetischen Funktion, angewendet auf den n-stelligen Konstruktor Kon, mit $0 \leq m$ und $0 \leq n$:

- Linke Seite der Definitionsgleichung einer synthetischen Funktion

  t@(Kon t_1 .. t_n) in
  synfun t y_1 .. y_m = ...

- Dazugehörige Haskell-Übersetzung

  synfun ta@(t0@(($\langle$*VTK von* t$\rangle$ t@(Kon t_1 .. t_n)): ts), z) y_1 .. y_m = ...

Findet in der rechten Seite der Definitionsgleichung einer synthetischen Funktion kein Aufruf einer inheriten Funktion statt, so kann die linke Seite der Definitionsgleichung einer synthetischen Funktion auch die Gestalt
synfun (Kon t_1 .. t_n) y_1 .. y_m = ...
besitzen, was aber keinen Einfluß auf ihre Übersetzung hat.

Sei inhfun der Name einer $(m+1)$-stelligen inheriten Funktion, angewendet auf den j-ten Nachfolger des n-stelligen Konstruktors Kon, mit $0 \leq m$, $1 \leq n$ und $1 \leq j \leq n$:

- Linke Seite der Definitionsgleichung einer inheriten Funktion

  t@(Kon t_1 .. t_n) in
  inhfun t_j y_1 .. y_m = ...

- Dazugehörige Haskell-Übersetzung

  inhfun ta@((_ :t0@(($\langle$*VTK von* t$\rangle$ t@(Kon t_1 .. t_n)): ts)), (j:z)) y_1 .. y_m = ...

Die in den Übersetzungen verwendeten As-Pattern werden verwendet, damit wir uns in den rechten Seiten der Funktionsdefinitionen darauf beziehen können. Die Methodik der Übersetzung einer linken Seite einer Definitionsgleichung wird uns später noch bei anderen Funktionstypen wiederbegegnen. Daher werden der Gleichbehandlung wegen, auch AS-Pattern verwendet, welche hier in den Übersetzungen der rechten Seiten nicht benötigt werden (z.B. AS-Pattern ta).

Übersetzung von Aufrufen synthetischer und inheriter Funktionen in den rechten Seiten

Die rechten Seiten der synthetischen und inheriten Funktionsdefinitionen entsprechen schon der Haskell-Syntax, mit Ausnahme von Aufrufen synthetischer und inheriter Funktionen. Wir beschreiben nun, wie diese Aufrufe bei der Übersetzung der rechten Seiten behandelt werden müssen.

Sei synfun der Name einer $(m+1)$-stelligen synthetischen Funktion, angewendet auf den j-ten Nachfolger des n-stelligen Konstruktors Kon, mit $0 \leq m$, $1 \leq n$ und $1 \leq j \leq n$:

- Aufruf einer synthetischen Funktion in einer rechten Seite

  t@(Kon t_1 .. t_n) in
  ... = ... (synfun t_j y_1 .. y_m) ...

- Dazugehörige Haskell-Übersetzung

  ... = ... (synfun (((⟨VTK *von* t_j⟩ t_j): t0), (j:z)) y_1 .. y_m) ...

Sei inhfun der Name einer (m+1)-stelligen inheriten Funktion, angewendet auf den n-stelligen Konstruktor Kon, mit $0 \leq m$ und $0 \leq n$:

- Aufruf einer inheriten Funktion in einer rechten Seite

  t@(Kon t_1 .. t_n) in

  ... = ... (inhfun t y_1 .. y_m) ...

- Dazugehörige Haskell-Übersetzung

  ... = ... (inhfun (t0, z) y_1 .. y_m) ...

6.1.3 Ausgabesyntax

In diesem Unterabschnitt beschreiben wir die Realisierung der textuellen Darstellung von $\underline{EK_L}$ für den $\underline{Nutzer_{SBE(L)}}$. Für die dazu notwendige Umwandlung von $\underline{EK_L}$ aus der internen Darstellungsform $\underline{EK_L < AbstrakteSyntax_L >}$ in die externe Darstellungsform $\underline{EK_L < Ausgabesyntax_L >}$ enthält die $\underline{Spez_{SBE(L)} < Haskell^+ >}$ die Definition einer bzw. mehrerer Unparse-Funktionen. Falls mehrere Unparse-Funktionen definiert worden sind, so geschah dies unabhängig voneinander, weil sie auch unabhängig voneinander verwendet werden sollen. Daher beziehen wir uns im folgenden der Einfachheit halber immer nur auf eine Unparse-Funktion.
Es sei noch einmal daran erinnert, daß eine in Haskell$^+$ innerhalb eines *Mag-Blocks* spezifizierte Unparse-Funktion auch als spezielle synthetische Funktion betrachtet werden kann, die keine Kontextparameter und stets den Ergebnistyp String besitzt. Dazu sei vermerkt, daß hier und auch in den folgenden Abschnitten dieses Kapitels das Rekursionsargument einer Funktion (wir nennen den Parameter an der ersten Argumentposition der Funktion Rekursionsargument, wenn die Funktion eine Rekursion auf ihm ausführt) nicht als Kontextparameter gezählt wird. Mit Kontextparameter bezeichnen wir alle anderen Parameter der Funktion.

Übersetzung einer Unparse-Funktion

Wir betrachten an dieser Stelle zuerst noch einmal genauer die von einer Unparse-Funktion zu bewältigenden Aufgaben innerhalb des zu erzeugenden $\underline{SBE(L)}$. Dabei kristallisieren sich die zwei folgenden Hauptaufgaben heraus:

1. Umwandlung der Darstellungsform von $\underline{EK_L}$ von $\underline{EK_L < AbstrakteSyntax_L >}$ nach $\underline{EK_L < Ausgabesyntax_L >}$:

   Dabei ist zu beachten, daß während das $\underline{EK_L}$ vom $\underline{Nutzer_{SBE(L)}}$ editiert wird, zu jedem Zeitpunkt ein Teil von $\underline{EK_L}$ als aktuell ausgewählt gilt. Diese vom $\underline{Nutzer_{SBE(L)}}$

getroffene Auswahl soll sich in der textuellen Darstellung $\underline{EK_L < Ausgabesyntax_L >}$ widerspiegeln können.

2. Generierung eines bestimmten Teils der Umwandlung von $\underline{EK_L < EditBefehl_L{}^+ >}$ nach $\underline{EK_L < AbstrakteSyntax_L >}$:

   Die schon in Aufgabe 1 besprochene Auswahl eines Teiles von $\underline{EK_L}$ durch den $\underline{Nutzer_{SBE(L)}}$ erfolgt innerhalb von $\underline{EK_L < Ausgabesyntax_L >}$. Damit nun aber eine neue vom $\underline{Nutzer_{SBE(L)}}$ getätigte Auswahl in die interne Darstellungsform $\underline{EK_L < AbstrakteSyntax_L >}$ übernommen werden kann, ist eine Abbildung von der getätigten Auswahl in $\underline{EK_L < Ausgabesyntax_L >}$ nach der Auswahl im Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ notwendig. Diese Abbildung soll von der Unparse-Funktion während ihrer Anwendung generiert werden können.

Daraus ergeben sich folgende Gründe für die Übersetzung einer vom $\underline{Entwickler_{SBE(L)}}$ spezifizierten Unparse-Funktion:

- Das Rekursionsargument einer Unparse-Funktion muß so übersetzt werden, daß in der rechten Seite der Übersetzung einer Unparse-Funktionsdefinition Aufrufe von den nach Haskell übersetzten synthetischen und inheriten Funktionen möglich sind.
- Die Ausgabe der textuellen Darstellungsform $\underline{EK_L < Ausgabesyntax_L >}$, das Entgegennehmen der vom $\underline{Nutzer_{SBE(L)}}$ getätigten Auswahl eines Teils von $\underline{EK_L}$ sowie deren Umwandlung in eine von der Auswerterkomponente des $\underline{SBE(L)}$ interpretierbare Anweisung soll von der Nutzerschnittstellenkomponente des $\underline{SBE(L)}$ realisiert werden. Die dafür notwendigen Informationen soll sie durch Anwendung der Unparse-Funktion erhalten. Die in Haskell$^+$ definierte Unparse-Funktion kann aber nicht alle diese Informationen erzeugen, da sie die Aufgabe 1 nur unvollständig (in bezug auf den in $\underline{EK_L}$ aktuell ausgewählten Teil) und Aufgabe 2 gar nicht realisiert.

Das gewünschte Ergebnis der Übersetzung einer vom $\underline{Entwickler_{SBE(L)}}$ in Haskell$^+$ programmierten Unparse-Funktion ist eine in Haskell spezifizierte Unparse-Funktion, die einerseits kompatibel zu den anderen Teilen der Übersetzung von $\underline{Spez_{SBE(L)}}$ ist und sich andererseits dazu in der Lage befindet, eine Datei zur Kommunikation mit der Nutzerschnittstelle von $\underline{SBE(L)}$ zu erzeugen, damit Aufgabe 1 und 2 erfüllt werden können.

<u>Grundidee der Übersetzung einer Unparse-Funktion</u>

Die vom $\underline{Entwickler_{SBE(L)}}$ definierte Unparse-Funktion baut während ihrer rekursiven Anwendung auf $\underline{EK_L < AbstrakteSyntax_L >}$ aus den Strings, welche sich aus der Auswertung der rechten Seiten der Unparse-Funktionsdefinitionen ergeben, eine Zeichenkette auf, die $\underline{EK_L < Ausgabesyntax_L >}$ ohne Hervorhebungen darstellt.
Im Unterschied dazu soll durch die Anwendung der nach Haskell übersetzten Unparse-Funktion auf $\underline{EK_L < AbstrakteSyntax_L >}$ schrittweise die besprochene Kommunikationsdatei für die Nutzerschnittstelle von $\underline{SBE(L)}$ erzeugt werden. Die in die Datei zu schreibenden Informationen sollen zeilenweise kodiert werden, wobei immer drei aufeinanderfolgende Zeilen eine Informationseinheit bilden. In einer solchen Informationseinheit sind

zusätzlich zu jedem auszugebenden Teilstring von $\underline{EK_L < Ausgabesyntax_L >}$ die Informationen über den Darstellungsmodus für diesen Teilstring (vgl. Aufgabe 1) sowie über den zu diesem Teilstring korrespondierenden Knoten aus $\underline{EK_L < AbstrakteSyntax_L >}$ (vgl. Aufgabe 2) enthalten.

Betrachten wir nun informell die Methodik der Übersetzung einer Unparse-Funktion in bezug auf die gewünschte Erzeugung der Kommunikationsdatei:

Sei upfun eine vom $\underline{Entwickler_{SBE(L)}}$ in Haskell$^+$ spezifizierte (primäre) Unparse-Funktion und die Definition von upfun angewendet auf den n-stelligen Konstruktor Kon mit $0 \leq n$ habe folgende Gestalt:

$$
\begin{array}{l}
\mathrm{t@(Kon\ t_1\ ..\ t_n)\ in} \\
\qquad \mathrm{upfun\ t = s_0\ {+}{+}\ (upfun\ t_1)\ {+}{+}\ s_1\ {+}{+}\ ..\ {+}{+}\ s_{(n-1)}\ {+}{+}\ (upfun\ t_n)\ {+}{+}\ s_n}
\end{array}
$$

(Natürlich ist die beschriebene Reihenfolge der rekursiven Aufrufe der Unparse-Funktion nicht erforderlich, es können die rekursiven Aufrufe auch fehlen oder mehrfach vorkommen - dies ist aber für die hier vorgestellte Übersetzungsmethodik nicht relevant und daher beziehen wir uns der Verständlickeit wegen auf oben genannte Gestalt der Unparse-Funktionsdefinition)

Weiterhin gelte $0 \leq r \leq n$ und s_r bezeichne entweder eine beliebige Verknüpfung von Stringkonstanten und/oder Aufrufen von Funktionen (mit dem Rückgabewert String) durch den Infix-Operator "++" oder den leeren String.
Der String $\mathrm{s}_r{}'$ bezeichnet den String, der durch Auswertung aller in s_r vorkommenden Funktionsaufrufe und Anwendung der Verknüpfungsoperatoren entsteht.

Die Unparse-Funktion upfun soll so in eine Haskell-Funktion übersetzt werden, daß beim Aufruf der übersetzten Funktion auf den n-stelligen Konstruktor Kon genau $(n+1)$-mal je drei zusammengehörende Zeilen in der Kommunikationsdatei erzeugt werden, welche die notwendigen Informationen über $\mathrm{s}_r{}'$ enthalten und folgende Gestalt haben:

1. Zeile : Darstellungsmodus des auszugebenden Strings $\mathrm{s}_r{}'$

   "H" falls der Knoten aus $\underline{EK_L < AbstrakteSyntax_L >}$ mit der Bezeichnung Kon, auf den die Unparse-Funktion upfun gerade angewendet wird, zum aktuell ausgewählten Teilbaum von $\underline{EK_L < AbstrakteSyntax_L >}$ gehört und damit $\mathrm{s}_r{}'$ zum ausgewählten Teil von $\underline{EK_L < Ausgabesyntax_L >}$ zählt. In diesem Fall soll daher $\mathrm{s}_r{}'$ in der Ausgabe von $\underline{EK_L < Ausgabesyntax_L >}$ besonders hervorgehoben dargestellt werden

   "L" andernfalls, was bedeutet, daß $\mathrm{s}_r{}'$ nicht von der Nutzerschnittstelle des $\underline{SBE(L)}$ besonders hervorgehoben dargestellt werden soll

2. Zeile : Information über den Pfad (nicht die komplette Pfadkodierung selbst) in $\underline{EK_L < AbstrakteSyntax_L >}$ zum Knoten mit der Bezeichnung Kon, auf den die Unparse-Funktion upfun gerade angewendet wird.

   "*" falls $r = 0$ und Konstruktor Kon ist das *Root-Symbol*, also der Wurzelknoten in $\underline{EK_L < AbstrakteSyntax_L >}$

"m" falls $r = 0$ und Konstruktor Kon ist m-ter Nachfolger seines Vaterknotens in $\underline{EK_L < AbstrakteSyntax_L >}$

"0" falls $r \geq 0$

Mit dieser Information kann der Pfad als (rückwärts kodierte) Liste von ganzen Zahlen von der Nutzerschnittstelle inkrementell berechnet werden.

3. Zeile : auszugebender String $s_r{}'$

Damit die übersetzte Unparse-Funktion in solcher Weise die Kommunikationsdatei erzeugen kann, benötigt sie die folgenden zwei Kontextparameter:

- der erste Kontextparameter (sel genannt) besitzt den Typ Integer-Liste und repräsentiert den Pfad im Eingabebaum $\underline{EK_L < AbstrakteSyntax_L >}$ zur Wurzel des Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$, welcher als aktuell ausgewählt gilt. Er wird aufgrund der Effizienz bei Listenmanipulationen rückwärts gehalten. Notwendig ist er zur Berechnung der Darstellungsmodi.
- der zweite Kontextparameter (fH genannt) ist vom Typ Handle und verweist auf die zu beschreibende Kommunikationsdatei.

Wir beschreiben nun formal die Übersetzung der Definitionsgleichungen einer Unparse-Funktion:

<u>Übersetzung der linken Seiten</u>

Die linke Seite einer Unparse-Funktionsdefinition wird in bezug auf das Rekursionsargument wie die linke Seite einer synthetischen Funktion übersetzt. Außerdem werden die beiden Kontextparameter *sel* und *fH* hinzugefügt.

Sei upfun der Name einer einstelligen Unparse-Funktion, angewendet auf den n-stelligen Konstruktor Kon, mit $0 \leq n$:

- Linke Seite der Definitionsgleichung einer Unparse-Funktion

  t@(Kon t_1 .. t_n) in
  upfun t = ...

- Dazugehörige Haskell-Übersetzung

  upfun ta@(t0@((⟨*VTK von* t⟩ t@(Kon t_1 .. t_n)): ts), z) sel fH = ...

Durch ⟨*VTK von* t⟩ beschreiben wir wieder den Konstruktor aus dem Vereinigungstyp, dessen Nachfolgertyp mit dem Typ von t übereinstimmt.

<u>Übersetzung der rechten Seiten</u>

Sei wieder upfun der Name einer einstelligen Unparse-Funktion, angewendet auf den n-stelligen Konstruktor Kon, mit $0 \leq n$.

Sei weiterhin $1 \leq q$, $1 \leq i \leq q$ und rst_i
- eine Stringkonstante oder
- ein rekursiver Aufruf (upfun t_j) dieser Unparse-Funktion mit $1 \leq j \leq n$ oder
- ein erlaubter Aufruf einer anderen Funktion mit dem Ergebnistyp String.

- Rechte Seite der Definitionsgleichung der Unparse-Funktion

  ... = rst_1 ++ .. ++ rst_q

- Dazugehörige Haskell-Übersetzung

  ... = do ⟨*berechne* sdm *aus* z *und* sel⟩
  ⟨*schreibe in Zeile der Datei* fH sdm *und beende sie*⟩
  ⟨*schreibe in Zeile der Datei* fH ⟨*kopf von* z⟩*und beende sie*⟩
  ⟨*U-Übersetzung von* rst_1⟩
  ..
  ⟨*U-Übersetzung von* rst_q⟩
  ⟨*beende Zeile in Datei* fH⟩

Die Variable `sdm` bekommt als Wert den Darstellungsmodus des auszugebenden Strings (siehe obige Erklärung zur ersten von drei Zeilen, welche eine Informationseinheit bilden). Mit ⟨*kopf von* z⟩ ist die Stringdarstellung des ersten Elementes der Integer-Liste `z` gemeint, bzw. "*", falls `z` die leere Liste repräsentiert.
Die Bedeutung von ⟨*U-Übersetzung von* rst_i⟩ wird durch folgende Fallunterscheidung erklärt:

Fall 1 : rst_i ist rekursiver Aufruf (upfun t_j) dieser Unparse-Funktion

⟨*U-Übersetzung von* rst_i⟩ :
⟨*beende Zeile in Datei* fH⟩
(upfun (((⟨*VTK von* t_j⟩ t_j): t0), (j:z)) sel fH)
⟨*schreibe in Zeile der Datei* fH sdm *und beende sie*⟩
⟨*schreibe in Zeile der Datei* fH `"0"` *und beende sie*⟩

Fall 2 : rst_i ist kein rekursiver Aufruf einer Unparse-Funktion

⟨*U-Übersetzung von* rst_i⟩ :
⟨*schreibe in Zeile derDatei* fH ⟨*Übersetzung von* rst_i⟩⟩

Bei ⟨*Übersetzung von* rst_i⟩ wird der Haskell^{+}-Code rst_i identisch übernommen, bis auf die eventuell enthaltenen Aufrufe von synthetischen und inheriten Funktionen. Diese werden, wie schon in Abschnitt 5.1.2 beschrieben, übersetzt.

