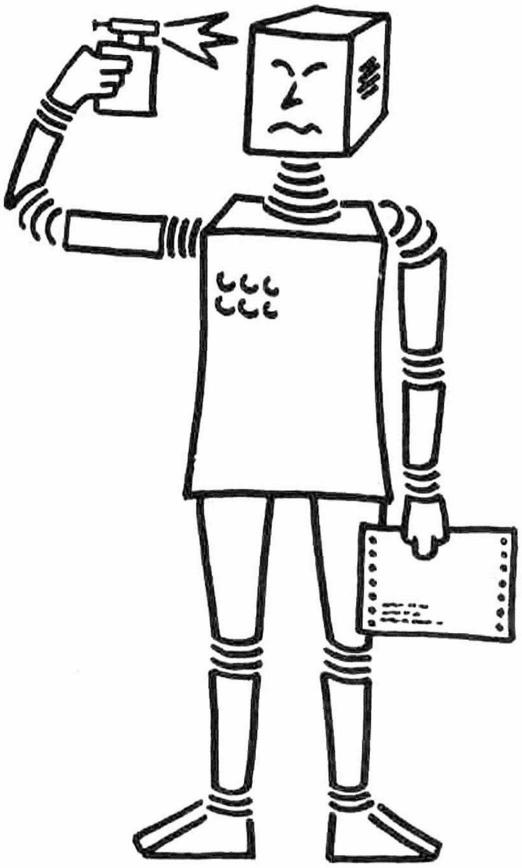
SEKI-PROJEKT

SEKI
MEMO

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



Spezifikation und
Abstrakte Implementierung des
Aufbereitungsteils von INTAKT

Wilfried Schrupp
Johann Tamme

Memo SEKI-83-01

SPEZIFIKATION UND ABSTRAKTE IMPLEMENTIERUNG

DES AUFBEREITUNGSTEILS

VON INTAKT

von

Wilfried Schrupp

Johann Tamme

Abstract

A worked example of a complete specification and abstract implementation of a sizable software system is given in terms of a predecessor of the software specification language ASPIK.

The specified software system INTAKT has been developed by Siemens AG (München). INTAKT is an interactive system for analyzing and upgrading programs written in a variety of programming languages. This paper only provides a specification and abstract implementation for part of the analysis support in INTAKT. The remainder is specified in a companion paper by H. Grieneisen. (Int. Bericht 83/83)

Zusammenfassung

Ein ausgearbeitetes Beispiel einer vollständigen Spezifikation und abstrakten Implementierung eines umfangreichen Softwaresystems wird angegeben, das in einem Vorläufer der Softwarespezifikationssprache ASPIK beschrieben ist.

Das spezifizierte Softwaresystem INTAKT wurde bei der Siemens AG (München) entwickelt. INTAKT ist ein Dialogsystem zur Analyse und Aufbereitung von Programmen verschiedener Programmiersprachen. Diese Arbeit beschreibt Spezifikation und abstrakte Implementierung eines Teils der Analyseunterstützung in INTAKT. Der restliche Teil ist spezifiziert in der Arbeit von H. Grieneisen. (Int. Bericht 83/83)

I N H A L T

0. Einleitung

Vergleicht man frühere Verfahren der Software-Entwicklung mit den in den letzten Jahren entwickelten Methoden, so stellt man einige elementare Unterschiede fest.
Spezifikationen im heutigen Sinne gab es nicht. Es wurden informelle Problembeschreibungen angefertigt, die naturgemäß interpretationsabhängig waren. Mit Hilfe von Datenflußanalysen, Struktogrammen, Flußdiagrammen u.ä. versuchte man von der informellen Ebene auf eine Abstraktionsebene zu gelangen. Die so erreichte Abstraktion war jedoch sehr maschinennah, d.h.: komplexe Probleme wurden auf einer niedrigen Ebene abstrahiert. Dieses hatte zur Folge, daß durch unwichtige Einzelheiten wie Variablen, Zuweisungen, Datenzugriffe u.ä., relevante Details unterdrückt oder verwischt wurden und komplexe Datenstrukturen durch vorgegebene einfache Datentypen beschrieben werden mußten, sodaß die Beschreibung unübersichtlich und schlecht nachvollzieh- und überprüfbar wurde. [Hrus 82]

Höchste Priorität galt der Schnelligkeit und dem Speicherbedarf der Software-Produkte, da die Leistungsfähigkeit der Rechner und die Speicherkapazitäten lange nicht so hoch und preiswert waren wie heute. Die Korrektheit der Programme, d.h. die Äquivalenz zwischen der informellen Anforderung an die Wirkung des Programms und der tatsächlichen Wirkung, versuchte man mit Hilfe von komplizierten Testprogrammen zu gewährleisten. Eine "gesicherte" Aussage, daß ein Software-Produkt das gestellte Problem auch wirklich implementiert, konnte nur mit Einschränkungen gemacht werden.

Durch die enormen Fortschritte der Hardware-Entwicklung - die Rechner arbeiten wesentlich schneller, die Speichermedien können mehr Daten auf weniger Platz aufnehmen, die Kosten für Leistung und Kapazität sind stark gesunken - und durch neue Einsatzgebiete, die eine sehr hohe Zuverlässigkeit erfordern wie Raumfahrt, Rüstungsprodukte, Atomkraftwerke, Verkehrssteuerung u.a. wird heute bei vielen Anwendungen die höchste Priorität der Korrektheit der Software-Produkte eingeräumt. Wegen den hohen Kosten bei der Wartung von Software ist es wirtschaftlicher, mehr Aufwand in die Erstellung zu investieren , als die Entwicklung von Software-Produkten möglichst schnell voranzutreiben und zu riskieren, daß das Endprodukt den gestellten Anforderungen nicht entspricht und deswegen zeitaufwendig geändert werden muß.