Zum besseren Verständnis folgt nun ein Beispiel der Übersetzung einer Unparse-Funktionsdefinition aus unserer Beispielsprache *P*:

Unparse-Funktionsdefinition in Haskell$^+$

```
t@(ASSIGN t1 t2) in
  up t = (up t1) ++
         (ite
              ((t1 == IDENTIFIERNULL) ||
               (isdeclared t1 (env t)))
              ""
              " { Not Declared } ") ++
         " := " ++
         (up t2) ++
         (ite
              (incomptypes
                  (typ t1)
                  (typ t2))
              " { Incompatibles Types In := } "
              "")
```

nach Haskell übersetzte Unparse-Funktionsdefinition

```
up ta@(t0@((UTStmt t@(ASSIGN t1 t2)):ts),z) sel fH =
  do sdm <- return (partOfSelection z sel)
     wTFPSLn fH sdm
     wTFPSLn fH (pathHead z)
     wTFPSLn fH ""
     up (((UTIdentifier t1):t0),(1:z)) sel fH
     wTFPSLn fH sdm
     wTFPSLn fH "0"
     wTFPS fH (ite
                   ((t1 == IDENTIFIERNULL) ||
                    (isdeclared t1 (env (t0,z))))
                   ""
                   " { Not Declared } ")
     wTFPS fH " := "
     wTFPSLn fH ""
     up (((UTExp t2):t0),(2:z)) sel fH
     wTFPSLn fH sdm
     wTFPSLn fH "0"
     wTFPS fH (ite
                   (incomptypes
                       (typ (((UTIdentifier t1):t0),(1:z)))
                       (typ (((UTExp t2):t0),(2:z))))
                   " { Incompatibles Types In := } "
                   "")
     wTFPSLn fileHandle ""
```

Erläuterungen zu den im Haskell-Code verwendeten Hilfsfunktionen:

partOfSelection :: [Int] → [Int] → String
(partOfSelection z sel) liefert "H", falls sel ein Postfix von z ist, ansonsten "L"

wTFPS :: Handle → String → IO ()
(wTFPS fH s) schreibt s' in die Datei fH, wobei s' aus s entsteht durch Substitution von
"\n" mit "\\n"
"\t" mit "\\t"
"\b" mit "\\b"

wTFPSLn ist analog zu wTFPS, nur daß die aktuell beschriebene Zeile beendet wird

pathHead :: [Int] → String
pathHead [] = "*"
pathHead (x:xs) = (show x)

Zum Schluß dieses Abschnittes besprechen wir noch die Notwendigkeit der Generierung einer Funktion zum Aufruf einer Unparse-Funktion. Diese Funktion, welche wir `upFunCall` nennen, wird so generiert, daß sie die Namen aller in $\underline{Spez_{SBE(L)} < Haskell^+ >}$ definierten Unparse-Funktionen kennt und daher in der Lage ist, diese aufzurufen. Bevor `upFunCall` eine Unparse-Funktion aufruft, öffnet sie die Kommunikationsdatei "unparsing.icf" im Schreibmodus und übergibt das so gewonnene Handle auf die Datei als zweiten Kontextparameter an die aufzurufende Unparse-Funktion. Nachdem die Unparse-Funktion ausgewertet worden ist, schließt `upFunCall` die nun fertigbeschriebene Kommunikationsdatei.
In der derzeitigen Version der Implementierung des $\underline{Gen_{SBE(L)}}$ werden nur solche Editoren $\underline{SBE(L)}$ erzeugt, bei denen der $\underline{Nutzer_{SBE(L)}}$ keine Möglichkeit hat, eine bestimmte Unparse-Funktion und damit eine bestimmte Ausgabesyntax der Sprache $\underline{L}$ auszuwählen, falls der $\underline{Nutzer_{SBE(L)}}$ in der $\underline{Spez_{SBE(L)}}$ mehrere Unparse-Funktionen definiert hat. Es wird immer die in der $\underline{Spez_{SBE(L)}}$ zuerst definierte Unparse-Funktion (und damit die zuerst definierte Ausgabesyntax von $\underline{L}$) verwendet.

Bleibt die Frage, wann überhaupt eine Unparse-Funktion aufzurufen ist während der $\underline{Nutzer_{SBE(L)}}$ mit dem $\underline{SBE(L)}$ das $\underline{EK_L}$ editiert. Die Frage soll jetzt noch offen bleiben, sie wird in Abschnitt 6.1.5 geklärt. Es sei hier nur erwähnt, daß aus $\underline{Spez_{SBE(L)}}$ eine sogenannte Travers-Funktion generiert werden muß, die $\underline{EK_L < AbstrakteSyntax_L >}$ rekursiv durchläuft und dabei an der Wurzel des Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$, welcher den aktuell ausgewählten Teil von $\underline{EK_L}$ repräsentiert, ihren Durchlauf stoppt. Die Travers-Funktion ruft dann unter noch zu beschreibenden Bedingungen die Funktion `upFunCall` auf und übergibt ihr als Kontextparameter den Pfad im Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ zu dem Knoten, an welchem die Travers-Funktion gerade verweilt. Dieser Kontextparameter wird dann von der Funktion `upFunCall` an die gewählte Unparse-Funktion als erster Kontextparameter übergeben.

6.1.4 Strukturelle Transformationen

Hier beschreiben wir die implementationstechnische Umsetzung der strukturellen Transformationen, die mit Hilfe eines $\underline{SBE(L)}$ durchgeführt werden können. Die Funktionalität dieser Transformationen wurde durch den $\underline{Entwickler_{SBE(L)}}$ in der $\underline{Spez_{SBE(L)}}$ mit Hilfe der Definition einer Transform-Funktion, welche man auch als spezielle synthetische Funktion sehen kann, innerhalb eines *Mag-Blockes* spezifiziert.

Zuerst betrachten wir wieder etwas genauer die zu realisierende Aufgabe der Transform-Funktion in dem zu erzeugenden $\underline{SBE(L)}$:

Die Definitionsgleichungen der Transform-Funktion in $\underline{Spez_{SBE(L)}}$ beschreiben je eine Transformation eines Teilbaumes von $\underline{EK_L < AbstrakteSyntax_L >}$ in Abhängigkeit seines Wurzel-Konstruktors (Rekursionsargument der Funktion) und einer Bezeichnung für die Transformation (Kontextparameter der Funktion) in einen neuen Teilbaum des Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$. Die Aufgabe der Transform-Funktion ist aber die Realisierung des strukturellen Transformierens von $\underline{EK_L}$, d.h. die Transformation des gesamten $\underline{EK_L < AbstrakteSyntax_L >}$, wobei aber nur der genannte Teilbaum durch den neuen Teilbaum ersetzt werden soll.
Weiterhin muß beachtet werden, daß der $\underline{Entwickler_{SBE(L)}}$ bei der Beschreibung der strukturellen Transformationen die Möglichkeit hat, in der rechten Seite einer Definitionsgleichung einer Transform-Funktion auch Aufrufe von synthetischen und inheriten Funktion zu verwenden.

Daraus ergeben sich folgende Gründe für eine Übersetzung der vom $\underline{Entwickler_{SBE(L)}}$ spezifizierten Transform-Funktion:

- Das Rekursionsargument der Transform-Funktion muß (wie schon bei den Unparse-Funktionen) in einer solchen Weise übersetzt werden, daß Aufrufe von synthetischen und inheriten Funktionen in der Übersetzung der Transform-Funktionsdefinitionen realisierbar sind.

- Sei eine Definitionsgleichung der Transform-Funktion (welche den Kontextparameter K verwendet) die Beschreibung einer Transformation (mit dem Namen K) des durch seinen Wurzel-Konstruktor Kon bestimmten Teilbaums t des Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$ in den Teilbaum t'.
  Die Definitionsgleichung soll nun eine Transformation (mit dem Namen K) des gesamten Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$ ermöglichen, wenn t der aktuell ausgewählte Teilbaum des Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$ ist und man als Transformationsergebnis den Baum erhält, der durch Ersetzung von t durch t' in $\underline{EK_L < AbstrakteSyntax_L >}$ entsteht.

<u>Grundidee der Übersetzung</u>

Wir generieren aus $\underline{Spez_{SBE(L)}}$ eine Funktion `execTrans`, welche die Transformation des gesamten Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$ in der gerade besprochenen Weise übernehmen kann. Sie arbeitet rekursiv auf $\underline{EK_L < AbstrakteSyntax_L >}$ und ruft an geeigneter Stelle die übersetzte Transform-Funktion auf. Damit sie die Transform-Funktion

in der Weise aufrufen kann, daß diese wiederum Aufrufe von synthetischen und inheriten Funktionen gestattet, muß ihr Rekursionsargument vom gleichem Typ sein, wie das der synthetischen und inheriten Funktionen.
Weiterhin besitzt die Funktion `execTrans` zwei Kontextparameter. Der erste Kontextparameter ist eine Integer-Liste, die den Pfad zum Wurzelknoten des aktuell ausgewählten Teilbaums aus $\underline{EK_L < AbstrakteSyntax_L >}$ repräsentiert, welcher durch Anwendung der Transform-Funktion transformiert werden soll.
Der zweite Kontextparameter repräsentiert die Bezeichnung der Transformation, die von der Transform-Funktion ausgeübt werden soll. Dieser Kontextparameter ist der Konstruktor TTTransCommand vom Typ TransType, welchen wir im folgenden noch genauer erklären werden. Der Konstruktor TTTransCommand ist einstellig und bekommt als Nachfolger einen nullstelligen Konstruktor vom Typ TransSelection. Dieser Typ wurde vom $\underline{Entwickler_{SBE(L)}}$ in $\underline{Spez_{SBE(L)}}$ zur Beschreibung des Kontextparameters der von ihm spezifizierten Transform-Funktionsdefinitionen definiert. Der zweite Kontextparameter der Funktion `execTrans` wird der Transform-Funktion bei ihrem Aufruf als einziger Kontextparameter übergeben.
Der Ergebnistyp der Funktion `execTrans` ist der Vereinigungstyp der Typen, welche die abstrakte Syntax der Sprache $\underline{L}$ beschreiben. Deshalb ist bei der Ergebniserzeugung während der rekursiven Anwendung der Funktion `execTrans` auf die notwendige Ein- bzw. Ausbettung der in $\underline{EK_L < AbstrakteSyntax_L >}$ verwendeten Typen in bzw. aus dem genannten Vereinigungstyp zu achten.
Bevor wir die formale Beschreibung der generierten Definitionsgleichungen der Funktion `execTrans` angeben, sei noch bemerkt, daß die Transform-Funktion nicht auf Teilbäume (bzw. Blätter) von $\underline{EK_L < AbstrakteSyntax_L >}$ angewendet werden darf, deren Wurzelknoten von einem nichtkonstruktorbasierten Typ (*IntInput*,*BoolInput* etc.) sind. Daher ist ein rekursiver Abstieg der Funktion `execTrans` zu derartigen Teilbäumen von $\underline{EK_L < AbstrakteSyntax_L >}$ nicht sinnvoll und wird daher ausgeschlossen.

Generierung der Funktion `execTrans`

Sei DB der in $\underline{Spez_{SBE(L)}}$ spezifizierte *Data-Block*, in dem die abstrakte Syntax der Sprache $\underline{L}$ beschrieben wird. Sei $n \geq 0$, dann wird für alle n-stelligen Konstruktoren Kon, die in *Data-Definitionen* von DB verwendet werden, folgende Definitionsgleichung der Funktion `execTrans` generiert:
Dabei gelte entweder

- $k = 0$, wenn $n = 1$ und der erste Nachfolger von Kon besitzt einen nichtkonstruktorbasierten Typ oder
- $k = n$, sonst

Generierte Definitionsgleichung

```
execTrans ta@(t0@((⟨VTK von t⟩ t@(Kon t_1 .. t_n)): ts), z) path command =
  case path of
    []          → (transFunCall ta command)
    (1:rpath)   → (⟨VTK von t⟩ Kon
                              (case (execTrans
                                        (((⟨VTK von t_1⟩ t_1): t0), (1:z))
                                        rpath
                                        command) of
                                  ((⟨VTK von t_1⟩ t') → t'))
                              t_2
                              ⋮
                              t_k)
    ⋮
    (j:rpath)   → (⟨VTK von t⟩ Kon
                              t_1
                              ⋮
                              t_{j-1}
                              (case (execTrans
                                        (((⟨VTK von t_j⟩ t_j): t0), (j:z))
                                        rpath
                                        command) of
                                  ((⟨VTK von t_j⟩ t') → t'))
                              t_{j+1}
                              ⋮
                              t_k)
    ⋮
    (k:rpath)   → (⟨VTK von t⟩ Kon
                              t_1
                              ⋮
                              t_{k-1}
                              (case (execTrans
                                        (((⟨VTK von t_k⟩ t_k): t0), (k:z))
                                        rpath
                                        command) of
                                  ((⟨VTK von t_k⟩ t') → t')))
```

Dabei gelte $1 \leq j \leq k$. Durch $\langle VTK\ von\ \mathrm{t}\rangle$ beschreiben wir wiederum den Konstruktor aus dem Vereinigungstyp, dessen Nachfolgertyp mit dem Typ von t übereinstimmt.

Nachdem wir nun die Funktion `execTrans` zur Verfügung haben, ist es nicht schwer, die Übersetzung der Definition der Transform-Funktion zu beschreiben. Das Rekursionsargument wird analog zur Übersetzung des Rekursionsargumentes einer synthetischen Funktion übersetzt. Der Kontextparameter der vom $\underline{Entwickler_{SBE(L)}}$ spezifizierten Transform-Funktionsdefinition wird in den Typ TransType eingebettet.
Die rechte Seite wird in den Vereinigungstyp eingebettet, wobei die in ihr eventuell vorhandenen Aufrufe von synthetischen oder inheriten Funktionen vorher übersetzt werden.

Übersetzung der Transform-Funktion

Sei transfun der Name einer zweistelligen Transform-Funktion, angewendet auf den n-stelligen Konstruktor Kon mit $0 \leq n$:

- Definitionsgleichung einer Transform-Funktion

  t@(Kon t_1 .. t_n) in
  transfun t K = RS

- Dazugehörige Haskell-Übersetzung

  transfun ta@(t0@(($\langle VTK\ von\, \mathrm{t}\rangle$ t@(Kon t_1 .. t_n)): ts), z) (($\langle TTK\ von\, \mathrm{K}\rangle$ K) =
  (($\langle VTK\ von\, \mathrm{t}\rangle$ $\langle \ddot{U}bersetzung\ von\ RS\rangle$)

Wir beschreiben mit $\langle TTK\, von \mathrm{K}\rangle$ den Konstruktor aus dem Typ TransType, dessen Nachfolgertypsynonym mit dem Typsynonym von K übereinstimmt. Bei $\langle \ddot{U}bersetzung\, von\, \mathrm{RS}\rangle$ wird der Haskell$^+$-Code RS identisch übernommen, bis auf die eventuell enthaltenen Aufrufe von synthetischen und inheriten Funktionen, die nach der schon in Abschnitt 6.1.2 beschriebenen Methodik übersetzt werden.

Textuelle Transformationen

Wir wenden uns jetzt dem Typ TransType zu. Seine Aufgabe ist es, die Funktion `execTrans` noch für einen weiteren Verwendungszweck einsetzen zu können. Wir erinnern uns dabei an die Eingabesyntax der Sprache $\underline{L}$. Zur textuellen Eingabe bei der Verwendung des $\underline{SBE(L)}$ können ausschließlich die für Haskell$^+$ vordefinierten nichtkonstruktorbasierten Datentypen *IntInput*, *BoolInput*, *CharInput*, *StringInput*, *IdentInput* und *QuotedStringInput* verwendet werden. Diese kommen in der abstrakten Syntax der Sprache $\underline{L}$ nur als Nachfolger einstelliger Konstruktoren vor. Die Einbindung des vom $\underline{Nutzer_{SBE(L)}}$ innerhalb von $\underline{EK_L}$ eingegebenen Textes in $\underline{EK_L < AbstrakteSyntax_L >}$ kann folgendermaßen geschehen:

Sei VT der Typ vom Wurzelknoten des vom $\underline{Nutzer_{SBE(L)}}$ aktuell ausgewählten Teilbaumes von $\underline{EK_L < AbstrakteSyntax_L >}$. Besitzt der Typ VT einen einstelligen Konstruktor NK, dessen Nachfolger den nichtkonstruktorbasierten Typ TT besitzt, so soll die textuelle Eingabe einer Instanz des Typs TT möglich sein. Kann der vom $\underline{Nutzer_{SBE(L)}}$ eingegebene Text text als Typ TT gescannt werden, so soll er in $\underline{EK_L < AbstrakteSyntax_L >}$ eingefügt werden können. Dies geschieht durch Ersetzen des aktuell ausgewählten Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$ durch den Baum (NK text).

Um dieses Ersetzen zu realisieren, kann man eine Funktion `transText` generieren, welche analog zur übersetzten Transform-Funktion aufgebaut ist. Der Kontextparameter der Funktion `transText` enthält aber nicht die Bezeichnung einer strukturellen Transformation, sondern die Information über den vom $\underline{Nutzer_{SBE(L)}}$ eingegebenen Text, welche aus dem Eingabetext selbst und dem Typen besteht, für die der Eingabetext erfolgreich gescannt werden konnte. Durch die Verwendung des Datentyps TransType als Typ des Kontextparameters der übersetzten Transform-Funktion sowie der Funktion `transText` kann die Funktion `execTrans` zum Aufruf beider Funktionen verwendet werden, wenn $\underline{EK_L}$ strukturell bzw. textuell vom $\underline{Nutzer_{SBE(L)}}$ editiert werden soll.

Wir geben nun den zu generierenden Datentyp TransTyp an:

```
data TransType = TTTransCommand TransSelection
               | TTToIntInput IntInput
               | TTToBoolInput BoolInput
               | TTToCharInput CharInput
               | TTToStringInput StringInput
               | TTToIdentInput IdentInput
               | TTToQuotedStringInput QuotedStringInput
                 deriving (Show,Eq,Read)
```

Die Definition des Typs TransSelection ist in der $\underline{Spez_{SBE(L)}}$ in Haskell-Syntax enthalten und kann daher ohne Veränderung übernommen werden.

Kommen wir nun zur Generierung der Funktion `transText`. Wir überlegen uns noch einmal, wann es dem $\underline{Nutzer_{SBE(L)}}$ erlaubt sein soll, eine textuelle Eingabe zu tätigen.
Sei VT wieder der Typ vom Wurzelknoten des aktuell ausgewählten Teilbaumes von $\underline{EK_L < AbstrakteSyntax_L >}$. Nun soll für alle nichtkonstruktorbasierten Typen TT eine textuelle Eingabe möglich sein, die als Nachfolger eines einstelligen Konstruktors NK des Typs VT vorkommen. In dieser Situation soll also die Funktion `transText` auf den Wurzel-Konstruktor des aktuell ausgewählten Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$ anwendbar sein. Dabei sind aber nur Kontextparameter erlaubt, welche sich aus der Einbettung einer als Typ TT gescannten textuellen Eingabe in den Typ TransType ergeben. Die bezüglich des Kontextparameters genannte Einbettung der vom $\underline{Nutzer_{SBE(L)}}$ getätigten textuellen Eingabe in den Typ TransType geschieht durch Voranstellen des Konstruktors aus TransType, dessen Nachfolger durch das Typsynonym beschrieben wird, für das die textuelle Eingabe erfolgreich gescannt worden ist.
Wenn die Funktion `transText` über die Funktion `execTrans` aufgerufen wird, bekommt sie ihren Kontextparameter über die Funktion `execTrans`. Zu diesem Zeitpunkt ist schon bekannt, daß die diesem Kontextparameter zugrundeliegende textuelle Eingabe erfolgreich gescannt werden konnte. Die Funktion `transText` muß diese Eingabe also nur noch korrekt in $\underline{EK_L < AbstrakteSyntax_L >}$ einfügen.

Wir beschreiben nun formal die Generierung der Definitionsgleichungen der Funktion `transText` aus der $\underline{Spez_{SBE(L)}}$:

Sei DB der in $\underline{Spez_{SBE(L)}}$ spezifizierte *Data-Block*, in dem die abstrakte Syntax der Sprache $\underline{L}$ beschrieben wird.

Für jeden Typ VT, welcher in DB definiert wird, tue folgendes:
  Für alle einstelligen Konstruktoren NK mit dem Typ VT,
  deren Nachfolger einen nichtkonstruktorbasierten Typ TT besitzt, tue folgendes:
    Für alle n-stelligen Konstruktoren AK ($0 \leq n$), mit dem Typ VT
    generiere folgende Definitionsgleichung der Funktion `transText`:

$$\mathrm{transText\ ta@(t0@((}\langle \mathit{VTK\ von}\ \mathrm{t}\rangle\ \mathrm{t@(AK\ t_1\ ..\ t_n)):\ ts),\ z)\ (}\langle \mathit{TTK\ von}\ \mathrm{NK}\rangle\ \mathrm{te) =}$$
$$\mathrm{((}\langle \mathit{VTK\ von}\ \mathrm{(NK\ te)}\rangle\ \mathrm{(NK\ te))}$$

Wir benötigen nun noch eine Funktion zum Aufruf der Transform-Funktion bzw. der Funktion `transText`. Wir generieren uns daher die Funktion mit dem Namen `transFunCall` in der Weise, daß sie den Namen der übersetzten Transform-Funktion kennt und daher in der Lage ist, diese aufzurufen. Die Funktion `transFunCall` wird selbst von der Funktion `execTrans` aufgerufen und bekommt von ihr den Kontextparameter vom Typ TransType, welchen sie an die aufzurufende übersetzte Transform-Funktion bzw. an die aufzurufende Funktion `transText` weitergibt. Ob die Funktion `transText` oder die übersetzte Transform-Funktion aufzurufen ist, erkennt die Funktion an dem Wurzel-Konstruktor des weiterzureichenden Kontextparameters. Lautet dieser TTTransCommand, so ist die übersetzte Transform-Funktion aufzurufen, ansonsten die Funktion `transText`.

Bleibt die Frage, wann und von welcher Funktion die Funktion `execTrans` aufgerufen werden soll, während der $\underline{Nutzer_{SBE(L)}}$ mit dem $\underline{SBE(L)}$ das $\underline{EK_L}$ editiert. Die Funktion `execTrans` soll natürlich dann aufgerufen werden, wenn der $\underline{Nutzer_{SBE(L)}}$ den Editierbefehl dazu gibt. Wie dieser Befehl von der Nutzerschnittstelle zum Auswerter des $\underline{SBE(L)}$ gelangt, werden wir später genauer erklären.
Aufgerufen wird die Funktion `execTrans` von der schon im Abschnitt 5.1.3. kurz erwähnten Travers-Funktion. Wenn die Travers-Funktion bei ihrem rekursiven, zyklischen Durchlauf durch $\underline{EK_L < AbstrakteSyntax_L >}$ an der Wurzel des aktuell ausgewählten Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$ stoppt und danach ein Editierbefehl zur strukturellen bzw. zur textuellen Transformation erfolgt, so ruft die Travers-Funktion die Funktion `execTransCall` auf, welche ihrerseits die Funktion `execTrans` aktiviert. Die Funktion `execTransCall` ist für die notwendige Einbettung des Rekursionsargumentes der Funktion `execTrans` in den Vereinigungstyp, sowie für die entsprechende Ausbettung des Ergebnisses der Funktion `execTrans` erforderlich. Die Funktion `execTransCall` leitet auch die zwei Kontextparameter unverändert an die Funktion `execTrans` weiter, die sie von der Travers-Funktion erhält. Der erste Kontextparameter ist der Pfad zum Knoten in $\underline{EK_L < AbstrakteSyntax_L >}$ an dem die Travers-Funktion gerade angewendet wird (und der transformiert werden soll), der zweite ist vom Typ TransTyp und enthält die in diesem Abschnitt beschriebenen Informationen zur auszuführenden Transformation.

6.1.5 Steuerung des Auswerters

In diesem Unterabschnitt besprechen wir die sogenannte Travers-Funktion, welche die Auswerterkomponente des $\underline{SBE(L)}$ steuert.

Die Aufgabe der Travers-Funktion liegt im zyklischen, rekursiven Durchlaufen des Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$, der dabei stattfindenden Steuerung des Informationstransfers zwischen der Auswerterkomponente (zu der sie selbst gehört) und der Nutzerschnittstellenkomponente des $\underline{SBE(L)}$ sowie in der davon abhängigen Anwendung der Unparse-Funktionen und der Transform-Funktion.
Bei dem rekursiven Durchlauf von $\underline{EK_L < AbstrakteSyntax_L >}$ (top-downleft-to-right) findet während der Anwendung der Travers-Funktion an demjenigen Knoten des Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$ eine Kommunikation der beiden Komponenten des $\underline{SBE(L)}$ statt, der die Wurzel des aktuell ausgewählten Teilbaums des Syntaxbaums $\underline{EK_L < AbstrakteSyntax_L >}$ ist. Diese Kommunikation läuft folgendermaßen ab:

Zuerst veranlaßt die Travers-Funktion die Übertragung von Informationen über den aktuell ausgewählten Teilbaum t_a von $\underline{EK_L < AbstrakteSyntax_L >}$ an die Nutzerschnittstelle. Zu diesen Informationen gehören die Bezeichner der auf den Teilbaum t_a anwendbaren strukturellen Transformationen, der Typ des Wurzelknotens von t_a und die zur Darstellung von $\underline{EK_L < Ausgabesyntax_L >}$ (bei welcher der aktuell ausgewählte Teil von $\underline{EK_L}$ besonders hervorgehoben werden soll) notwendigen Daten.
Danach wartet die Travers-Funktion auf die Information von der Nutzerschnittstelle über einen vom $\underline{Nutzer_{SBE(L)}}$ getätigten Editierbefehl.
In Abhängigkeit der eintreffenden Information erfolgt dann entweder ein Aufruf der Travers-Funktion

- auf dem unveränderten $\underline{EK_L < AbstrakteSyntax_L >}$ mit einer neuen aktuellen Auswahl eines Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$

  (wenn der $\underline{Nutzer_{SBE(L)}}$ einen anderen Teil von $\underline{EK_L}$ auswählen will) oder

- auf dem unveränderten $\underline{EK_L < AbstrakteSyntax_L >}$ mit derselben aktuellen Auswahl eines Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$, wobei vorher $\underline{EK_L}$ in geeigneter Darstellung in einer Datei abgespeichert wurde

  (wenn der $\underline{Nutzer_{SBE(L)}}$ das Speichern von $\underline{EK_L}$ wünscht) oder

- auf einem neuen $\underline{EK_L < AbstrakteSyntax_L >}$

  (wenn der $\underline{Nutzer_{SBE(L)}}$ das Laden eines $\underline{EK_L}$ wünscht bzw. mit dem Editieren eines neuen $\underline{EK_L}$ beginnen möchte) oder

- auf dem transformierten $\underline{EK_L < AbstrakteSyntax_L >}$

  (falls der $\underline{Nutzer_{SBE(L)}}$ das $\underline{EK_L}$ strukturell oder textuell editieren möchte)

oder die Travers-Funktion wird beendet (wenn der $\underline{Nutzer_{SBE(L)}}$ das Editieren abschließen will), was zum Beenden der Auswerterkomponente des $\underline{SBE(L)}$ führt.

Wir beschreiben jetzt den Aufbau der Travers-Funktion informell:
Da die Travers-Funktion rekursiv auf $\underline{EK_L < AbstrakteSyntax_L >}$ angewendet werden soll, besitzt ihr Rekursionsargument den Vereinigungstyp für die Datentypen, welche die abstrakte Syntax der Sprache $\underline{L}$ beschreiben. Für jeden Konstruktor, der in dem *Data-Block* zur Beschreibung der abstrakten Syntax von $\underline{L}$ verwendet wurde, muß daher auch eine Definitionsgleichung der Travers-Funktion generiert werden. Die Travers-Funktion besitzt folgende Kontextparameter :

- Der erste Kontextparameter ist vom Typ des Wurzelknotens des Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$, also des Konstruktors, der als *Root-Symbol* in den *Mag-Blöcken* definiert worden ist, welche den *Data-Block* zur Beschreibung der abstrakten Syntax von $\underline{L}$ als *Input-Data-Block* verwenden. Er enthält stets das gesamte $\underline{EK_L < AbstrakteSyntax_L >}$.

- Der zweite Kontextparameter hat den Typ einer Integer-Liste und repräsentiert den aktuellen Pfad in $\underline{EK_L < AbstrakteSyntax_L >}$ (Pfad von der Wurzel zum Knoten in $\underline{EK_L < AbstrakteSyntax_L >}$ auf den die Travers-Funktion gerade angewendet wird).

- Der dritte Kontextparameter ist auch vom Typ einer Integer-Liste und repräsentiert den Pfad von der Wurzel zum Knoten in $\underline{EK_L < AbstrakteSyntax_L >}$, der die Wurzel des neu vom $\underline{Nutzer_{SBE(L)}}$ ausgewählten Teilbaums des Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$ ist.

- Der vierte Kontextparameter ist vom Typ Bool und trägt die Information, ob die Travers-Funktion zu dem Knoten in $\underline{EK_L < AbstrakteSyntax_L >}$ "durchlaufen" soll, der durch den dritten Kontextparameter beschrieben wird.

Die Auswerterkomponente des $\underline{SBE(L)}$ beginnt ihre Arbeit, indem sie die Travers-Funktion folgendermaßen aufruft:

travers ($\langle VTK\ von$ initialTree$\rangle$ initialTree) initialTree [ ] [ ] False

wobei initialTree der initiale Syntaxbaum $\underline{EK[Init]_L < AbstrakteSyntax_L >}$ ist und wir mit $\langle VTK\ von$ initialTree$\rangle$ den Konstruktor aus dem Vereinigungstyp bezeichnen, dessen Nachfolgertyp mit dem Typ von initialTree übereinstimmt.

Der Ergebnistyp der Travers-Funktion ist IO (Bool,Bool) (Es wird IO (Bool,Bool) statt (Bool,Bool) wegen der besseren Handhabbarkeit in der Implementierung verwendet). Dabei enthält das erste Element des Ergebnispaares die Information darüber, ob die Travers-Funktion ihren zyklischen, rekursiven Lauf durch $\underline{EK_L < AbstrakteSyntax_L >}$ weiterführen soll, damit der $\underline{Nutzer_{SBE(L)}}$ weiterhin $\underline{EK_L}$ editieren kann, oder ob ein Abbruch der Funktion gewünscht wird. Anhand des zweiten Elementes des Ergebnispaares kann entschieden werden, ob eine erneut aufgerufene Travers-Funktion zu dem Knoten im Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ "durchlaufen" soll, welcher durch den dritten Kontextparameter der Travers-Funktion spezifiziert wird, oder nicht.
Es sei noch bemerkt, daß die Travers-Funktion nur auf konstruktorbasierte Typen angewendet werden soll.

Wir beschreiben nun die zu generierenden Definitionsgleichungen der Travers-Funktion formell, wobei wir sie abschnittsweise erläutern:

Sei DB der in $\underline{Spez_{SBE(L)}}$ spezifizierte *Data-Block*, in dem die abstrakte Syntax der Sprache $\underline{L}$ beschrieben wird. Sei $n \geq 0$, dann wird für alle n-stelligen Konstruktoren Kon, die in *Data-Definitionen* von DB verwendet werden, folgende Definitionsgleichung der Funktion `travers` generiert:
Dabei gelte entweder

- $k = 0$, wenn $n = 1$ und der erste Nachfolger von Kon besitzt einen nichtkonstruktorbasierten Typ oder
- $k = n$, sonst.

Generierte Definitionsgleichung:

travers t0@(⟨*VTK von* t⟩ t@(Kon t_1 .. t_n)) tree sel jPoint jPred =
  do
      −− Abschnitt 1
      jPred ← (if (sel == jPoint) then (return False) else (return jPred))

      −− Abschnitt 2
      command ← (if jPred
                        then (return "walk")
                        else (do commandPrint ["⟨TK_1⟩", .. ,"⟨TK_r⟩"]
                                  typPrint "⟨*Typ von* Kon⟩"
                                  upFunCall ([⟨*VTK von* tree⟩ tree] , [ ]) sel
                                  getCommandFromFile)

      −− Abschnitt 3
      information ← n⟨k⟩_childs_travers_scheme
                          tree
                          sel
                          command
                          jPred
                          (textScanTries ["⟨IT_1⟩", .. ,"⟨IT_s⟩",""])
                          (travers (⟨*VTK von* t_1⟩) t_1) tree (1:sel) jPoint)
                            ..
                          (travers (⟨*VTK von* t_k⟩) t_k) tree (k:sel) jPoint)

      −− Ist Kon das *Root-Symbol*, so folgt Abschnitt 4a
      jPred ← return (jump_ahead information)
      (if (travers_ahead information)
       then (travers t0 tree sel jPoint jPred)
       else return (False,jPred))

      −− Ist Kon nicht das *Root-Symbol*, so folgt Abschnitt 4b
      return information

Dabei ist r die Anzahl der für (Kon t_1 .. t_n) spezifizierten Transform-Funktionsdefinitionen und wie bezeichnen mit $\langle TK_1 \rangle$ .. $\langle TK_r \rangle$ die Konstruktoren vom Typ TransSelection, welche jeweils in einer Transform-Funktionsdefinition für (Kon t_1 .. t_n) als erste Kontextparameter vorkommen, und zwar in der Reihenfolge des Auftretens der Transform-Funktionsdefinitionen.

Weiterhin ist s die Anzahl der nichtkonstruktorbasierten Typen, die in der Typdefinition, in welcher der Konstruktor Kon definiert wird, als Nachfolger eines einstelligen Konstruktors verwendet werden. Mit $\langle IT_1 \rangle$ .. $\langle IT_s \rangle$ bezeichnen wir diese nichtkonstruktorbasierten Typen in der Reihenfolge ihres Auftretens in der Typdefinition.

Erklärungen zu den einzelnen Abschnitten:

Abschnitt 1: Wird die Travers-Funktion auf den Knoten angewendet, zu dem "durchgelaufen" werden sollte, so belege die Variable `jPred` mit dem Wert False (was bedeutet, daß nicht mehr "durchgelaufen" werden soll), ansonsten übernehme den alten Wert der Variable.

Abschnitt 2: Hat die Variable `jPred` den Wert True (es soll "durchgelaufen" werden), so belege die Variable `command` mit dem Wert "walk".
Ist dies nicht der Fall, so wird (Kon t_1 .. t_n) zum aktuell ausgewählten Teilbaum von $\underline{EK_L < AbstrakteSyntax_L >}$. Deshalb übermittle der Nutzerschnittstelle durch die Erzeugung von Informationsdateien die Informationen

- über die auf den ausgewählten Teil von $\underline{EK_L}$ anwendbaren strukturellen Transformationen (Aufruf von `commandPrint`),
- über den Typ von (Kon t_1 .. t_n) (Aufruf von `typPrint`) sowie
- über die Darstellung von $\underline{EK_L < Ausgabesyntax_L >}$ (Aufruf von `upFunCall`).