Dies alles erzwingt eine andere Vorgehensweise bei der Erstellung von Software.

Im Rahmen der Grundlagen-Forschung und der Entwicklung von neuen Software-Entwicklungsmethoden wurde das Projekt "Programmverifikation" unter der Leitung von Prof. Dr. P. Raulefs , Universität Bonn, und Dr. J. Siekmann, Universität Karlsruhe, unter Mitwirkung der Firma SIEMENS gestartet. Die vorliegende Arbeit ist in dieses Projekt eingebettet und wendet einen Teil der dort erzielten Ergebnisse an.

Allgemein kann man das Vorgehen bei der Erstellung eines Software-Produktes unter Anwendung der u.a. hier entwickelten Methoden folgendermaßen beschreiben :
Die informelle Problembeschreibung wird in eine eindeutige, abstrakte Beschreibung auf einer hohen Abstraktionsebene überführt, um evtl. folgende Implementierungsschritte als korrekt beweisen zu können. Dazu bedient man sich neuerer Spezifikationssprachen bzw. -methoden, wie z.B. TRIPLEX oder HDM, die mit Termsprachen, Algebren, Abbildungen u.ä. arbeiten und durch abstrakte Datentypen bzw. abstrakte Maschinen jedes Problem spezifizierbar machen und zwar völlig unabhängig von einer späteren Implementierung in einer Programmiersprache oder auf einer konkreten Maschine.
Die Äquivalenz einer so erstellten Top-Level-Spezifikation zur informellen Problembeschreibung läßt sich intuitiv sicherer zeigen als die Äquivalenz zwischen informeller Beschreibung und Programmtext, wie es die früheren Methoden erfordert haben.
Die Implementierung wird schrittweise durchgeführt. Die in einem Schritt spezifizierten abstrakten Datentypen werden durch andere, auf tieferer abstrakter Ebene spezifizierten, abstrakten Datentypen implementiert. Ein Implementierungsschritt wird durch eine Implementierungs- oder eine Repräsentationsfunktion spezifiziert und bzgl. dieser Funktion als korrekt bewiesen; es wird somit bewiesen, daß die implementierenden abstrakten Datentypen die Anforderungen, die an die zu implementierenden abstrakten Datentypen gestellt sind, erfüllen.
Die verschiedenen Implementierungsschritte sind so angelegt, daß ein Abstieg von der abstrakten Ebene hin zur endgültigen Implementierungsebene (der konkreten Ebene) erfolgt und evtl. auf der abstrakten Ebene spezifizierte "ineffiziente" Operationen in "bessere" bzgl. der Implementierungsebene umgeändert werden.
Ist jeder Schritt als korrekt bewiesen, so ist sichergestellt , daß das erstellte Software-Produkt die Anforde-

rungen der Top-Level-Spezifikation erfüllt.

Die Aufgabe der vorliegenden Diplom-Arbeit besteht nun darin, die Anwendungsmöglichkeiten der Spezifikatiossprache TRIPLEX exemplarisch zu zeigen, einen Implementierungsschritt durchzuführen und diesen als korrekt zu beweisen.
Nach Einführung von grundlegenden Definitionen und der Festlegung der Notation wird eine kurze Darstellung der Programmiersprache SPL', in der die zu bearbeitenden Programme vorliegen, und der Spezifikationssprache TRIPLEX, die für die Spezifikation des Problems benutzt wird, gegeben. Dann folgt die informelle Problembeschreibung eines Teils des Software-Systems INTAKT und in Kapitel 5 die erste formale Spezifikation, die Top-Level-Spezifikation. Zum leichteren Verständnis werden in dem darauf folgenden Kapitel Erläuterungen zur Spezifikation und zu den einzelnen Datentypen gegeben. In Kapitel 7 wird der Implementierungsschritt spezifiziert und erläutert, der schließlich im vorletzten Kapitel als korrekt bewiesen wird. Ein kurzer Erfahrungsbericht und eine Ausschau auf eine mögliche weitere Entwicklung auf dem Gebiet des Software-Engineering schließt die Arbeit ab.

1. Definitionen und Notation

1.1. Definitionen

In diesem Kapitel werden grundlegende Begriffe definiert, die in den folgenden Kapiteln ohne weitere Reflektion verwendet werden.

Bemerkung :
- Die Definitionen und das Theorem dieses Kapitels sind orientiert an [Kreo 79].
- λ steht für die leere Folge von Sortennamen $s \in S$.

Definition 1 : **Signatur**

Eine Signatur ist ein Paar (S,Σ) **mit**
- S ist eine endliche Menge von Sorten
- $\Sigma := \{ \Sigma_{w,s} \mid w \in S^* , s \in S \}$
  ist eine Familie von paarweise disjunkten endlichen Mengen $\Sigma_{w,s}$

Definition 2 : **Operatoren**

Die Elemente $\sigma \in \Sigma_{w,s}$ (aus Definition 1) heißen Operatoren .
$w \in S^*$ legt den Argumentbereich,
$s \in S$ den Wertebereich des Operators **fest.**