Danach warte auf Information von der Nutzerschnittstelle über den neuen Editierbefehl des $\underline{Nutzer_{SBE(L)}}$ und belege die Variable `command` mit diesem Wert (Aufruf von `getCommandFromFile`).
Die Variable `command` besitzt danach in jedem Fall einen Wert vom Typ String, welcher die Art des vom $\underline{Nutzer_{SBE(L)}}$ getätigten Editierbefehls, angewendet auf $\underline{EK_L}$, kodiert.

Abschnitt 3: Starte einen Aufruf der $(5 + k)$-stelligen Travers-Scheme-Funktion (diese hat kein Rekursionsargument !) mit der Bezeichnung `n`$\langle k \rangle$`_childs_travers_scheme` und übergebe ihren Rückgabewert an die Variable `information`. Die Travers-Scheme-Funktion besitzt den gleichen Ergebnistyp wie die Travers-Funktion und die im Ergebnis enthaltenen Daten sind auch genauso zu interpretieren. Die Travers-Scheme-Funktion übernimmt die Aufgabe der Auswertung der Variablen `command` und verwaltet auch die rekursiven Aufrufe der Travers-Funktion. Die ersten vier Kontextparameter, welche die Travers-Scheme-Funktion von der Travers-Funktion erhält, sind schon erläutert worden.
Der fünfte Kontextparameter ist die partiell applizierte Funktion `textScanTries`. Diese Funktion erhält von der Travers-Funktion nur ihren ersten Kontextparameter, nämlich die Liste der Namen der nichtkonstruktorbasierten Typen, welche für eine textuelle Transformation des aktuell ausgewählten Teils von $\underline{EK_L}$ in Frage kommen (diese Liste schließt

immer mit dem leeren String ab). Die Aufgabe der Funktion `textScanTries` liegt im Bereich der Verarbeitung von textuellen Eingaben des $\underline{Nutzer_{SBE(L)}}$, ihre Funktionsweise wird in Abschnitt 6.2.1 erklärt. Die letzten k Kontextparameter bestehen aus den rekursiven Aufrufen der Travers-Funktion auf die Nachfolger des Konstruktors Kon, wobei die rekursiven Aufrufe nur partiell appliziert sind (letzter Kontextparameter fehlt).

Abschnitt 4a: Die Variable `jPred` bekommt ihren Wert durch das zweite Element des Paares `information` durch die Funktion `jump_ahead` zugewiesen. Der Wert des ersten Paarelementes der Variablen `information` wird durch die Funktion `travers_ahead` ausgelesen, wobei die Travers-Funktion erneut aufgerufen wird, falls der Wert True ist (der $\underline{Nutzer_{SBE(L)}}$ wünscht weiterhin zu editieren), ansonsten wird das Paar (False,jPred) zurückgegeben, was zum Beenden aller äußeren Travers-Funktionen führt.

Abschnitt 4b: Gib den Wert der Variablen `information` zurück.

Travers-Scheme-Funktionen:

Wir besprechen nun noch die Generierung der Travers-Scheme-Funktionen, welche die Aufgabe haben, der Travers-Funktion die Arbeit abzunehmen, die bei allen Definitionsgleichungen der Travers-Funktion in gleicher Weise ablaufen würde. Somit kann der Umfang einer Travers-Funktionsdefinition drastisch reduziert werden, was im Hinblick auf die Größe des erzeugten Haskell-Moduls **Evaluator** von Vorteil ist, da sehr viele Travers-Funktionsdefinitionsgleichungen (eine pro Konstruktor der abstrakten Syntax von $\underline{L}$) generiert werden müssen.
Wir generieren mehrere Travers-Scheme-Funktionen, die sich in ihrer Stelligkeit unterscheiden, damit in jeder Travers-Funktionsdefinitionsgleichung eine (im Hinblick auf die Anzahl der zu verwaltenden rekursiven Aufrufe der Travers-Funktion) passende Travers-Scheme-Funktion aufgerufen werden kann.

Generierung der Travers-Scheme-Funktionen

Sei DB der in $\underline{Spez_{SBE(L)}}$ spezifizierte *Data-Block*, in dem die abstrakte Syntax der Sprache $\underline{L}$ beschrieben wird. Sei m die maximale Nachfolgeranzahl der Konstruktoren, die in DB auftreten. Für alle p, mit $0 \leq p \leq m$ wird die folgende Travers-Scheme-Funktion deklariert und definiert:

Deklaration:

np_childs_travers_scheme ::
$\langle Roottyp \rangle \rightarrow$ [Int] $\rightarrow$ String $\rightarrow$ Bool $\rightarrow$
$(\langle Roottyp \rangle \rightarrow$ [Int] $\rightarrow$ String $\rightarrow \langle Roottyp \rangle) \rightarrow$
(Bool $\rightarrow$ IO (Bool,Bool)) $\rightarrow$ } p - mal
IO (Bool,Bool)

<u>Definition:</u>

np_childs_travers_scheme
    oTree sel command jPred tryToScanAndTrans travers1 .. travers$p$ =

```
  case command of

    "quit"   → (return (False,jPred))

    "walk"   → ⟨Rekursionsverwaltung⟩

    "jump"   → (do jPoint ← getSelectionFromFile
                   (travers (⟨VTK von oTree⟩ oTree) oTree [] jPoint True))

    "save"   → (do file ← getFileFromFile
                   saveTreeString (show oTree) file
                   (travers (⟨VTK von oTree⟩ oTree) oTree [] sel True))

    "load"   → (do file ← getFileFromFile
                   treeString ← loadTreeString file
                   nTree ← getTreeFromString treeString
                   (travers (⟨VTK von nTree⟩ nTree) nTree [] [] False))

    "new"    → (let nTree = initialTree
                   in (travers (⟨VTK von nTree⟩ nTree) nTree [] [] False))

    "text"   → (do text ← getTextFromFile
                   (let nTree = (tryToScanAndTrans oTree sel text)
                        in (travers (⟨VTK von nTree⟩ nTree) nTree [] sel True)))

    _        → (let nTree = (execTransCall
                                oTree
                                (reverse sel)
                                (trans2TransSelection command))
                   in (travers (⟨VTK von nTree⟩ nTree) nTree [] sel True))
```

⟨*Rekursionsverwaltung*⟩ erklärt sich durch die folgenden Fälle:

Fall 1: $p = 0$

```
(return (True,jPred))
```

Fall 2: $p = 1$

```
(travers1 jPred)
```

Fall 3: $2 \leq p \leq m$

```
(do information ← (travers1 jPred)
    jPred ← return (jump_ahead information)
    (if (travers_ahead information)
     then
           ..
           (do information ← (traversj jPred)
               jPred ← return (jump_ahead information)
               (if (travers_ahead information)
                then
                      ..
                      (traversp jPred)
                            ..
                else (return (False,jPred))))
               ..
    else (return (False,jPred))))
```

wobei $2 \leq j \leq p$ gilt.

Wir erläutern nun noch kurz die einzelnen Fälle in dem case-Ausdruck, der den Körper einer Travers-Scheme-Funktion bildet:

`command` hat den Wert:

- "quit" - Tritt auf, wenn der $\underline{Nutzer_{SBE(L)}}$ das Editieren von $\underline{EK_L}$ beenden will.

  Es erfolgt kein Aufruf einer Travers-Funktion und der Rückgabewert (False,jPred) bewirkt das Beenden aller äußeren Travers-Funktionen und damit auch das Beenden der Auswerterkomponente von $\underline{SBE(L)}$.

- "walk" - Tritt auf, wenn der $\underline{Nutzer_{SBE(L)}}$ den Editierbefehl Travers auslöst, weil er möchte, daß die aktuelle Auswahl eines Teiles von $\underline{EK_L}$ einen Schritt weiterläuft, in Hinsicht auf ihren Durchlauf (top-down left-to-right) durch den Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$.
  Außerdem tritt dieser Fall ein, wenn ein automatischer "Durchlauf" zu einem bestimmten Knoten in $\underline{EK_L < AbstrakteSyntax_L >}$ noch nicht abgeschlossen ist.

  Es erfolgt die Steuerung der rekursiven Aufrufe der Travers-Funktion, wobei der Wert der Variable `jPred` (der sich durch Anwendung der rekursiven Aufrufe der Travers-Funktion ändern kann) die nur teilweise applizierten Aufrufe der Travers-Funktion, die hier als Kontextparameter vorliegen, vervollständigt. Liegt mehr als ein rekursiver Aufruf der Travers-Funktion vor, so wird immer erst anhand der schon durchgeführten entschieden, ob ein Aufruf zur Anwendung kommen oder ob das Paar

(False,jPred) zurückgegeben werden soll (was zum Abbruch aller äußeren Travers-Funktionen führt).

- "jump" - Tritt auf, wenn der $\underline{Nutzer_{SBE(L)}}$ einen neuen Teil von $\underline{EK_L}$ auswählt.

  Zuerst wird mit der Funktion `getSelectionFromFile` der Pfad zur Wurzel des ausgewählten Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$ aus einer von der Nutzerschnittstelle des $\underline{SBE(L)}$ angelegten Datei gelesen. Danach erfolgt ein Aufruf der Travers-Funktion an der Wurzel von $\underline{EK_L < AbstrakteSyntax_L >}$ mit dem ermittelten Pfad als dritten Kontextparameter und dem Wert True als vierten Kontextparameter.

- "save" - Tritt auf, wenn der $\underline{Nutzer_{SBE(L)}}$ $\underline{EK_L}$ abspeichern will.

  Es wird der Name der abzuspeichernden Datei durch die Funktion `getFileFromFile` gelesen und danach $\underline{EK_L < AbstrakteSyntax_L >}$ als String in dieser Datei gespeichert. Die Travers-Funktion wird anschließend an der Wurzel des Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$ mit dem aktuellen Pfad als dritten Kontextparameter und dem Wert True als vierten Kontextparameter aufgerufen, damit der aktuell ausgewählte Teil von $\underline{EK_L}$ erhalten bleibt.

- "load" - Tritt auf, wenn der $\underline{Nutzer_{SBE(L)}}$ ein vorher gespeichertes $\underline{EK_L}$ laden möchte.

  Zuerst wird der Name der gewünschten Datei mittels der Funktion `getFileFromFile` gelesen, wonach der in dieser Datei enthaltene String mit der Funktion `loadTreeString` ausgelesen und mit der Funktion `getTreeFromString` in den neuen Syntaxbaum $\underline{EK_L < AbstrakteSyntax_L >}$ umgewandelt wird. Nun erfolgt ein Aufruf der Travers-Funktion an der Wurzel des neuen $\underline{EK_L < AbstrakteSyntax_L >}$ und dem Wert False als vierten Kontextparameter.

- "new" - Tritt auf, wenn der $\underline{Nutzer_{SBE(L)}}$ mit dem Editieren eines neuen $\underline{EK_L}$ beginnen möchte.

  Wir erhalten hier durch den Wert der Konstante initialTree den initialen Syntaxbaum $\underline{EK[Init]_L < AbstrakteSyntax_L >}$. Danach erfolgt ein Aufruf der Travers-Funktion an der Wurzel von $\underline{EK[Init]_L < AbstrakteSyntax_L >}$ und dem Wert False als vierten Kontextparameter.

- "text" - Tritt auf, wenn der $\underline{Nutzer_{SBE(L)}}$ eine textuelle Eingabe abgeschlossen hat.

  Zunächst wird der vom $\underline{Nutzer_{SBE(L)}}$ eingegebene Text mit Hilfe der Funktion `getTextFromFile` aus einer von der Nutzerschnittstelle des $\underline{SBE(L)}$ angelegten Datei gelesen. Danach erfolgt mit diesem Text die textuelle Transformation des Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$ durch die als Kontextparameter `tryToScanAndTrans` vorliegende Funktion `textScanTries` (Diese Funktion wird ebenfalls mitsamt benötigter Hilfsfunktionen generiert, ihre Funktionsweise besprechen wir in Abschnitt 6.2.2). Im Anschluß daran wird die Travers-Funktion an der Wurzel des transformierten $\underline{EK_L < AbstrakteSyntax_L >}$ aufgerufen, und zwar mit dem aktuellen Pfad als dritten Kontextparameter und dem Wert True als vierten Kontextparameter.

- default - Tritt auf, wenn der $\underline{Nutzer_{SBE(L)}}$ eine strukturelle Transformation fordert.

  Der Wert der Variablen `command` kann nun nur noch den Namen einer strukturellen Transformation als Wert besitzen. Mit Hilfe der Funktion `execTransCall` wird die strukturelle Transformation von $\underline{EK_L < AbstrakteSyntax_L >}$ ausgeführt. Sie bekommt als Parameter den aktuellen Pfad (Parameter `sel` invertiert) und den in den Typ TransTyp umgewandelten Wert von `command`. Danach wird die Travers-Funktion an der Wurzel des transformierten $\underline{EK_L < AbstrakteSyntax_L >}$ aufgerufen, wobei der dritte Kontextparameter den aktuellen Pfad als Wert erhält, und der Wert True dem vierten Kontextparameter zugewiesen wird.

<u>Beispiel zur Anwendung der Travers-Funktion:</u>

Zum Abschluß dieses Teilabschnittes beschreiben wir zum besseren Verständnis der Funktionsweise der Travers-Funktion noch ein kleines Beispiel. Wir erinnern noch einmal daran, daß durch die Travers-Funktion ein Durchlauf (top-down left-to-right) durch den abstrakten Syntaxbaum des vom $\underline{Nutzer_{SBE(L)}}$ editierten Konstruktes $\underline{EK_L}$ realisiert wird, um ihm eine aktuelle Auswahl im editierten Konstrukt zu erlauben und die Unparse-, sowie die Transform-Funktionen (strukturell oder textuell) aufzurufen, falls der $\underline{Nutzer_{SBE(L)}}$ einen entsprechenden Editierbefehl auslöst. Wir beschränken uns bei folgendem Beispiel auf die Beschreibung der Werte der Kontextparameter der Travers-Funktion sowie die in dem Körper der Travers-Funktion zu belegenden Variablenwerte. Besondere Aufmerksamkeit sollte der Leser auf den Kontextparameter `jPred` legen, dessen Wert im Funktionskörper geändert werden kann, wobei der eventuell geänderte Wert in den folgenden rekursiven Aufrufen der Travers-Funktion wieder als Kontextparameter verwendet wird. Dieser Kontextparamter enthält die Information darüber, ob im Körper der Travers-Funktion eine Kommunikation zwischen Auswerter- und der Nuzterschnittstellenkomponente des sprachbasierten Editors erfolgen, oder ob der Durchlauf (top-down left-to-right) sofort ohne diesen weitergeführt werden soll.

Sei der Syntaxbaum t, welcher in Abbildung 6.1 betrachtet werden kann, der Syntaxbaum $\underline{EK_P < AbstrakteSyntax_P >}$, der die interne Darstellung des $\underline{EK_P}$ ist, welches der $\underline{Nutzer_{SBE()}}$ während einer Editiersitzung mit dem $\underline{SBE(P)}$ (dessen Spezifikation $\underline{Spez_{SBE(P)}}$ wir in Kapitel 4 beschrieben haben) gerade bearbeitet. Zu diesem Zeitpunkt gilt das gesammte $\underline{EK_P}$ als aktuell ausgewählt. Wir beschreiben im folgenden drei vom $\underline{Nutzer_{SBE(P)}}$ getätigte Editierbefehle und die dadurch ausgelösten Aufrufe der Travers-Funktion:

Zuerst setzt der $\underline{Nutzer_{SBE(P)}}$ den Cursor im Editierfenster so, daß er den Teilbaum von $\underline{EK_P < AbstrakteSyntax_P >}$ mit dem Pfad 2.1.1 aktuell auswählt. Das heißt er wählt den Platzhalter in der Deklarationsschablone des $\underline{P}$-Programms $\underline{EK_P}$, welcher für den Namen der dort zu deklarierenden Variable steht. Dadurch erfolgt ein Aufruf der Travers-Funktion mit folgender Gestalt:

travers ($\langle VTK\ von$ t$\rangle$ t) t [ ] [1,1,2] True

Der durch das Setzen des Cursors mit der Maus ausgelöste Editierbefehl führt zur ak-

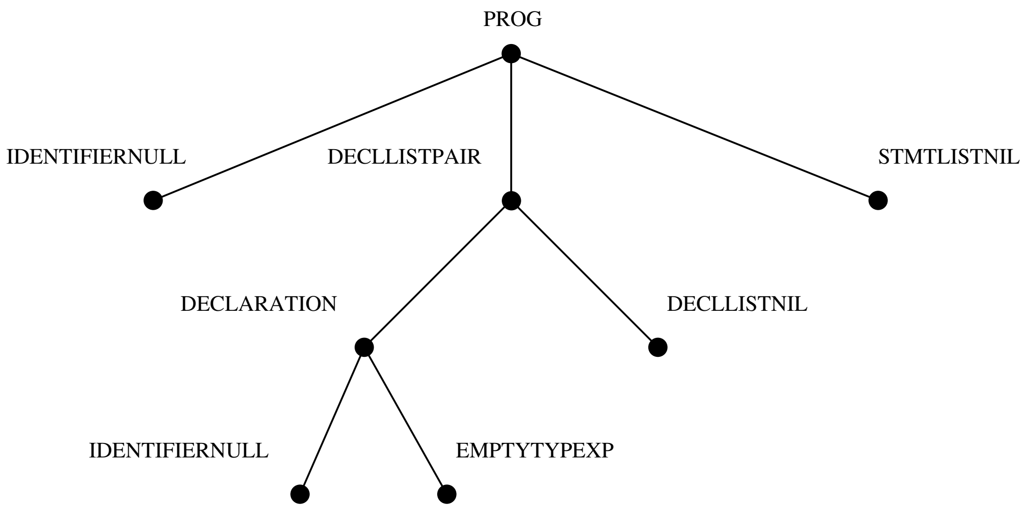

Abbildung 6.1: Syntaxbaum t unserer Beispielsprache $\underline{P}$

tuellen Auswahl des Teilbaums von $\underline{EK_P < AbstrakteSyntax_P >}$ mit dem Pfad 2.1.1. Dies geschieht durch eine Folge von Anwendungen der Travers-Funktion. In der folgenden Tabelle beschreiben dies die ersten fünf Zeilen. Die erste Spalte der Tabelle zeigt an, auf welchem Teilbaum von $\underline{EK_P < AbstrakteSyntax_P >}$ die Travers-Funktion aufgerufen wird (Beschreibung des Pfades von der Wurzel von $\underline{EK_P < AbstrakteSyntax_P >}$ zur Wurzel dieses Teilbaums).
Die nächsten vier Spalten geben die Werte der Kontextparameter an, mit denen die Travers-Funktion aufgerufen wird. Dabei ist zu beachten, daß die Integer-Listen zur Beschreibung von Pfaden in $\underline{EK_P < AbstrakteSyntax_P >}$ rückwärts kodiert sind (der Effizienz wegen).
Die fünfte und die sechste Spalte der Tabelle beschreiben die Werte, mit denen die im Körper der Travers-Funktion verwendeten Variablen `jPred` und `command` (in Abschnitt 1 bzw. 2 der formalen Beschreibung der Travers-Funktionsdefinition) belegt werden.
Die letzte Spalte zeigt an, ob es beim Aufruf der Travers-Funktion zu einer Kommunikation zwischen der Auswerterkomponente (AW) und der Nutzerschnittstellenkomponente (NS) des $\underline{SBE(P)}$ kommt oder nicht. Findet eine solche Kommunikation nicht statt, weil der Wert der Variablen `jPred` mit True belegt wurde, so wird die Variable `command` stets mit dem Wert "walk" besetzt, ansonsten ist diese Wertebelegung abhängig von dem Editierbefehl des $\underline{Nutzer_{SBE(P)}}$ (siehe Abschnitt 2 der formalen Beschreibung der Travers-Funktionsdefinition). Die in der Tabelle enthaltenen Pfeile sollen die Herkunft des Wertes der Variablen `command` verdeutlichen.

Aufruf bei Pfad	tree	Kontextparameter sel	jPoint	jPred	Variablen jPred	command	(AW $\longleftrightarrow$ NS)
ε	t	[ ]	[1,1,2]	True	True $\rightarrow$	“walk”	nein
1	t	[1]	[1,1,2]	True	True $\rightarrow$	“walk”	nein
2	t	[2]	[1,1,2]	True	True $\rightarrow$	“walk”	nein
2.1	t	[1,2]	[1,1,2]	True	True $\rightarrow$	“walk”	nein
2.1.1	t	[1,1,2]	[1,1,2]	True	False	“walk”	$\leftarrow$ ja
2.1.2	t	[2,1,2]	[1,1,2]	False	False	“InsertChar”	$\leftarrow$ ja
ε	t’	[ ]	[2,1,2]	True	True $\rightarrow$	“walk”	nein
1	t’	[1]	[2,1,2]	True	True $\rightarrow$	“walk”	nein
2	t’	[2]	[2,1,2]	True	True $\rightarrow$	“walk”	nein
2.1	t’	[1,2]	[2,1,2]	True	True $\rightarrow$	“walk”	nein
2.1.1	t’	[1,1,2]	[2,1,2]	True	True $\rightarrow$	“walk”	nein
2.1.2	t’	[2,1,2]	[2,1,2]	True	False	?	$\leftarrow$ ja

Nachdem die ersten fünf Zeilen der Tabelle die Umsetzung der aktuellen Auswahl des Teilbaums von $\underline{EK_P < AbstrakteSyntax_P >}$ mit dem Pfad 2.1.1 durch die Aufrufe der Travers-Funktion erläutern, beschreibt die sechste Zeile den Travers-Funktionsaufruf, welcher durch die Betätigung des Travers-Kommandoknopfes durch den $\overline{Nutzer_{SBE(P)}}$ in dieser Situation ausgelöst wird. Dieser zweite Editierbefehl des $\overline{Nutzer_{SBE(P)}}$ ist auch der Grund des Wertes “walk” von `command` in der fünften Zeile der Tabelle. Dies führt zur aktuellen Auswahl des Teilbaums von $\underline{EK_P < AbstrakteSyntax_P >}$ mit dem Pfad 2.1.2 (top-down left-to-right Durchlauf durch $\underline{EK_P < AbstrakteSyntax_P >}$), also dem Platzhalter in der Deklarationsschablone des $\underline{P}$-Programms $\underline{EK_P}$, der für den Typ der dort zu deklarierenden Variable steht.
Der letzte hier zu beschreibende Editierbefehl des $\overline{Nutzer_{SBE(P)}}$ sei die Auslösung der strukturellen Transformation (durch Betätigen des Kommandoknopfes mit der Aufschrift “InsertChar”) des aktuell ausgewählten Teilbaums (Blatt) `EMPTYTYPEXP` in den Teilbaum (Blatt) `CHARTYPEXP`. Daher ist auch der Wert der Variablen `command` in der sechsten Zeile der Tabelle mit dem Wert “InsertChar” belegt. Alle folgenden Zeilen beschreiben die resultierende Folge von Aufrufen der Travers-Funktion, welche mit dem Travers-Funktionsaufruf der folgenden Gestalt beginnt:

travers ($\langle VTK\ von$ t’$\rangle$ t’) t’ [ ] [2,1,2] True

Dabei sei t’ der Syntaxbaum, der durch die genannte Transformation von t entsteht. Er kann in Abbildung 6.2 betrachtet werden. Der Wert des letzten Kontextparameters (`jPred`) beträgt wieder True, da die Travers-Funktion an der Wurzel des transformierten $\underline{EK_P < AbstrakteSyntax_P >}$ aufgerufen wird, sich aber die aktuelle Auswahl nicht ändern soll.
Der Teilbaum t’ ist daher die aktuelle Auswahl im neuen $\underline{EK_P}$, d.h. der $\overline{Nutzer_{SBE(P)}}$ kann nun einen Editierbefehl in bezug auf diese Auswahl tätigen. Wir beenden das Beispiel

aber hier, daher ist der neue Befehl unbekannt, was durch das Fragezeichen in der letzten Tabellenzeile verdeutlicht wird.

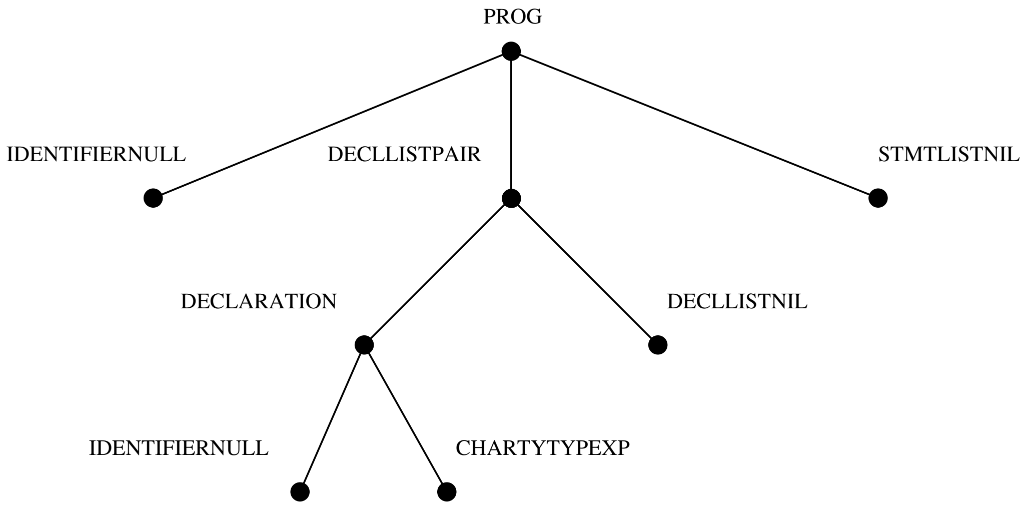

Abbildung 6.2: Syntaxbaum t' unserer Beispielsprache $\underline{P}$

6.1.6 Haskell-Code in $\underline{Spez_{SBE(L)}}$

Der in $\underline{Spez_{SBE(L)} < Haskell^+ >}$ enthaltene Haskell-Code, der vom $\underline{Entwickler_{SBE(L)}}$ programmiert wurde, um Hilfsfunktionen für die innerhalb von *Mag-Blöcken* definierten Funktionen (synthetische, inherite etc.) zu erstellen, wird unverändert in das zu erzeugende Haskell-Modul **Evaluator** übernommen.

6.1.7 Generierung von $\underline{EK[Init]_L < AbstrakteSyntax_L >}$

Es wird aus dem *Data-Block* zur Beschreibung der abstrakten Syntax der Sprache $\underline{L}$ innerhalb der $\underline{Spez_{SBE(L)}}$ eine nullstellige Funktion (Konstante) initialTree generiert, die den Syntaxbaum $\underline{EK[Init]_L < AbstrakteSyntax_L >}$ bereitstellt. Ihr Rückgabewert besitzt daher den Typ des *Root-Symbols*, welches in den *Mag-Blöcken* spezifiziert wurde, die diesen *Data-Block* in ihrer *Input-Definition* verwenden. $\underline{EK[Init]_L < AbstrakteSyntax_L >}$ wird durch die Verwendung der in dem *Data-Block* spezifizierten Default-Konstruktoren und des *Root-Symbols* erzeugt. Die Generierung des Ergebniswertes der Konstante initialTree läuft in folgender Form in Phase 3 der Generierung des $\underline{Spez_{SBE(L)}}$ ab:
Die Wurzel von $\underline{EK[Init]_L < AbstrakteSyntax_L >}$ ist der Konstruktor, der das *Root-Symbol* spezifiziert. Danach werden schrittweise die zur Konstruktion des Syntaxbaumes $\underline{EK[Init]_L < AbstrakteSyntax_L >}$ notwendigen Nachfolgerkonstruktoren eingefügt, welche sich als Default-Konstruktoren der entsprechenden Nachfolgertypen ermitteln lassen. Die kontextsensitiven Nebenbedingungen der Sprache Haskell$^+$, die dem $\underline{Entwickler_{SBE(L)}}$ bei der Spezifizierung eines *Data-Blocks* auferlegt sind, garantieren die Erzeugung eines

endlichen Syntaxbaumes $\underline{EK[Init]_L < AbstrakteSyntax_L >}$, in dem keine nichtkonstruktorbasierten Typinstanzen vorkommen.

6.1.8 Generierung der Auswerter-Funktion

Die Funktion, welche die Auswerterkomponente des erzeugten $\underline{SBE(L)}$ implementiert, wird ebenfalls in der Phase 3 generiert. Sie trägt den Namen `evaluator` und ruft die Travers-Funktion an der Wurzel des Syntaxbaumes $\underline{EK[Init]_L < AbstrakteSyntax_L >}$ auf. Sie besitzt den gleichen Ergebnistyp wie die Travers-Funktion, wobei aber das Ergebnis der Funktion `evaluator` nicht mehr ausgewertet wird, da mit Beenden dieser Funktion auch die Auswerterkomponente des $\underline{SBE(L)}$ ihr Ende findet.

evaluator :: IO (Bool,Bool)
evaluator = travers ($\langle VTK\ von$ initialTree$\rangle$ initialTree) initialTree [ ] [ ] False

Das *Root-Symbol* ist dabei der Konstruktor, welcher die Wurzel des initialen Syntaxbaumes $\underline{EK_L < AbstrakteSyntax_L >}$ (initialTree) beschriftet.

6.2 Quell-Code für Phase 5 - $\underline{UC_{AW} < Haskell >}$

Wir besprechen hier die Haskell-Module, die zusammen mit dem Haskell-Modul **Evaluator** die Auswerterkomponente des $\underline{SBE(L)}$ implementieren. Die folgenden Module sind im Rahmen dieser Arbeit, also vom $\underline{Entwickler_{Gen_{SBE}}}$ erstellt worden:

- **Main** - Dieses Modul enthält die main-Funktion und startet den Auswerter durch Aufruf der aus dem Modul **Evaluator** importierten Funktion `evaluator`.
- **HelpFuns** - Hier sind universelle Hilfsfunktionen enthalten, welche in das Modul **Evaluator** exportiert werden.
- **InterChange** - In diesem Modul sind die Funktionen zur Kommunikation mit der Nutzerschnittstellenkomponente des $\underline{SBE(L)}$ implementiert. Sie werden in das Modul **Evaluator** exportiert.
- **UserInput**, **UserIntInputScanner**, **UserBoolInputScanner**, **UserCharInputScanner**, **UserStringInputScanner**, **UserIdentInputScanner** und **UserQuotedStringInputScanner** -
Diese Module implementieren Funktionen zur Verarbeitung einer textuellen Eingabe des $\underline{Nutzer_{SBE(L)}}$ bei seiner Arbeit mit dem $\underline{SBE(L)}$. Diese Funktionen werden ebenfalls vom Modul **Evaluator** importiert.

Auf die ersten zwei Haskell-Module soll hier nicht weiter eingegangen werden, sie befinden sich auf der beigefügten Diskette und können nachgelesen werden. Wir widmen unsere Aufmerksamkeit zum einen dem Modul **InterChange** mit den Funktionen zur Kommunikation und zum anderen den restlichen Modulen, welche die Grundlage der für

Haskell^{+} vordefinierten Typen zur Beschreibung der Eingabesyntax der Sprache $\underline{L}$ durch den $\underline{Entwickler_{SBE(L)}}$ bilden.

6.2.1 Kommunikation zwischen Auswerter- und Nutzerschnittstellenkomponente des $\underline{SBE(L)}$

Wir gehen hier noch einmal kurz auf die Kommunikation der beiden Komponenten des $\underline{SBE(L)}$ (Auswerter und Nutzerschnittstelle) ein, welche in unterschiedlichen Sprachen (Haskell und Java) implementiert sind, ein (siehe auch Abschnitt 4.5.3). Die Kommunikation erfolgt über das Dateisystem mit dem folgenden Prinzip:
Jede der beiden Komponenten des $\underline{SBE(L)}$ schreibt Informationen in dafür vorgesehene Dateien, während die jeweils andere Komponente das Dateisystem auf die Aktualität dieser Dateien überprüft und in Abhängigkeit davon die darin enthaltenen Informationen ausliest. Wir interessieren uns auch bei Kommunikation besonders für deren Realisierung bei der Auswerterkomponente des $\underline{SBE(L)}$. Betrachtet man die Kommunikation von Seiten der Auswerterkomponente, so erfolgt sie im Rahmen eines Aufrufes der Travers-Funktion und dem damit verbundenen Aufruf einer Travers-Scheme-Funktion; dies wurde schon im Abschnitt 6.1.5 besprochen. Der Leser sei bei auftretenden Unklarheiten zum Vergleich mit diesem Abschnitt 6.1.5 aufgefordert.
Wir bieten an dieser Stelle eine Übersicht über die Namen der Kommunikationsdateien und deren Verwendungszweck sowie den Haskell-Funktionen, welche zum Schreiben bzw. zum Lesen dieser Dateien verwendet werden. Die Funktionen zur Kommunikation auf Seiten der Nutzerschnittstellenkomponente seien hier nicht näher erläutert; der interessierte Leser sei auf die entsprechenden Programm-Codes auf der der Arbeit beigefügten Diskette verwiesen.

Auswerterkomponente	Nutzerschnittstellenkomponente
$\longrightarrow$	$\longleftarrow$
transform.icf	command.icf
typ.icf	selection.icf
unparsing.icf	text.icf
	file.icf

- transform.icf   Enthält die Namen der aktuell möglichen strukturellen Transformationen. Diese sind zeilenweise in der Datei abgelegt und werden von der Nutzerschnittstelle als Aufschriften der Kommandoknöpfe auf der Oberfläche des Editors $\underline{SBE(L)}$ verwendet.
  Die Haskell-Funktion zum Schreiben dieser Datei:
  ```
  commandPrint :: [String] -> IO ()
  ```
- typ.icf   Beinhaltet den Typ des als aktuell ausgewählt geltenden Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$. Diese Information wird in der Anzeige `Typ` auf der Oberfläche des Editors $\underline{SBE(L)}$ ausgegeben.
  Die Haskell-Funktion zum Schreiben dieser Datei:
  ```
  typPrint :: String -> IO ()
  ```

- unparsing.icf Trägt alle Informationen zur Darstellung und Behandlung des externen Formates $\underline{EK_L < Ausgabesyntax_L >}$. Der Aufbau und die Verwendung des Inhalts wurde schon in Abschnitt 6.1.3 erläutert.
  Die Haskell-Funktion zum Schreiben dieser Datei:
  Eine der übersetzten Unparse-Funktionen aus der $\underline{Spez_{SBE(L)}}$.
  Aufgerufen wird diese Datei von der Haskell-Funktion `upFunCall`
  (siehe auch dazu Abschnitt 6.1.3).

- command.icf Enthält die Kodierung eines Editorbefehls (einzeiliger String), den der $\underline{Nutzer_{SBE(L)}}$ an die Nutzerschnittstelle gerichtet hat. Der Auswerter wird durch sie über die nächste notwendige Auswertung von $\underline{EK_L < AbstrakteSyntax_L >}$ unterichtet.
  Die Haskell-Funktion zum Lesen dieser Datei:
  `getCommandFromFile :: IO String`

Die bis hierhin erläuterten Funktionen werden durch die Travers-Funktion (Abschnitt 2 der formalen Beschreibung der Travers-Funktionsdefinition in Abschnitt 5.1.5 dieser Arbeit) aufgerufen.
Die noch folgenden Funktionen werden innerhalb der Travers-Scheme-Funktionen (siehe formale Beschreibung der Travers-Scheme-Funktionsdefinitionen in Abschnitt 6.1.5 dieser Arbeit) verwendet.