Definition 3 : **(S,Σ)-Algebra A**

Sei (S,Σ) eine Signatur.
Eine (S,Σ)-Algebra A ist ein Tripel (D,K,F) **mit** :
- $D := \{ A_s \mid s \in S \}$
  ist die Menge der zu S gehörigen Datenmengen A_s (Carrier)
- $K := \{ \sigma_A \mid \sigma_A \in \Sigma_{\lambda,s} , s \in S \}$
  ist eine Menge von konstanten Funktionen (Konstruktoren) mit
  $\forall s \in S.\ \forall \sigma \in \Sigma_{\lambda,s}.: \quad \sigma_A : \{\lambda\} \longrightarrow A_s$
- $F := \{ \sigma_A \mid \sigma \in \Sigma_{w,s} , w \in S^* , s \in S \}$
  ist eine Menge von Funktionen (Operationen) **mit**
  $\forall n \in N.\ \forall s,s1,s2,\ldots,sn \in S.\ \forall \sigma \in \Sigma_{s1s2\ldots sn,s}.:$
  $\sigma_A : A_{s1} x A_{s2} x \ldots x A_{sn} \longrightarrow A_s$

Definition 4 : **Carrier** (Trägermenge)

Die Mengen A_s aus Definition 3 werden Carrier oder auch Trägermengen genannt.

Definition 5 : **Konstruktoren**

Die Elemente der Menge K aus Definition 3 werden Konstruktoren genannt.

Definition 6 : **Operationen**

Die Elemente der Menge F aus Definition 3 werden Operationen genannt.

Definition 7 : **(s,Σ)-Terme** $T_{\Sigma,s}$

Sei (S,Σ) eine Signatur.
Zu jeder Sorte $s \in S$ ist die Menge der (s,Σ)-Terme $T_{\Sigma,s}$ induktiv definiert durch :
(i) $\forall s \in S.: \quad \Sigma_{\lambda,s} \subseteq T_{\Sigma,s}$
(ii) $\forall \sigma \in \Sigma_{s1\ldots sn,s}.\ \forall i \in [1:n].\ ti \in T_{\Sigma,si}.:$
$\sigma(t1,\ldots,tn) \in T_{\Sigma,s}$

Definition 8 : **(S,Σ)-Termalgebra** T_Σ

Zu einer Signatur (S,Σ) ist die <u>(S,Σ)-Termalgebra</u> T_Σ gegeben durch :
(Index T steht stellvertretend für Index T_Σ)
i) die Datenmengen $T_{\Sigma,s}$ aller (s,Σ)-Terme $\forall s \in S$
ii) die konstanten Operationen $\sigma_T \in \Sigma_{\lambda,s}$ $\forall \sigma \in \Sigma_{\lambda,s} \wedge \forall s \in S$
iii) die Operationen $\sigma_T : T_{\Sigma,s_1} \times \ldots \times T_{\Sigma,s_n} \longrightarrow T_{\Sigma,s}$ mit
$\sigma_T(t1,\ldots,tn) := \sigma(t1,\ldots,tn)$
$\forall \sigma \in \Sigma_{s_1\ldots s_n,s} \wedge \forall i \in [1:n].\ ti \in T_{\Sigma,s_i}$

Definition 9 : **Herbrand-Universum** CTA

Eine (S,Σ)-Term-Algebra T_Σ wird auch <u>Herbrand-Universum</u> oder <u>frei generierte Termalgebra über Σ</u> oder <u>kanonische Termalgebra</u> (CTA) genannt.

Definition 10 : **(S,Σ)-Homomorphismus** h : A ⟶ A'

A und A' seien (S,Σ)-Algebren.
Eine Familie von Abbildungen
$h = \{ h_s \mid h_s : A_s \longrightarrow A'_s \}$
heißt <u>(S,Σ)-Homomorphismus</u> , wenn gilt :
(i) $h_s(\sigma_A) = \sigma_{A'}$ $\forall \sigma \in \Sigma_{\lambda,s} \wedge s \in S$
(ii) $h_s(\sigma_A(a1,\ldots,an)) = \sigma_{A'}(h_{s_1}(a1),\ldots,h_{s_n}(an))$
$\forall \sigma \in \Sigma_{s_1\ldots s_n,s} \wedge \forall i \in [1:n].\ ai \in A_{s_i}$

Definition 11 : **(S,Σ)-Isomorphismus**

Ein (S,Σ)-Homomorphismus h : A ⟶ A' heißt <u>(S,Σ)-Isomorphismus</u> , wenn für jede Sorte s∈S die Abbildung h_s bijektiv ist.
Zwei (S,Σ)-Algebren A und A' sind <u>isomorph</u> , wenn es einen (S,Σ)-Isomorphismus h : A ⟶ A' gibt.

Definition 12 : **Initiale (S,Σ)-Algebra** A^i

Eine (S,Σ)-Algebra A^i heißt <u>initial</u> , wenn für jede (S,Σ)-Algebra A genau ein (S,Σ)-Homomorphismus h : A^i ⟶ A existiert.

Definition 13 : **Variable , Belegung**

Sei (S,Σ) eine Signatur und A eine (S,Σ)-Algebra.
i) Eine Mengenfamilie $X = \{ X_s \mid s \in S \}$ heißt <u>Variablenmenge von S</u> .
Ein Element $x \in X_s$ heißt <u>Variable der Sorte s</u> .
ii) Eine Familie von Abbildungen (h : X ⟶ A)
$h = \{ h_s \mid h_s : X_s \longrightarrow A_s ,\ s \in S \}$
bzgl. einer (S,Σ)-Algebra A wird <u>Belegung der Variablen X in A</u> genannt.

Definition 14 : **Signatur mit Variablen** (S,Σ(X))

Eine <u>Signatur mit Variablen</u> (S,Σ(X)) ist eine Erweiterung einer Signatur (S,Σ) durch Hinzufügen der Variablen als konstante Operatoren (Konstruktoren) :
$\Sigma(X) := \{ \Sigma(X)_{w,s} \mid w \in S^* ,\ s \in S \}$
mit $\Sigma(X)_{\lambda,s} := \Sigma_{\lambda,s} \cup X_s$ $\forall s \in S$
$\Sigma(X)_{w,s} := \Sigma_{w,s}$ $\forall w \in S^* \wedge w \neq \lambda$

Definition 15 : **(S,Σ(X))-Termalgebra mit Variablen** $T_\Sigma(X)$

Eine (S,Σ(X))-Termalgebra $T_{\Sigma(X)}$ heißt <u>Σ -Termalgebra mit Variablen in X</u> , wenn die in Definition 14 eingeführten konstanten Operatoren für Variablen weggelassen und alle anderen Operatoren beibehalten werden.
Für die Trägermengen gilt :
$T_\Sigma(X)_s := T_{\Sigma(X),s}$ $\forall s \in S$

<u>**Theorem 1 :**</u>

Zu jeder (S,Σ)-Algebra A und jeder Belegung h : X ⟶ A existiert genau ein (S,Σ)-Homomorphismus h' : $T_\Sigma(X)$ ⟶ A
mit : $h'_s(x) = h_s(x)$ $\forall x \in X_s ,\ \forall s \in S$.

Definition 16 **Gleichung**

Sei (S,Σ) eine Signatur und $X = \{ X_s \mid s \in S \}$ ist eine Mengenfamilie von Variablen.
Ein Paar e=(L,R) mit :
L und R sind (S,Σ)-Terme der Sorte s **mit** Variablen in X
$(L,R \in T_\Sigma(X)_s \wedge s \in S)$
heißt <u>Gleichung der Sorte s</u> .

Definition 17 : Spezifikation $(S,\Sigma,\mathcal{E})$

Eine Spezifikation ist ein Tripel $(S,\Sigma,\mathcal{E})$ bestehend aus den Komponenten einer Signatur (S,Σ) und einer Mengenfamilie von Gleichungen $\mathcal{E} := \{ \mathcal{E}_s \mid s \in S \}$, wobei die Elemente von $\mathcal{E}_s$ Gleichungen der Sorte s sind.

Definition 18 : $(S,\Sigma,\mathcal{E})$-Algebra

i) eine (S,Σ)-Algebra A erfüllt die Gleichnug $e=(L,R)\in\mathcal{E}$,
   wenn für alle Belegungen $h : X \longrightarrow A$ gilt :
   $h'(L) = h'(R)$ (siehe Theorem 1) ;
   d.h. wenn die linke und die rechte Seite immer gleich interpretiert werden.
ii) Eine (S,Σ)-Algebra A heißt $(S,\Sigma,\mathcal{E})$-Algebra ,
   wenn A alle Gleichungen aus $\mathcal{E}$ erfüllt.

Definition 19 : abstrakter Datentyp ADT

Sei $(S,\Sigma,\mathcal{E})$ eine Spezifikation.
Dann heißt die Isomorphieklasse der initialen $(S,\Sigma,\mathcal{E})$-Algebren der durch $(S,\Sigma,\mathcal{E})$ spezifizierte abstrakte Datentyp ADT .

1.2. Notation

In diesem Kapitel werden Notationen eingeführt, die notwendig sind, um zwischen Schlüsselwörtern von "TRIPLEX", Operations- und Sortennamen der Spezifikationen und den zu bearbeitenden Zeichen und Zeichenfolgen unterscheiden zu können.

Beipiele :	Buchstabe	a	und	Operation	a
	Datentyp	Bool	und	Sorte	Bool

Die folgende Notation wird im weiteren mit Ausnahme der "formalen" Kapitel 5, 7.1., 7.2. und 8. und mit kleinen Abweichungen in 6.7.12.1 und 6.7.12.2. benutzt :

- Namen von Spezifikationen werden mit großen Buchstaben geschrieben z.B. BOOL , INCLAUF
- Namen von Operationen werden mit kleinen Buchstaben geschrieben und unterstrichen
  z.B. and , body
- Schlüsselwörter von "TRIPLEX" werden mit großen Buchstaben geschrieben und unterstrichen
  z.B. PUBLIC OPS , PROPERTIES
- Sortennamen werden mit kleinen Buchstaben geschrieben und in Quotes eingeschlossen
  z.B. 'nat' , 'qprog'
- sonstige Eigennamen werden mit großen Buchstaben geschrieben und in Anführungszeichen eingeschlossen
  z.B. "TRIPLEX" , "INTAKT"
- Erklärungen zu Namen von Datentypen, Operationen und Sorten werden in Klammern hinter dem Namen aufgeführt. Die einzelnen Buchstaben, die den jeweiligen Namen definieren, und Anfangsbuchstaben von Substantiven werden groß geschrieben :
  z.B. ZEICHEN (erlaubte ZEICHEN)
  INCLAUF (INCLude-AUFlösung)

2. Spezifikationssprache "TRIPLEX"

"TRIPLEX" ist eine an der Universität Bonn in den Jahren 1981/82 entwickelte formale Sprache zur Beschreibung von Systemen von algebraischen Spezifikationen.
In diesem Kapitel wird eine kurze Beschreibung von "TRIPLEX", eingeschränkt auf die einfachen Spezifikationen mit Bezug auf [Voß 81a] und [Voß 81b] gegeben, um das Verständnis für die in den weiteren Kapiteln gegebenen formalen Spezifikationen zu erleichtern. Weitere Angaben zu "TRIPLEX" findet man in [BGO 82].

Das auf der nächsten Seite abgebildete Schema gibt einen Überblick über einen mit "TRIPLEX" spezifizierten abstrakten Datentyp und gibt Hinweise auf die zugrundeliegenden mathematischen Objekte.

Es gibt drei verschieden Arten von Spezifikationen :

- einfache Spezifikationen (simple specification, SSPEC)
  Diese können nicht durch Übergabe von Parametern aktualisiert werden; es gibt keine Instanzen.
- parametrisierte Spezifikationen
  (parameterized specification, PSPEC)
  Die Instanzen ergeben sich aus der Ersetzung der formalen Parameter durch die Definitionen der aktuellen Parameter.
- Parameter (parameter, PARM)
  Diese sind erforderlich, um parametrisierte Datentypen instantiieren zu können, da die formalen Parameter als abstrakte Datentypen definiert sein müssen.
  Es ist damit möglich, verschiedene Datentypen mit der durch den entsprechenden parametrisierten Datentyp spezifizierten Struktur zu erzeugen.

Die Namen innerhalb eines Spezifikationssystems müssen disjunkt sein.

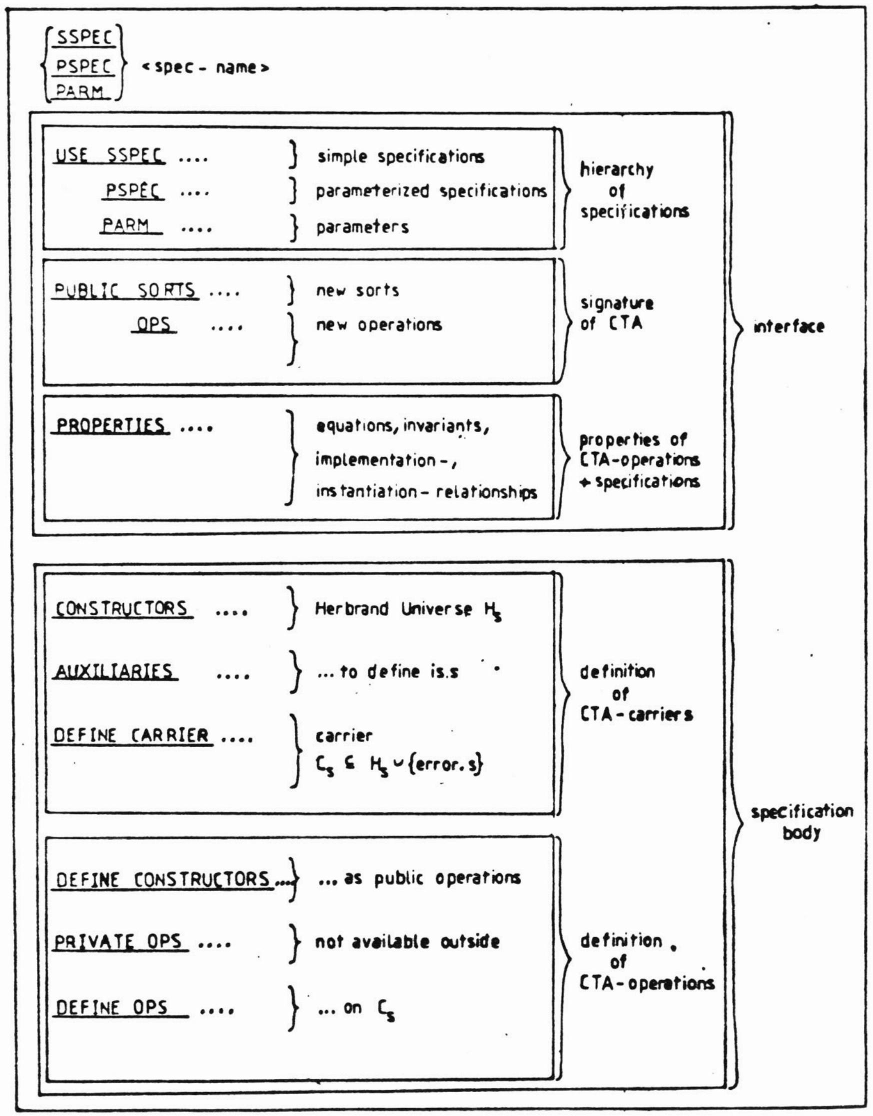


Abb.: Schema einer "TRIPLEX"-Spezifikation (entnommen aus [Bei 82a])

Jede Spezifikation besteht aus drei Hauptteilen :

- Art und Name der Spezifikation
- Spezifikations-Kopf (Interface)
- Spezifikations-Körper

Der Kopf einer Spezifikation, die im folgenden S genannt wird, stellt das Interface zu ihrer Umgebung dar.
Er beschreibt die nach außen hin sichtbaren Fähigkeiten durch Angabe ihres Namens, der benutzten Spezifikationen und der neu definierten Sorten und Operationen. Es wird definiert, was der abstrakte Datentyp leistet, und nicht, wie es realisiert ist.
Der Kopf definiert somit bzgl. einer Top-Level-Spezifikation auch die Schnittstelle zwischen informeller Beschreibung und abstrakter Spezifikation.

Der Spezifikations-Kopf besteht aus maximal drei Teilen :

- "USE-CLAUSE"
- "PUBLIC-CLAUSE"
- "PUBLIC-DESCRIPTIVE-CLAUSE"

Der "USE-CLAUSE" muß auf jeden Fall vorhanden sein, die anderen beiden Teile sind optional.

Die "USE-CLAUSE"s aller Spezifikationen eines Systems definieren eine Hierarchie auf den zugehörigen abstrakten Datentypen.
Das Downwards-Interface, also die Schnittstelle zu den abstrakten Datentypen, die in der Hierarchie tiefer als S stehen, kann durch Angabe eines "RESTRICT-CLAUSE" eingeschränkt werden. Die dort aufgeführten Operationen sind dann in S benutzbar, werden aber nicht von S exportiert.

Der "PUBLIC-CLAUSE" ist in zwei Bereiche aufgeteilt :

- PUBLIC SORTS

Hier sind die Namen der Sorten, die der Datentyp neu zur Verfügung stellt, aufgelistet.

- PUBLIC OPS

Hier sind die Namen der Operationen, die der Datentyp zur Verfügung stellt, zusammen mit ihren Definitions- und Wertebereichen aufgelistet.

Im "PUBLIC-CLAUSE" wird die zu S gehörende kanonische Term-Algebra (CTA) definiert.
Damit ist auch das Upwards-Interface, d.h. die Schnittstelle zu den abstrakten Datentypen, die in der durch die Use-Relationen festgelegten Hierarchie über S stehen, als Vereinigung des Downwards-Interface von S mit den PUBLIC SORTS und den PUBLIC OPS von S festgelegt.

Der "PUBLIC-DESCRIPTIVE-CLAUSE" beschreibt durch Angabe von PROPERTIES (Gleichungen, Invarianten, u.a.) Eigenschaften der Sorten und Operationen und damit das Verhalten der Spezifikation.
Falls hier alle Operationen von S vollständig definiert sind, liegt auch eine algebraische Definition der Operationen von S vor.

Der Körper einer Spezifikation S enthält die induktiven Definitionen der Carrier aller Sorten von S und die zu den im Kopf von S aufgeführten Operationen zugehörigen algorithmischen Definitionen. Außerdem können noch zusätzliche private Operationen eingeführt und definiert werden. Der Körper ist von außen nicht sichtbar.
Wenn nachgewiesen wird, daß alle angegebenen Algorithmen terminieren, dann ist durch "TRIPLEX" gewährleistet, daß eine korrekte Algebra spezifiziert ist.

Der Spezifikations-Körper besteht aus maximal zwei Teilen :

- "CARRIER-PART"
- "OPS-PART"

Auf jeden Fall existiert der "OPS-PART". Der "CARRIER PART" ist optional.

Der "CARRIER-PART" definiert für jede Sorte den Carrier der CTA, falls dieser nicht identisch zu dem durch die Konstruktoren aufgespannten Herbrand-Universum ist.
Die Elemente (Terme) des Carrier sind mit dem Präfix * gekennzeichnet.

Der "CARRIER-PART" ist in maximal drei Bereiche aufgeteilt :

- "CONSTRUCTOR-CLAUSE"
Hier werden die Köpfe der Operationen angegeben, die das Herbrand-Universum induzieren.

- "AUXILIARIES-CLAUSE"
Hier können Hilfsfunktionen (durch ein Präfix $ gekennzeichnet) definiert werden, die zur leichteren Definition des "ACCEPT-CLAUSE" dienen.
Es gibt vordefinierte Hilfsfunktionen zur Überprüfung der Sortenzugehörigkeit ($eq.s, für jede Sorte s), zur Bestimmung des äußersten Konstruktors eines Terms ($is_*opid, für jeden Konstruktor opid) und zur Selektion von Subtermen ($arg_i_*opid, für jeden Konstruktor opid).

- "ACCEPT-CLAUSE"
Hier wird zu jeder Sorte s, deren Carrier nicht identisch zu dem durch die Konstruktoren der Sorte s aufgespannten Herbrand-Universum ist, ein Prädikat angegeben, das für jedes Element des zu s gehörenden Herbrand-Universums entscheidet, ob es zu dem gewünschten Carrier gehört.

Der "OPS-PART" definiert die CTA-Operationen. Er besteht ebenfalls aus maximal drei Teilen :

- <u>DEFINE CONSTRUCTORS</u>
Hier wird für jeden Konstruktor, der eine Sorte definiert, deren Carrier von dem Herbrand-Universum abweicht, ein Algorithmus angegeben, der die Menge der erzeugbaren Terme auf den Carrier einschränkt.

- <u>PRIVATE OPS</u>
Hier können private Operationen, die man zur Definition der öffentlichen Operationen benutzen will, durch die Angabe ihrer Namen, ihrer Definitionsbereiche und ihrer Wertebereiche eingeführt werden.

- <u>DEFINE OPS</u>
Hier werden alle öffentlichen und privaten Operationen durch Angabe von Algorithmen definiert. Die Algorithmen sind über den entsprechenden Carriern definiert.

3. Programmiersprache "SPL'"

"SPL3" ist eine von "SIEMENS" entwickelte Sprache, die zum Austesten von interaktiven Software-Systemen benutzt wird. In dieser Arbeit wird vorausgesetzt, daß Programme, die von dem in Kapitel 5 spezifizierten Aufbereitungsteil des Software-Systems "INTAKT" bearbeitet werden sollen, in der von "SPL3" abgeleiteten Sprache "SPL'" geschrieben sind.

Die für diese Arbeit relevanten Teile von "SPL'" werden in diesem Kapitel definiert. Alle hier nicht explizit definierten syntaktischen Einheiten sind identisch der "SPL3"-Syntax.

<u>Syntax von "SPL'"</u>

```
<alpha>      :: a|b|c|d|e|f|g|h|i|j|k|l|m|n
                o|p|q|r|s|t|u|v|w|x|y|z

<num>        :: 0|1|2|3|4|5|6|7|8|9

<alpha-num>  :: <alpha>|<num>

<blank>      :: ␣

<quote>      :: '

<star>       :: *

<slash>      :: /

<del-basis>  :: ,|.|;|:|_|+|?|%|!|"|=|(|)|<|>|

<del-zkk>    :: <del-basis>|<star>|<slash>

<del-com>    :: <del-basis>|<star>|<quote>

<id>         :: <alpha-num>[<id>]

<comp-expr>  :: %eject|%expand|%noexpand|%etpnd

<incl-expr>  :: %include<blank><id>
                [(<alpha>=<zkk>[,<alpha>=<zkk>]*)]
```



```
<string>     :: <id>[<string>]|<del-zkk>[string]

<zkk>        :: '<string>'|'<string>'<zkk>|''<zkk>|''

<comment>    :: <id>[<comment>]|<blank>[<comment>]|
                <del-basis>[<comment>]|<quote>[<comment>]|
                <slash>[<comment>]|<star><id>[<comment>]|
                <star><del-com>[<comment>]

<com-expr>   :: /*<comment>*/|/**/
```

4.Software-System "INTAKT"

"INTAKT", ein Sofware-System von der SIEMENS-Arbeitsgruppe SW-Spezifikationsmethoden/Test- und Qualitätskontrollsysteme, hatte ursprünglich zwei Hauptziele :

- neuere Ergebnisse des Software-Engineering sollten erprobt und für die industrielle Praxis vorbereitet werden
- das Funktionsangebot an Test- und Qualitätskontrollsystemen für Software-Produkte sollte wesentlich erweitert und vereinheitlicht werden.

Das alles sollte bei größtmöglicher Sprachunabhängigkeit des Systems erreicht werden. Man bediente bzw. bedient sich deshalb der Spezifikationsmethode "HDM" (Hierarchical Development Methodology) , [SRL 79].

Im Rahmen der Zusammenarbeit von SIEMENS und der Universität Bonn (Institut für Informatik) wurde noch ein drittes Hauptziel hinzugefügt :

- ein Vergleich von Software-Entwicklungssystemen.

Hierbei sollen insbesondere die Methoden der Spezifikation mit abstrakten Maschinen ("HDM") und mit algebraischen abstrakten Datentypen ("TRIPLEX") miteinander verglichen und bewertet werden.

Das System "INTAKT" gliedert sich in drei Hauptteile :

- Dialog : Kommunikation zwischen Benutzer und System
- Analyse : syntaktische Analyse des eingegebenen Quellprogramms
- Auswertung : Aussagen bzgl. des analysierten Quellprogramms

In der vorliegende Diplomarbeit werden der vollständige Aufbereitungsteil (Teil der Analyse) und einige statistische Aussagen (Teil der Auswertung) von "INTAKT" für Anwendungen auf Quellprogramme in "SPL'" spezifiziert. Außerdem wird ein Implementierungsschritt durchgeführt, der von der sehr hohen abstrakten Ebene der Top-Level-

Spezifikation zu einer niedrigeren Ebene führt.
Es wird somit ein Beitrag zum Erreichen des dritten Hauptziels geliefert.

In diesem Kapitel wird die informelle Problembeschreibung, die der Arbeit zugrunde liegt [Sch 81a], gegeben.

4.1. Aufbereitungsteil

Der Aufbereitungsteil von "INTAKT" hat die Aufgabe, ein eingegebenes Quellprogramm so zu verarbeiten, daß die weitere Bearbeitung des Programms vereinfacht wird, aber keine Information verloren geht, die für die weitere Ausführung und/oder später erwünschte Aussagen relevant ist.

Als Eingabe erhält der Aufbereitungsteil :

- ein syntaktisch korrektes Quellprogramm P (eine Folge von "SPL'"-Zeichen mit Zeilenende-Markierungen ∘1)
- eine Include-Bibliothek

Als Ausgabe liefert der Aufbereitungsteil ein Programm P' (eine Folge von "SPL'"-Zeichen mit Zeilenende-Markierungen und weiteren Sonderzeichen) für das gilt :

i) alle Kommentare sind entfernt, Kommentar-Anfang und Kommentar-Ende sind markiert
ii) alle redundanten Blanks sind entfernt
iii) alle nicht zu interpretierenden Zeichen (Compile-Anweisungen) sind entfernt
iv) alle Include-Aufrufe sind aufbereitet und aufgelöst
v) die Aussagen von i) - iv) gelten nicht für Zeichenfolgen, die innerhalb von Zeichenkettenkonstanten stehen. Zeichenkettenkonstanten bleiben immer erhalten, wenn sie keine Parameterwerte darstellen.

zu i) <u>Kommentare</u>

Kommentare sind eingeschlossen von "/*" am Anfang und "*/" am Ende. Innere "/*" kennzeichnen keinen neuen Kommentaranfang.
"/*" innerhalb von Zeichenkettenkonstanten wird nicht als Kommentaranfang interpretiert.

Beispiele :

```
/*kommentar*/          ist der Kommentar "kommentar"
/*xyz/*abcd*/          ist der Kommentar "xyz/*abcd"
'/*abcdefg*/'          ist kein Kommentar
```

Nach der Aufbereitung sind sämtliche Kommentare entfernt. An ihrer Stelle stehen nun die Sonderzeichen

∘2 für den Kommentaranfang
∘3 für das Kommentarende .

Zeilenendemarkierungen, die innerhalb eines Kommentars stehen, bleiben erhalten .

Beispiele :

```
.../*kommentar*/...     geht über in     ...∘2∘3...
.../*xyz/*abcd*/...     geht über in     ...∘2∘3...
.../*komme∘1da*/...     geht über in     ...∘2∘1∘3...
```

zu ii) redundante Blanks

Ein Blank ist redundant, wenn

- links oder rechts von ihm ein Sonderzeichen steht, z.B. Delimiter oder internes Sonderzeichen
- zwei Blanks nebeneinander stehen

Redundante Blanks werden bei der Aufbereitung ersatzlos entfernt.
Blanks werden im weiteren Text durch ␣ dargestellt.

Beispiele :

```
ab␣c␣␣d        geht über in      ab␣c␣d
a␣;␣␣b*␣)      geht über in      a;b*)
'a␣b␣␣c'       geht über in      'a␣b␣␣c'
```

zu iii) Compile-Anweisungen

Folgende Compile-Anweisungen sind in "SPL'" erlaubt :

```
%eject
%expand
%noexpand
%etpnd
```

Bei der Aufbereitung werden alle Compile-Anweisungen ersatzlos entfernt. Compile-Anweisungen abschließende Sonderzeichen (";") bleiben erhalten, um später die genaue Anzahl der Statements bestimmen zu können.

Beispiel :

```
i=1;%eject;i=2;     geht über in     i=1;;i=2;
```


zu iv) Include-Aufrufe

Durch einen Include-Aufruf kann zusätzlicher Quellcode aus einer speziellen Include-Bibliothek, in der die sogenannten Include-Member zur Verfügung stehen, in das Quellprogramm übernommen werden.
Der aufbereitete Include-Member wird hinter dem aufbereiteten Include-Aufruf eingefügt.

In der Include-Bibliothek sind Include-Member mit Parametern durch ein Prozent-Zeichen als Präfix gekennzeichnet.

Beispiel : %mbname(default-parameter-liste)body

Include-Member ohne Parameter müssen kein % als Präfix haben. Es können also z.B. die beiden folgenden Fälle auftreten :

```
%mbname body
 mbname body .
```

Include-Member sind charakterisiert durch ihren Namen und evtl. vorhandene Parameter. Daher sind folgende Möglichkeiten eines Include-Aufrufs zu unterscheiden :

- Aufruf ohne Parameter :
  Die in dem Include-Member-Head angegebenen Default-Werte werden im Include-Body anstelle der formalen Parameter eingesetzt.
- Aufruf mit Parameter :
  Angegebene aktuelle Parameter-Werte werden für die entsprechenden formalen Parameter im Include-Member-Body eingesetzt.
  Parameter ohne angegebene aktuellen Werte werden wie oben behandelt.

Bei der Aufbereitung des Include-Aufrufs und dem Einsetzen des Include- Members in das Quellprogramm werden die folgenden internen Sonderzeichen als Markierungen gesetzt :

°4 : Include-Aufruf-Anfang
anstelle des %include
°8 : Parameter-Kennzeichnung
vor jedem Parameternamen der aktuellen Parameterliste
°9 : String-Kennzeichnung
vor jedem Parameterwert der aktuellen Parameterliste
°5 : Include-Aufruf-Ende
hinter der Parameterliste
°6 : Include-Member-Anfang
vor dem eingesetzten aufgelösten Include-Member
°7 : Include-Member-Ende
hinter dem eingesetzten aufgelösten Include-Member

Ausführliche Beispiele sind in der formalen Spezifikation INCLAUF unter <u>PROPERTIES</u> angegeben.

<u>Bemerkung</u>

Parameternamen in Include-Membern bestehen immer nur aus einem Buchstaben.
Include-Aufrufe innerhalb von eingesetzten Include-Membern werden ebenfalls aufbereitet und aufgelöst.

4.2. Statistische Aussagen

Es sollen die folgenden statistischen Ausssagen gemacht werden :

i) Anzahl der Kommentare

ii) Anzahl der Kommentarzeilen

Summe aller Zeilen, die nur bestehen aus :

- Kommentar(en) oder Kommentarteilen
- Kommentar(en) oder Kommentarteilen und Blanks

iii) Anzahl der Leerzeilen

iv) Klammerausdruck, der die Include-Aufruf-Hierarchie wiedergibt

5. Formale Top-Level-Speztifikation

In diesem Kapitel werden alle zur Spezifizierung des in Kapitel 4 informell beschriebenen Sofware-Systems benötigten abstrakten Datentypen formal in "TRIPLEX" spezifiziert.

In Kapitel 6 werden Erläuterungen sowohl zum Aufbau des gesamten Spezifikations-Systems alsauch zu den einzelnen Datentypen gegeben.
Sie ergänzen die als Kommentare in den Spezifikationen angegebenen Beschreibungen der Operationen.

<u>Bemerkung</u>

Die hier vorliegende formale Spezifikation wurde zu einem Zeitpunkt erstellt, als die Syntax von TRIPLEX noch nicht vollständig festgelegt war (Grundlage ist [Voß 81a] und [Voß 81b]).
Aus diesem Grund gibt es einige rein syntaktische Abweichungen unserer formalen Spezifikation von der jetzt bestehenden Version von "TRIPLEX", so sind hier die Argumente einer Operation durch ein Blank getrennt, in der neuesten Version von "TRIPLEX" werden die Argumente durch Kommata getrennt.

5.1. Boolesche Algebra

```
SSPEC   B O O L

PUBLIC SORTS     bool

          OPS    true     :  —> bool
                 false    :  —> bool
                 not      :  bool —> bool
                 and, or  :  bool bool —> bool
                 exor     :  bool bool —> bool

PROPERTIES

not ( true ) == false
not ( false ) == true
not ( not ( vb )) == vb
or ( b1 true ) == true
or ( b1 false ) == false
or ( b1 b2 ) == or ( b2 b1 )
and ( b1 true ) == b1
and ( b1 false ) == false
and ( b1 b2 ) == and ( b2 b1 )
not ( and ( b1 b2 )) ==  or ( not ( b1 ) not ( b2 