- selection.icf Trägt die Information über den Pfad des vom $\underline{Nutzer_{SBE(L)}}$ neu ausgewählten Teilbaums in $\underline{EK_L < AbstrakteSyntax_L >}$. Diese Datei birgt so die notwendige Zusatzinformation zu der Befehlkodierung "jump".
  Die Haskell-Funktion zum Lesen dieser Datei:
  `getSelectionFromFile :: IO [Int]`

- text.icf Enthält die vom $\underline{Nutzer_{SBE(L)}}$ abgeschlossene textuelle Eingabe. Sie beinhaltet die notwendige Zusatzinformation zu der Befehlkodierung "text".
  Die Haskell-Funktion zum Lesen dieser Datei:
  `getTextFromFile :: IO String`

- file.icf Beinhaltet den Namen der Datei aus der bzw. in die das vom $\underline{Nutzer_{SBE(L)}}$ aktuell bearbeitete $\underline{EK_L < AbstrakteSyntax_L >}$ zu laden bzw. zu speichern ist. Diese Datei enthält die notwendige Zusatzinformation zu den Befehlkodierungen "load" und "save".
  Die Haskell-Funktion zum Lesen dieser Datei:
  `getFileFromFile :: IO String`

Bis auf die aus der $\underline{Spez_{SBE(L)}}$ übersetzten Unparse-Funktionen und der in diesem Zusammenhang generierten Haskell-Funktion `upFunCall` sind alle hier genannten Haskell-Funktionen zum Lesen bzw. Schreiben von Dateien völlig unabhängig von der $\underline{Spez_{SBE(L)}}$ und damit in $\underline{UC_{AW} < Haskell >}$ enthalten, und zwar in dem Haskell Modul **InterChange**. Sie können daher im Anhang dieser Arbeit nachgelesen werden.

6.2.2 Für Haskell$^+$ vordefinierte Typen

In diesem Unterabschnitt besprechen wir die für Haskell$^+$ vordefinierten Typen, welche in der $\underline{Spez_{SBE(L)}}$ dazu verwendet werden können, in ersten Ansätzen eine Eingabesyntax der Sprache $\underline{L}$ zu spezifizieren. Da in der Sprache Haskell$^+$ die Möglichkeit fehlt, reguläre Ausdrücke beschreiben zu können, um damit auch Sprachkonstrukte der Sprache $\underline{L}$ zu definieren, die sich intern nicht durch Konstruktoren beschreiben lassen (wie z.B. Zahlen, Bezeichner etc.), werden dem $\underline{Entwickler_{SBE(L)}}$ die sechs vordefinierten, nichtkonstruktorbasierten Typen *IntInput*, *CharInput*, *BoolInput*, *StringInput*, *IdentInput* und *QuotedStringInput* zur Verfügung gestellt. Diese nichtkonstruktorbasierten Typen können dann von ihm in der Beschreibung der abstrakten Syntax der Sprache $\underline{L}$ (also innerhalb eines *Data-Blocks*) verwendet werden, wodurch der $\underline{Nutzer_{SBE(L)}}$ beim Editieren von $\underline{EK_L}$ die Möglichkeit bekommt, einfache Sprachelemente (wie z.B. Zahlen, Buchstaben, Bezeichner etc.) textuell manipulieren zu können.
Die textuelle Eingabemöglichkeit von konstruktorbasierten Typen wurde innerhalb dieser Arbeit nicht behandelt, da dies ein Ziel ist, welches erst in weiteren Entwicklungsschritten erreicht werden soll.

Die genannten nichtkonstruktorbasierten Typen sind vordefiniert, daher gehören die Beschreibungen der ihnen zugrundeliegenden regulären Ausdrücke nicht zum Inhalt von $\underline{Spez_{SBE(L)}}$, sondern sie sind in $\underline{UC_{AW} < Haskell >}$ enthalten und müssen damit vom $\underline{Entwickler_{Gen_{SBE}}}$ spezifiziert werden.

Die Festlegung der regulären Ausdrücke für die nichtkonstruktorbasierten Typen erfolgte im Rahmen der Arbeit mit dem Haskell-Tool `Alex`, einem Scanner-Generator auf der Basis von Haskell. Mit Hilfe der von `Alex` erzeugten Scanner ist es möglich, die textuelle Eingabe des $\underline{Nutzer_{SBE(L)}}$ auf ihre lexikalische Korrektheit zu prüfen, um entscheiden zu können, ob die textuelle Eingabe in die interne Darstellung $\underline{EK_L < AbstrakteSyntax_L >}$ von $\underline{EK_L}$ eingepaßt werden kann, oder ob sie vom $\underline{SBE(L)}$ abzuweisen ist.
Es wurde für jeden der sechs nichtkonstruktorbasierten Typen ein eigener Scanner spezifiziert. Der Grund dafür wird noch erläutert.

Wir beschreiben nun die Scanner-Spezifikationen, aus denen `Alex` die jeweiligen Haskell-Codes generiert, welche die Scanner realisieren. Die Beschreibung erfolgt am Beispiel der Scanner-Spezifikation für den nichtkonstruktorbasierten Typ *IntInput*.
Innerhalb einer Spezifikation wird Programm-Code, welcher in den zu generierenden Haskell-Code unverändert einzubinden ist, mit `%{ ... %}` geklammert. Die Spezifikation des Scanners für den Typ *IntInput* beginnt mit einem solchen Abschnitt:

```
%{
module UserIntInputScanner (userIntInputScanErg) where

import Alex
import UserInput
%}
```

Das zu generierende Haskell-Modul bekommt den Namen **UserIntInputScanner** und exportiert die Scan-Funktion `userIntInputScanErg`. Die importierten Module **Alex** bzw.

UserInput enthalten die vom Scanner-Generator `Alex` benötigten Definitionen bzw. die Daten-Typdefinitionen für die noch einzuführenden Token.

```
-- Alex-Anweisung
"userIntInput_lx"/"userIntInput_acts":-
```

Diese Zeile leitet die Scannerdefinition ein und weist `Alex` an, die generierten Tabellen mit userIntInput_lx zu benennen und die Aktionen für jedes Token als userIntInput_acts auszugeben.

```
-- Token-Definitionen
<>             ::= ^w+
<userInt>   ::= ([^-](1-9)(0-9)*)|(0|(1-9)(0-9)*)
```

Die erste Token-Definition wird genutzt, um Whitespace-Zeichen beim Scannen zu übergehen, die zweite definiert das Token userInt.
Das Symbol ^ verwenden wir in einer Alex-Spezifikation, um Makros oder Sonderzeichen benutzen zu können, die durch das nachfolgende Zeichen bestimmt sind. Das vordefinierte Makro `^w` steht für ein Leerschritt-, Tabulator- oder Zeilenwechsel-Zeichen. Mit `c+` bezeichnen wir eine beliebige Wiederholung von `c`, wobei `c` mindestens einmal vorhanden sein muß. Die beliebige Wiederholung von `c`, bei der auch das leere Wort ε zuläßig ist, spezifizieren wir durch `c*`. Das Symbol `.` steht für ein beliebiges Zeichen außer dem Zeilenwechsel-Zeichen. Für die Vorrangregelung verwenden wir `(...)` und `|` bezeichnet die Alternative. Durch `[...]` wird ein beliebiges Element aus diesem Klammerausdruck beschrieben. Mit `~c` kennzeichnen wir ein beliebiges Zeichen außer `c`. Das vordefinierte Makro `^p` bezeichnet ein druckbares Zeichen und mittels `#` wird die Zeichenmengen-Differenz gebildet.

```
-- Scan-Funktionen
%{
userInt p s              = (UserIntInput s, p)

userIntInput :: String -> [(UserInput, Posn)]
userIntInput inp = scan userIntInput_scan inp

userIntInput_scan :: Scan (UserInput, Posn)
userIntInput_scan = load_scan (userIntInput_acts,stop_act) userIntInput_lx
  where
  stop_act p ""  = [(EndOfUserInput, p)]
  stop_act p inp = [(UserInputScanError inp, p)]

userIntInputScanErg :: String -> (Bool,String)
userIntInputScanErg s =
  (let scanResult = (userIntInput s)
     in (case scanResult of
              ((UserIntInput ui1, _):[(EndOfUserInput,_)]) -> (True,ui1)
              _                                 -> (False,"")))
%}
```

Die Funktion `userInt` gibt die gescannten Token aus. Sie bekommt vom Scanner als Argumente die Position und die zugehörige Zeichenkette, als Ergebnis liefert sie ein Paar, in dem das Token und seine Position enthalten ist.
Die Funktion `userIntInput` stellt den spezifizierten Scanner bereit. Sie bekommt als Eingabe den Eingabetext und gibt eine Liste von Paaren zurück, welche jeweils ein Token und seine Position enthalten.
Die Aufgabe der Alex-spezifischen Funktion `userIntInput_scan` liegt darin, daß beim Erreichen des Eingabetextendes das Token EndOfFile eingefügt und beim Auftreten einer Fehlersituation beim Scannen das Token ScanError in die zurückgelieferte Liste mit aufgenommen wird.
Die Funktion `userIntInputScanErg`, die vom erzeugten Haskell-Modul exportiert wird, bekommt bei ihrer Anwendung im $\underline{SBE(L)}$ als Argument den vom $\underline{Nutzer_{SBE(L)}}$ eingegebenen Text, ruft die Funktion `userIntInput` auf und liefert ein Ergebnispaar zurück. Das erste Element dieses Paares ist vom Typ Bool und trägt im Falle eines erfolgreich beendeten Scanvorgangs den Wert True, ansonsten False. Der Typ des zweiten Paarelementes ist String und der Wert ist entweder der (gegebenfalls von Whitespace-Zeichen gesäuberte) Eingabetext oder im Falle eines fehlerhaften Scanvorgangs der leere String.

Das Modul **UserInput** sowie die fünf restlichen Scanner-Spezifikationen für die vordefinierten nichtkonstruktorbasierten Typen können nachgelesen werden (sie befinden sich auf der beigelegten Diskette). Es sei aber darauf hingewiesen, daß sich die anderen Scanner-Spezifikationen nur in Bezeichnern und der Token-Definition unterscheiden. Ersetzt man in allen Bezeichnern, in denen der Teilstring "Int" vorkommt, diesen Teilstring mit dem Namen eines anderen vordefinierten Typs (ohne die Endung "Input"), so erhält man die entsprechende Scanner-Spezifikation, wobei natürlich noch die Token-Definition angepaßt werden muß. Wir geben daher in folgender Übersicht alle Token-Definitionen der sechs vordefinierten Typen an:

<table>
<tr><th>Typname</th><th>Token</th><th>regulärer Ausdruck (Alex-Syntax)</th></tr>
<tr><td>IntInput</td><td><code><userInt></code></td><td><code>([^-](1-9)(0-9)*)|(0|(1-9)(0-9)*)</code></td></tr>
<tr><td>BoolInput</td><td><code><userBool></code></td><td><code>True | False</code></td></tr>
<tr><td>CharInput</td><td><code><userChar></code></td><td><code>^'(^p#[^'^\])^'</code></td></tr>
<tr><td>StringInput</td><td><code><userString></code></td><td><code>.+</code></td></tr>
<tr><td>IdentInput</td><td><code><userIdent></code></td><td><code>((a-z)|(A-Z))((a-z)|(A-Z)|(0-9)|^_)*</code></td></tr>
<tr><td>QuotedStringInput</td><td><code><userQuotedString></code></td><td><code>^"((^p#[^"^\])|^\^p)*^"</code></td></tr>
</table>

Eine erfolgreich gescannte textuelle Eingabe vom $\underline{Nutzer_{SBE(L)}}$ wird im $\underline{SBE(L)}$ intern stets als String repräsentiert. Soll dieser String in $\underline{EK_L}$ eingefügt werden, so erfolgt dies über das Ersetzen des in $\underline{EK_L < AbstrakteSyntax_L >}$ aktuell ausgewählten Teilbaums durch den Teilbaum, welcher aus einem einstelligen Konstruktor, für dessen Nachfolgertyp die textuelle Eingabe des $\underline{Nutzer_{SBE(L)}}$ erfolgreich gescannt worden ist, und dem String als Nachfolger besteht.
Das Verarbeiten der textuellen Eingabe des $\underline{Nutzer_{SBE(L)}}$ mit dem Ziel der Erzeugung einer entsprechend geänderten internen Darstellung von $\underline{EK_L}$ erfolgt durch die Verwen-

dung der Funktion `textScanTries` während der Anwendung der Travers-Funktion auf den Wurzel-Konstruktor des aktuell ausgewählten Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$. Die Funktion `textScanTries` ruft dabei für alle vordefinierten Typen, welche zur textuellen Eingabe an dieser Stelle in Frage kommen, die Hilfsfunktion `textScanTryAndTrans` auf und erzeugt dabei eine Liste von textuell transformierten $\underline{EK_L < AbstrakteSyntax_L >}$, wobei sie als Ergebnis das erste Element dieser Liste zurückgibt. Die Listenelemente, die von der Funktion `textScanTryAndTrans` erzeugt werden, tragen nur im Falle eines erfolgreichen Scanvorgangs (in bezug auf den jeweiligen vordefinierten Typ) den Wert des entsprechend textuell transformierten $\underline{EK_L < AbstrakteSyntax_L >}$ (durch Anwendung der Funktion `execTransCall`), ansonsten erzeugt die Funktion ein Listenelement mit dem Wert der leeren Liste, welches durch die Anwendung Listenvereinigung nichts zu der von der Funktion `textScanTryAndTrans` zu erzeugenden Liste beiträgt. Aufgrund der Eigenschaft der Sprache Haskell, Funktionen "lazy" auszuwerten, berechnet die Funktion `textScanTries` die Liste der textuell transformierten $\underline{EK_L < AbstrakteSyntax_L >}$ nur so weit, bis sie ein erstes Element besitzt (da nur das erste angefordert wird). Es kommt also höchstens einmal zu einer textuellen Transformation von $\underline{EK_L < AbstrakteSyntax_L >}$. Die zu erzeugende Liste besitzt immer mindestens ein Element, dieses trägt den Wert des unveränderten $\underline{EK_L < AbstrakteSyntax_L >}$ und wird nur dann zum ersten Element der Liste, falls alle vorhergehenden Scanvorgänge erfolglos beendet wurden.

Wir besprechen nun den Grund für die Entscheidung, für jeden der sechs vordefinierten Typen einen eigenen Scanner zu erstellen. Wenn für jeden vordefinierten Typ ein eigener Scanner vorliegt, kann es bei der Anwendung eines Scanners nicht zu Problemen mit der Erkennung von regulären Ausdrücken, die sich überschneiden (wie z.B. IntInput und StringInput), kommen. Hätte man nur einen gemeinsamen Scanner, so würde man nicht zweifelsfrei entscheiden können, ob z.B. der $\underline{Nutzer_{SBE(L)}}$ mit der textuellen Eingabe "55" eine Zahl (IntInput) oder eine Zeichenkette (StringInput) meint, selbst wenn die aktuelle Auswahl eines Teiles von $\underline{EK_L}$ keine Eingabe von Zahlen erlaubt.
Das Problem bleibt allerdings bestehen, wenn der $\underline{Entwickler_{SBE(L)}}$ bei der Beschreibung der abstrakten Syntax der Sprache $\underline{L}$ die nichtkonstruktorbasierten Typen so einsetzt, daß im Falle einer textuellen Eingabe bei aktueller Auswahl eines bestimmten Teiles von $\underline{EK_L}$ mehrere Scanner aufgerufen werden müssen, bei denen sich die zu prüfenden regulären Ausdrücke überschneiden (wie z.B. IntInput und StringInput). In diesem Fall wird in der von uns implementierten Version die textuelle Eingabe dem Typ zugeordent, für den sie zuerst korrekt gescannt werden kann. Die Scanner werden hierbei in der gleichen Reihenfolge aufgerufen in der die zugehörigen vordefinierten nichtkonstruktorbasierten Typen in der entsprechenden *Data-Definition* vorkommen. Besitzt z.B. der *Data-Block* zur Beschreibung der abstrakten Syntax von $\underline{L}$ die *Data-Definition*

data KonfliktTyp = INT IntInput | STRING StringInput ,

hat außerdem noch der Wurzel-Konstruktor des aktuell ausgewählten Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$ den Typ KonfliktTyp und tätigt nun der $\underline{Nutzer_{SBE(L)}}$ die textuelle Eingabe von "55", so würde der aktuell ausgewählte Teilbaum durch den Teilbaum (INT "55") ersetzt und damit die Eingabe als ganze Zahl aufgefaßt, auch wenn der $\underline{Nutzer_{SBE(L)}}$ eine Zeichenkette eingeben wollte.
Dieses Problem kann der $\underline{Entwickler_{SBE(L)}}$ verhindern, wenn er die besprochene *Data-*

Definition z.B. durch folgende drei *Data-Definitionen* ersetzt:

data KonfliktTyp = INTCONST Inttyp | STRING Stringtyp
data Inttyp = INT IntInput
data Stringtyp = STRING StringInput

6.3 Quell-Code für Phase 6 - $\underline{UC_{NS} < Java >}$

Die Implementierung der Nutzerschnittstellenkomponente eines mit dem System $\underline{Gen_{SBE}}$ aus einer $\underline{Spez_{SBE(L)}}$ zu erzeugenden $\underline{SBE(L)}$ erfolgte in der objektorientierten Programmiersprache Java. Bei der Implementierung des Java-Codes $\underline{UC_{NS} < Java >}$ wurde das Ziel verfolgt, die Erzeugung der Nutzerschnittstellenkomponente völlig unabhängig von einer $\underline{Spez_{SBE(L)}}$ zu halten.
Die moderne und heutzutage in der Praxis viel verwendete objektorientierte Sprache Java eignete sich sehr gut zur Implementierung der ereignisgesteuerten, fensterorientierten Oberfläche (Nutzerschnittstellenkomponente) der zu erzeugenden $\underline{SBE(L)}$. Durch das Konzept der Klassenvererbung dieser Sprache konnte die Grundfunktionalität der vom Java Development Kit 1.2 zur Verfügung gestellten Klassen, deren Instanzen Oberflächenkomponenten implementieren, wie z.B. Textfenster, Kommandoknöpfe etc., genutzt werden. Ebenso konnten wir bei der Programmierung des gewünschten ereignisbasierten Verhaltens der Peripheriegeräte Maus und Tastatur auf Grundfunktionen zurückgreifen, welche das Java Development Kit 1.2 seinem Nutzer bereitstellt.
Die zwei Hauptklassen des Programm-Codes $\underline{UC_{NS} < Java >}$ sind folgende: Die Klasse **Editor** übernimmt die Darstellung der fensterorientierten Oberfläche und deren ereignisbedingte Steuerung. Die Kommunikation mit der Auswerterkomponente des zu erzeugenden $\underline{SBE(L)}$ wird von der Klasse **Exposition** realisiert. Wir verzichteen hier aber auf eine konkrete Beschreibung der Implementation dieser Klassen, da wir uns in dieser Arbeit nicht so sehr auf die Nutzerschnittstelle, sondern hauptsächlich auf die Grundideen und die Implementierung der Auswertungsstrategien, die in der Auswerterkomponente eines $\underline{SBE(L)}$ zur Anwendung kommen, befassen. Der Programm-Code $\underline{UC_{NS} < Java >}$ ist aber in dokumentierter Form auf der der Arbeit beigefügten Diskette zu finden, so daß es dem interessierten Leser nicht allzu schwer fallen sollte, sich in die konkrete Implementation der Nutzerschnittstellenkomponente eines mit dem System $\underline{Gen_{SBE}}$ erzeugten sprachbasierten Editors einzuarbeiten.

Kapitel 7

Ausblick

7.1 Ergebnisse, Probleme sowie Änderungs- und Erweiterungsmöglichkeiten des Systems $\underline{Gen_{SBE}}$

In dieser Arbeit erfolgte die Erstellung eines Konzeptes für ein System, welches zu einer Haskell$^+$-Spezifikation $\underline{Spez_{SBE(L)}}$ einer (Programmier-)Sprache $\underline{L}$ einen sprachbasierten (bzw. syntaxgesteuerten) Editor $\overline{SBE(L)}$ für $\underline{L}$ generiert. Die Implementierung einer ersten Version dieses System, das wir in der Arbeit mit $\underline{Gen_{SBE}}$ bezeichnet haben, erfolgte nach dieser Konzeption unter Verwendung der Programmiersprachen Haskell und Java. Dabei wurde besonders auf die Kompatibilität zu den bereits existierenden Implementierungs-, Analyse- und Transformationsprogrammen sowie auf die Möglichkeit der späteren Erweiterbarkeit des Systems $\underline{Gen_{SBE(L)}}$ geachtet.
Die Implementierung des Generatorsystems beruht so auf den Implementierungen, welche im Rahmen der Arbeit [Les99] von Herrn Lescher entwickelt worden sind. An dieser Stelle sei der Leser nochmal auf den Abschnitt 4.6 der hier vorliegenden Arbeit verwiesen, in dem die Generierung eines $\underline{Spez_{SBE(L)}}$ aus einer $\underline{Spez_{SBE(L)}}$ durch den $\underline{Gen_{SBE(L)}}$ phasenweise beschrieben wird. Für Phase 1 wurden die von Herrn Lescher zur Verfügung gestellten Scanner und Parser für Haskell$^+$ verwendet. Außerdem bedienten wir uns gewisser Teile des Programmcodes von Herrn Lescher für die Übersetzung eines abstrakten Syntaxbaumes eines Haskell$^+$-Programms in ein entsprechendes Haskell-Programm.
Weiterhin wurden in Phase 2 die Analyseprogramme von Hernn Lescher zur Überprüfung der kontextsensitiven Nebenbedingungen der Sprache Haskell$^+$ sowie der Prüfung der Nichtzirkularität von Makroattributgrammatiken eingebunden. Diese Analysen werden auf dem Format $\underline{Spez_{SBE(L)} < AbstrakeSyntax_{Haskell^+} >}$ angewendet. Der in diesem Format vorliegende Syntaxbaum eignet sich hervorragend als Schnittstelle zu anderen Analyse- und Transformationsprogrammen.
So könnte hier auch leicht das Analyse-Programm zum Testen der Eigenschaften von Makro Attributed Tree Transducern, die eventuell innerhalb der $\underline{Spez_{SBE(L)}}$ vorkommen, zum Einsatz kommen. Dieses Analyse-Programm wurde von Herrn Vater entwickelt und prüft die Klassenzugehörigkeit eines Tree Transducers. Außerdem wird untersucht, ob er z.B. die Vorrausetzungen für einen *single-used* oder *producing* Tree Transducer erfüllt sowie

welche Eigenschaften bezüglich der Linearität er besitzt. Diese Eigenschaften interessieren besonders, wenn man sich Gedanken um Optimierungen (in Hinblick auf Speicherplatz und Laufzeitverhalten der Implementierungen) der Tree Transducer macht.
In Hinblick auf solche Optimierungen wurden von Herrn Hoeff Transformationen (insbesondere auch Kompositionen und Dekompositionen) von Tree Transducern mit bestimmten Eigenschaften programmiert. Diese Transformationsprogramme könnten ebenfalls auf das Format $\underline{Spez_{SBE(L)} < AbstrakeSyntax_{Haskell^+}}$ angewendet werden. Dies könnte durch Umprogrammierung der Steuerung von Phase 2 in dieser Phase automatisch erfolgen (durch vorherige Anwendung des Analyseprogramms von Herrn Vater). Leider sind die theoretischen Vorarbeiten aber noch nicht an dem Punkt, an dem solcher Automatismus erstellt werden könnte, der dann auch zweifelsfrei zu einer Optimierung führt. Denkbar wäre daher auch alternativ die Anwendung der Transformationsprogramme von Herrn Hoeff in Abhängigkeit der Entscheidung des $\underline{Entwickler_{SBE(L)}}$, der den vom $\underline{Gen_{SBE(L)}}$ durchgeführten Generierungsprozess in Phase 2 stoppt, die Transformationen "per Hand" aufruft und danach den Generierungsprozess wieder weiterlaufen läßt (ab Phase 3).
Es sei aber nochmal darauf hingewiesen, daß die Makroattributgrammatiken, aus denen eine $\underline{Spez_{SBE(L)}}$ besteht, nicht immer Makro Attributed Tree Transducer darstellen, da diese als Spezialfall anzusehen sind. Nichtsdestotrotz könnte man sich aber auch Transformationen zur Optimierung von Makroattributgrammatiken vorstellen, allerdings fehlen dafür noch entscheidende theoretische Aussagen und Algorithmen.

Die Implementation der $\underline{Gen_{SBE(L)}}$ erfolgte außerdem so, daß ein aus einer $\underline{Spez_{SBE(L)}}$ generierter Editor $\underline{SBE(L)}$ über eine fensterorientierte, ereignisgesteuerte Oberfläche (Nutzerschnittstelle) verfügt. Allerdings trat hierbei das Problem auf, daß in der vorliegenden ersten Version des $\underline{Gen_{SBE(L)}}$ die Nutzerschnittstellenkomponente in einer anderen Programmiersprache als die Auswerterkomponente des $\underline{SBE(L)}$ implementiert werden muß. Dadurch und durch die Art der Kommunikation (über das Dateisystem des Rechners, auf dem der $\underline{SBE(L)}$ ausgeführt wird) kommt es zu spürbaren Effizienzverlusten. Verstärkt wird dieser Nachteil noch durch die nichtinkrementellen Berechnungen der Auswerterkomponente des $\underline{SBE(L)}$. Da aber während der Verwendung des $\underline{SBE(L)}$ durch den $\underline{Nutzer_{SBE(L)}}$ sehr häufig Makroattributberechnungen notwendig werden (vorallem die Berechnung des Wertes des Unparsing-Attributes, d.h. der Berechnung der externen Darstellungsform $\underline{EK_L < AusgabeSyntax_L >}$ von $\underline{EK_L}$ aus der internen Darstellung $\underline{EK_L < Abstraktesyntax_L >}$), kommt es zu unschönen Verzögerungen beim Editieren, welche linear abhängig (proportional) zur Größe von $\underline{EK_L}$ sind. Beide Probleme können aber gelöst werden, indem ein inkrementeller Auswerter in Java generiert wird, was die Änderung der Phasen 3, 4 und 5 erfordern würde. Natürlich müßte der Quell-Code für Phase 5 ebenfalls ersetzt werden. Die Erzeugung eines inkrementellen Auswerters in der Sprache Java (aus dem Resultat von Phase 2, d.h. $\underline{Spez_{SBE(L)} < AbstrakeSyntax_{Haskell^+} >}$ dient als Schnittstelle) ist bereits in der Professur GdP (Grundlagen der Programmierung) geplant. Durch die Verwendung von Java zur Implementierung der Auswerterkomponente des $\underline{SBE(L)}$ kann auch die Kommunikation zwischen dieser Komponente und der Nutzerschnittstellenkomponente viel effizienter gestaltet werden. Da die Nutzerschnittstellenkomponente des $\underline{SBE(L)}$ so konzipiert worden ist, daß sie völlig unabhängig von der $\underline{Spez_{SBE(L)}}$ erzeugt wird und funktioniert, muß hierfür nur ein sehr geringer Teil des Quell-

Codes $\underline{UC_{NS} < Java >}$ für Phase 6 (derjenige, der die Funktionen zur Kommunikation mit der Auswerterkomponte erzeugt) umprogrammiert werden.

Ein weiterer Nachteil der hier vorliegenden ersten Implementierungsversion des $\underline{Gen_{SBE}}$ liegt in der noch unzureichenden Nutzerfreundlichkeit der Oberfläche eines generierten $\underline{SBE(L)}$. Nachfolgende Versionen sollten daher Oberflächen bereitstellen, mit denen der $\underline{Nutzer_{SBE(L)}}$ mehr intuitiv arbeiten kann, welche eine höhere Funktionalität aufweisen und robuster sind (zur Zeit stürzt z.B. ein $\underline{SBE(L)}$ noch ab, wenn eine fehlerhafte Datei zum Laden eines $\underline{EK_L}$ verwendet wird). Da die Programmierung des Programm-Codes $\underline{UC_{NS} < Java >}$ objektorientiert erfolgte, sind Änderungen dieses Codes zur Generierung der Oberfläche leicht möglich.

In folgenden Versionen des $\underline{Gen_{SBE}}$ ist dringend die Überprüfung aller kontextsensitiven Nebenbedingungen der Sprache Haskell$^+$ anzuraten. In dieser Implementierungsversion fehlt noch die Überprüfung der Nebenbedingungen, die sich durch die hier vorliegende Arbeit in bezug auf die in [Les99] vorgestellten geändert haben und jenen, welche neu hinzugekommen sind. Diese Überprüfung der neuen kontextsensitiven Nebenbedingungen verlangt eine Erweiterung der in Phase 2 stattfindenden Analyse des Formates $\underline{Spez_{SBE(L)} < AbstrakeSyntax_{Haskell^+}}$ und wird von uns als nicht sehr aufwendig eingeschätzt.
In diesem Zusammenhang sei noch auf einen weiteren Nachteil bei der Verwendung von $\underline{Spez_{SBE(L)}}$ verwiesen, der darin liegt, daß der Haskell-Anteil nicht gescannt und geparst wird und somit keine Überprüfungen und Optimierungen dieses Teils möglich sind. Dies ist ebenfalls eine interessante und lohnenswerte Aufgabe für spätere Versionen. Weiterhin wäre auch eine automatische Typinferenz bei der Verwendung des Haskell$^+$-Progamms einer $\underline{Spez_{SBE(L)}}$ sinnvoll.

In der vorliegenden ersten Version des Systems $\underline{Gen_{SBE}}$ werden von einer Spezifikation einer (Programmier-) Sprache $\underline{L}$ die abstrakte Syntax, die Unparsing-Funktionen, die Transform-Funktion und weiterhin auch die synthetischen und inheriten Funktionen zur Beschreibung der kontextsensitiven Nebenbedingungen der Sprache $\underline{L}$ berücksichtigt. Die textuelle Eingabemöglichkeit bei dem erzeugten $\underline{SBE(L)}$ beschränkt sich auf einfache Sprachelemente wie z.B. Bezeichner, Buchstaben, ganze Zahlen, Strings und booleschen Werten.
Unter diesen Voraussetzungen erstellt Herr Motzkau gerade die Spezifikation eines sprachbasierten Editor für die Sprache Haskell$^+$. Dadurch können im größeren Umfang Erfahrungen mit dem System $\underline{Gen_{SBE}}$ (Stärken und Schwächen) gesammelt werden. Außerdem kann der nach Fertigstellung von $\underline{Spez_{SBE(Haskell^+)}}$ mit dem $\underline{Gen_{SBE}}$ generierbare $\underline{SBE(Haskell^+)}$, als sprachbasierter Editor für die Eingabesprache des $\underline{Gen_{SBE}}$, also zur Hilfe beim Erstellen von beliebigen $\underline{Spez_{SBE(L)}}$ genutzt werden. Dabei können auch die von Herrn Hoeff programmierten Transformations-Programme (auf die wir weiter oben in diesem Kapitel schon hingewiesen haben) so vom $\underline{Entwickler_{SBE(Haskell^+)}}$ in der $\underline{Spez_{SBE(Haskell^+)}}$ eingesetzt werden, daß ihre Funktionalität im $\underline{SBE(Haskell^+)}$ vom $\underline{Nutzer_{SBE(Haskell^+)}}$ in Form von strukturellen Transformationen sozusagen "auf Knopfdruck" genutzt werden kann.

7.2 Vorschläge für Spracherweiterungen von Haskell$^+$

Zum Abschluß möchten wir noch ein paar Vorschläge zur Erweiterung der Sprache Haskell$^+$ besprechen.
Die Sprache Haskell$^+$ wurde aus zwei verschiedenen Gründen ins Leben gerufen. Zum einen soll sie als Beschreibungssprache für Makroattributgrammatiken dienen. Zum anderen birgt sie die Möglichkeit zur Spezifikation von sprachbasierten Editoren, welche dann als Eingabe eines System zur Generierung dieser Editoren, wie z.B. des in dieser Arbeit vorgestellten Systems $\underline{Gen_{SBE}}$, dienen kann. Bei der Erstellung einer solchen Spezifikation ist es notwendig besondere Sprachelemente wie z.B. Unparse-Funktionen oder eine Transform-Funktion zur Verfügung zu haben. Weiterhin ist auch die Möglichkeit zur Beschreibung einer Eingabesprache sinnvoll. All dies wird aber nicht benötigt, falls wir "nur" Makroattributgrammatiken beschreiben wollen. Daher ist es ratsam die Entwicklung der Sprache Haskell$^+$ getrennt in bezug auf die zwei genannten Ziele weiterzuführen. Dies würde zu zwei verschiedenen Sprachen führen. Dann wäre es zum einen möglich, bei der puren Beschreibung von Makroattributgrammatiken sehr nahe an der der Sprachgestaltung zu verbleiben, wie sie bei funktionalen Sprachen üblich ist, um einem mit der Kunst der funktionalen Programmierung vertrauten Programmierer ein leichtes Einarbeiten in die Sprache eine Weiterentwicklung der Sprache Haskell$^+$ zu ermöglichen. Die andere und davon unabhängige Weiterentwicklung von Haskell$^+$ könnte dann so gestaltet werden, daß sie dem $\underline{Entwickler_{SBE(L)}}$ die Programmierung einer $\underline{Spez_{SBE(L)}}$ wesentlich erleichtert, indem man solche Sprachkonstrukte anbietet, die speziell auf diese Aufgabe zugeschnitten sind. Diese verlangen dann zwar am Anfang etwas mehr Einarbeitungsarbeit, ermöglichen aber bei ihrer Kenntnis eine komfortablere Programmierung einer $\underline{Spez_{SBE(L)}}$ zur Erzeugung eines $\underline{SBE(L)}$.
Wir geben im folgenden ein paar Vorschläge zur Erweiterung der Sprache Haskell im zuletzt genannten Sinne:

Zuerst ist natürlich die verbesserte Möglichkeit zur Beschreibung einer Eingabesyntax der zu beschreibenden Sprache $\underline{L}$ zu schaffen. Dafür ist zunächst einmal die Beschreibung von regulären Ausdrücken zu etablieren, welche die vom zu generierenden $\underline{SBE(L)}$ erlaubten textuellen Eingaben lexikalisch festlegen. Danach ist für den $\underline{Entwickler_{SBE(L)}}$ in Haskell$^+$ die Möglichkeit zur Festlegung der Definition eines Parsers für die vom $\underline{Nutzer_{SBE(L)}}$ getätigte textuelle Eingabe zu schaffen. Dieser Parser ist notwendig um textuelle Eingaben, welche komplexe Sprachkonstrukte der Sprache $\underline{L}$ repräsentieren, ordnungsgemäß in die interne Darstellungsform $\underline{EK_L < AbstrakteSyntax_L >}$ des vom $\underline{Nutzer_{SBE(L)}}$ bearbeiteten $\underline{EK_L}$ übertragen zu können bzw. als nicht korrekte Eingabe ablehnen zu können. Als Anregung zur Art und Weise der Beschreibung der gewünschten Parsevorgänge sowie der Weiterverwendung des dabei entstehenden Parsebaums kann die Herangehensweise bei dem System SGen (siehe dazu [RT89], Abschnitt 5.4) dienen.
In Zusammenhang mit der Beschreibung der Eingabesyntax der Sprache $\underline{L}$ innerhalb von $\underline{Spez_{SBE(L)}}$ wäre es auch sinnvoll, wenn der $\underline{Entwickler_{SBE(L)}}$ festlegen könnte, welche Sprachkonstrukte der Sprache $\underline{L}$ textuell mit dem $\underline{SBE(L)}$ editiert werden dürfen und welche nicht. Wenn man die textuelle Eingabe von bestimmten Sprachkonstrukten von vornherein verbietet, erhöht man die Sicherheit vor syntaktischen Fehlern beim Editieren

eines $\underline{EK_L}$, da man eine höhere Programmierdisziplin beim $\underline{Nutzer_{SBE(L)}}$ erzwingt.
Um den Sinn des folgenden Spracherweiterungsvorschlages zu verstehen, sei daran erinnert, daß es dem $\underline{Entwickler_{SBE(L)}}$ möglich ist, mehrere Ausgabesyntaxen für die Sprache $\underline{L}$ durch die Definition mehrerer Unparse-Funktion zu beschreiben. Damit der $\underline{Nutzer_{SBE(L)}}$ bei seiner Arbeit mit dem $\underline{SBE(L)}$ die Möglichkeit hat, eine bestimmte aus vielen auszuwählen, wäre eine eindeutige Bezeichnung für jede Ausgabesyntax sinnvoll. Dafür eignet sich der Namen der zugehörigen Unparse-Funktion selten. Daher wird vorgeschlagen, die Deklaration einer Unparse-Funktion in $\underline{Spez_{SBE(L)}}$ um den Namen der damit beschriebenen Ausgabesyntax zu erweitern.
Wir wenden uns nun einem anderen Sprachkonstrukt der Sprache Haskell$^+$ zu, nämlich dem *Data-Block*. In jeder $\underline{Spez_{SBE(L)}}$ ist ein *Data-Block* zur Beschreibung der abstrakten Syntax der Sprache $\underline{L}$ erforderlich. Es wäre sinnvoll diesen durch ein Schlüsselwort kenntlich zu machen, da nur dieser bei den Definitionen von Unparse-Funktionen und der Definition der Transform-Funktion verwendet werden darf. Da alle diese Funktionen nur in *Mag-Blöcken* definiert werden dürfen, welche diesselbe *Root-Symbol*-Definition (Konstruktor aus dem besprochenen *Data-Block*) aufweisen, wäre es sinnvoll die Definition des *Root-Symbol* in den *Data-Block* selbst zu verlegen. Weiterhin wäre es sicher besser, wenn man die modulare Beschreibung von *Data-Blöcken* zulassen würde. Zu diesem Zeitpunkt darf man in den rechten Seiten der *Data-Definitionen* eines *Data-Blocks* nicht auf Datentypen zurückgreifen, welche in anderen *Data-Blöcken* definiert worden sind. Gerade bei der Spezifikation einer umfangreichen Sprache $\underline{L}$ ist es aber angebracht die abstrakte Syntax dieser Sprache modular beschreiben zu können, damit $\underline{Spez_{SBE(L)}}$ übersichtlicher programmiert werden kann.
Als nächstes befassen wir uns mit der Definition einer Transform-Funktion. Die linke Seite einer Definitionsgleichung dieser Art legt u.a. fest, wann für den $\underline{Nutzer_{SBE(L)}}$ eine strukturelle Transformation des $\underline{EK_L}$ erlaubt ist. Da aber das Pattern in der linken Seite nicht mehrstufig sein darf, kann nur anhand des Wurzelkonstruktors eines Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$ entschieden werden, ob eine strukturelle Transformation dieses Teilbaums durchgeführt werden kann oder nicht. In einigen Fällen entscheidet sich dies aber nicht nur anhand des Wurzelkonstruktors und damit wird derzeit eine strukturelle Transformation vom $\underline{SBE(L)}$ auch dann angeboten, wenn zwar der Wurzelkonstruktor stimmt, nicht aber der übrige Aufbau des vom $\underline{Nutzer_{SBE(L)}}$ ausgewählten Teilbaums von $\underline{EK_L < AbstrakteSyntax_L >}$. In diesem Fall wird beim Aufruf der strukturellen Transformation das $\underline{EK_L}$ zwar nicht transformiert, aber das Erscheinen des entsprechenden Kommandoknopfes zur Auslösung der strukturellen Transformation ist verwirrend für den $\underline{Nutzer_{SBE(L)}}$. Als Beispiel sei auf die Definitionsgleichung der Transform-Funktion aus Abschnitt 4.4 verwiesen, welche die Transform-Funktion für den einstelligen Konstruktor `COMPOUND` und den Kontextparameter `DisBlock` definiert. Durch die Verwendung der einstelligen Haskell-Funktion `disBlock` werden nur Teilbäume der Struktur `(COMPOUND (STMTLISTPAIR _ STMNTLISTNIL))` transformiert, aber der strukturelle Transformation mit der Bezeichnung `DisBlock` wird vom $\underline{SBE(P)}$ für alle Teilbäume des $\underline{EK_P < AbstrakteSyntax_P >}$ mit der allgemeineren Struktur `(COMPOUND _)` angeboten. Es wäre also sehr ratsam dem $\underline{Entwickler_{SBE(P)}}$ zu erlauben, die Pattern in der linken Seite einer Transform-Funktionsdefinitionsgleichung mehrstufig zu beschreiben.
Es ist in einer $\underline{Spez_{SBE(L)}}$ möglich in der rechten Seite einer Transform-Funktionsgleichung

inherite und synthetische Funktionen aufzufrufen. Diese müssen natürlich einer Makroattributgrammatik entnommen sein, welche die Eigenschaften eines Macro Attributed Tree Transducers besitzt. Ein Macro Attributed Tree Transducer besitzt aber immer ein ausgezeichnetes synthetisches Attribut, welches an der Wurzel definiert ist und die Berechnung des Tree Transducers realisiert. Dieses synthetische Bedeutungsattribut kann aber in einer rechten Seite der Transform-Funktionsdefinitionsgleichung nicht an der Wurzel des zu transformierenden Teilbaums aufgerufen werden, da die Transform-Funktion als spezielle synthetische Funktion angesehen wird und daher die Normalform bei der Definition von synthetischen Makro-Attributen verletzt werden würde. Da aber die Transform-Funktion sich nicht selbst rekursiv aufrufen darf, würde es nicht zu Problemen führen, wenn man die Normalform an dieser Stelle in solcher Weise verletzen würde. Daher schlagen wir vor, Aufrufe von synthetischen Funktionen in der rechten Seite einer Defintionsgleichung der Transform-Funktion, auch auf den durch das Pattern in der linken Seite der Gleichung beschriebenen Teilbaum, zu erlauben.
Weiterhin wäre es von Vorteil in der rechten Seite eine Definitionsgleichung der Transform-Funktion do-Ausdrücke verwenden zu können (allerdings nicht als Parameter der Aufrufe von synthetischen und inheriten Funktionen). Dies würde die Beschreibungsmöglichkeiten des $\underline{Entwickler_{SBE(L)}}$ erhöhen, aber nicht zu Problemen bei der Makro-Attribut-Auswertung führen (mit einem do-Ausdruck verläßt man ja genaugenommen das Feld der funktionalen Programmierung), da die Transform-Funktion sich ja nicht rekursiv aufrufen darf. Außerdem könnte man in den rechten Seiten aller Funktionen, die innerhalb eines *Mag-Blockes* definiert werden, die Verwendung von let-Ausdrücken und auch von case-Ausdrücken (insbesondere auch if-then-else-Ausdrücke), die sich ja auf bedingte Anweisungen zurückführen lassen, erlauben.
Unser letzter Vorschlag bezieht sich auf die gesamte $\underline{Spez_{SBE(L)}}$. Es ist schon möglich die enthaltenen *Mag-Blöcke* innerhalb einer Datei modular zu programmieren, aber man kann keine spezifizierten Funktionen aus den *Mag-Blöcken* zur Verwendung in anderen Dateien (Modulen) exportieren bzw. solche Funktionen aus anderen Dateien (Modulen) importieren. Dies könnte der Form nach wie in Haskell beschrieben werden, d.h. die Datei beginnt mit dem Namen des Moduls, danach folgen geklammert die zu exportierenden Funktionen (ansonsten wird alles exportiert) und in den sich anschließenden Zeilen können die Namen der Module aufgeführt werden, aus denen importiert werden soll. Zur Zeit wird die $\underline{Spez_{SBE(L)}}$ als ganzheitlich betrachtet, d.h. nicht als Modul. Daher wird auch kein Modulname vergeben, wodurch natürlich auch nicht exportiert bzw. importiert werden kann.

Die hier vorgeschlagenen Spracherweiterungen, wie auch Spracherweiterungen der Sprache Haskell$^+$ allgemein, können eine Änderung der abstrakten Syntax der Sprache Haskell$^+$ erfordern. Dies ist natürlich problematisch, da alle Programme, die auf dieser Schnittstelle (also auf $\underline{Spez_{SBE(L)} < AbstrakteSyntax_{Haskell^+} >}$) angewendet werden, an eine geänderte abstrakte Syntax korrekt angepaßt werden müßten, was schon einen gewissen Aufwand bedeutet. Aber auch wenn die abstrakte Syntax von Haskell$^+$ nicht verändert werden muß, sind Neu- bzw. Umprogrammierungen von bestimmten Teilen des Systems $\underline{Gen_{SBE}}$ erforderlich, damit die Spracherweiterungen auch korrekt und sinnvoll behandelt werden können.

Anhang A

Installation und Bedienung des Systems $\underline{Gen_{SBE}}$

A.1 Installation

Die Diskette, welche dieser Arbeit beigefügt ist, enthält alle Quelldateien (dokumentiert) für das System $\underline{Gen_{SBE}}$. Dazu gehören Spezifikationen für Scanner und Parser, Haskell- und Java-Quelltexte. Außerdem befindet sich auf der Diskette die $\underline{Spez_{SBE(P)}}$, also die Spezifikation des Editors für die in Kapitel 3 und 5 dieser Arbeit verwendete Beispielsprache $\underline{P}$. Da der Umfang der Quelltexte sehr groß ist (mehr als 100 Seiten), wird hier auf den Ausdruck der Listings verzichtet. Der an der konkreten Implementierung des Systems $\underline{Gen_{SBE}}$ interessierte Leser sei gebeten, sich die für ihn interessanten Teile der Programmierung selbst auszudrucken, wenn er sich ausführlich damit beschäftigen möchte.

Um das System $\underline{Gen_{SBE}}$ zur Erzeugung von sprachbasierten Editoren nutzen zu können, benötigen Sie einen auf ihrem Solaris-Rechner installierten Glasgow-Haskell-Compiler der Version 2.10 (oder einer höheren Version). Dann müssen der Scanner-Generator Alex (s. [Dor97]) und der Parser-Generator (s. [Mar97]) korrekt auf ihrem System eingerichtet sein. Weiterhin ist das Java-Development-Kit 1.2 (oder eine höhere Version) notwendig. Bei diesbezüglichen Unklarheiten fragen Sie bitte ihren Administrator.

Wenn diese Voraussetzungen gegeben sind, kopieren Sie bitte alle auf der Diskette abgespeicherten Dateien in ein eigenes Verzeichnis auf Ihrem Rechner. Starten Sie dann zur Installation des Systems $\underline{Gen_{SBE}}$ auf ihrem Rechnersystem das vorbereitete Shell-Skript `installGSBE`. Dadurch wird die Generierung der Scanner- und Parsermodule, die Übersetzung der Haskell-Module sowie die Compilierung der Java-Klassen ausgelöst; es erfolgen Hinweise auf die einzelnen Installationsschritte durch das Shell-Skript. Danach ist das System $\underline{Gen_{SBE}}$ ordnungsgemäß auf ihrem Rechner installiert und einsatzbereit.

A.2 Aufruf des Systems $\underline{Gen_{SBE}}$

Um einen sprachbasierten Editor für die Sprache $\underline{L}$ mit dem $\underline{Gen_{SBE}}$ erzeugen zu können, benötigen Sie natürlich zunächst einmal die Spezifikation für diesen Editor. Diese kann mit einem beliebigen Texteditor erstellt werden. Ist die $\underline{Spez_{SBE(L)}}$ vorhanden, geben Sie einfach folgendes ein:

`gsbe` *Editorname*

Dabei steht *Editorname* für den von Ihnen gewählten Namen für den zu erzeugenden $\underline{SBE(L)}$. Sie werden dann aufgefordert, den Namen (und den Pfad) der Datei anzugeben, in welcher sich die $\underline{Spez_{SBE(L)}}$ befindet. Nach Angabe der Datei wird der $\underline{SBE(L)}$ vom $\underline{Gen_{SBE}}$ generiert.

Um z.B. den in Kapitel 3 der Arbeit vorgestellten Editor $\underline{SBE(P)}$ zu erzeugen, rufen Sie einfach

```
gsbe P
```

auf und wenn sie nach der Spezifikationsdatei gefragt werden, geben Sie `p_spez.hp` ein (diese Datei ist auf ebenfalls auf der Diskette enthalten).

A.3 Aufruf eines erzeugten sprachbasierten Editors

Wenn Sie einen erzeugten $\underline{SBE(L)}$ starten wollen, geben Sie einfach folgendes ein:

`sbe` *Editorname*

Falls Sie z.B. den in Kapitel 3 der Arbeit vorgestellten Editor $\underline{SBE(P)}$ aufrufen wollen (und ihn vorher mit dem Namen P erzeugt haben), muß Ihre Eingabe

```
sbe P
```

lauten.

Anhang B

Legende

$\underline{Gen_{SBE}}$ - Generator für sprachbasierte Editoren

$\underline{Spez_{SBE(L)}}$ - Spezifikation eines sprachbasierten Editors für die Sprache $\underline{L}$

$\underline{SBE(L)}$ - sprachbasierter Editor für die Sprache $\underline{L}$

$\underline{UC_{AW}}$ - Universeller Code für die Auswerterkomponente

$\underline{UC_{NS}}$ - Universeller Code für die Nutzerschnittstellenkomponente

$\underline{EK_L}$ - Editiertes Konstrukt der Sprache $\underline{L}$

$\underline{EK_L[Init]}$ - initiales editiertes Konstrukt der Sprache $\underline{L}$

$Editbefehl_L^+$ bedeutet 'Folge von Editierbefehlen' bzgl. der Sprache $\underline{L}$

$< Y_{(L)} >$ bedeutet 'spezifiziert in/als Y (der Sprache $\underline{L}$)'

$\Longrightarrow$	Übersetzungs- bzw. Analysephase
A $\rightsquigarrow$ **B**	A programmiert / editiert B
$\longrightarrow$	Ein- bzw. Ausgabe
$\circlearrowright$	Schreiben und Lesen von Informationen in Dateien

I $\underline{Entwickler_{Gen_{SBE}}}$ - Entwickler des $\underline{Gen_{SBE}}$

II $\underline{Entwickler_{SBE(L)}}$ - Entwickler einer $\underline{Spez_{SBE(L)}}$

III $\underline{Nutzer_{SBE(L)}}$ - Nutzer des $\underline{SBE(L)}$

Literaturverzeichnis

[AJKV98] I. Adamski, B. Jokubeit, A. Kühnemann, and H. Vogler. *Incremental evaluators for ordered macro attribute grammars.* Technical Report TUD/FI98/10, Technical University of Dresden, 1998

[Dor97] C. Dornan. *Alex: A Lex for Haskell Programmers.* http://www.cs.ucc.ie/ dornan/ alex.html, 1997.

[DRT81] A. Demers, T. Reps, and T. Teitelbaum. *Incremental evaluation for attribute grammars with application to syntax-directed editors.* Conference Record of the 8th ACM Symposium on Principles of Programming Languages, pages 105-116, 1981.

[FHP97] J. H. Fasel, P. Hudak, and J. Peterson. *A gentle introduction to Haskell – Version 1.4 –.* Tutorial, Yale University and University of California, http://www.haskell.org/tutorial/, 1997.

[HP97] K. Hammond and J. Peterson (editors). *Report on the programming language Haskell — A non-strict purely functional language (Version 1.4)* Technical Report YALEU / DCS / RR-1106, Yale University, Department of Computer Science, 1997.

[Jok96] B. Jokubeit. *Konstruktion eines inkrementellen Auswerters für geordnete Makroattributgrammatiken* Master's thesis, TU Dresden, Fakultät Informatik, D-01062 Dresden, Germany, 1996.

[Kas80] U. Kastens. *Ordered attribute grammars.* Acta Informatica, 13:229-256,1980.

[Knu68] D. E. Knuth. *Semantics of context-free languages.* Math. Syst. Theory, 2:127-145, 1968. Corrections in Math. Syst. Theory, 5:95-96, 1971.

[KV94] A. Kühnemann und H. Vogler. *Synthesized and inherited functions — a new computational model for syntax-directed semantics.* Acta Informatica, 31:431-477, 1994.

[Les99] C. Lescher. *Entwurf und Implementierung einer Eingabesprache für ein System zur Erzeugung syntaxgesteuerter Editoren.* Großer Beleg, Technical University of Dresden, 1999.

[Mar97] S. Marlow. *Happy User Guide.* http://www.dcs.gla.ac.uk/fp/software/happy/, 1997.

[RT89] T. W. Reps and T. Teitelbaum. *The synthesizer generator — a system for constructing language-based editors.* Springer-Verlag, 1989.

[Sch98] H.-J. Schneider. *Lexikon der Informatik und Datenverarbeitung - 4., aktualisierte und erw. Aufl.* R. Oldenbourg Verlag, 1998.

Eigenständigkeitserklärung:

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig und nur unter Zuhilfenahme der im Literaturverzeichnis angegebenen Literatur angefertigt habe. Sie wurde von mir nicht in dieser oder einer ähnlichen Fassung an einer anderen Fakultät zur Erlangung eines akademischen Grades eingereicht.

Datum: Unterschrift: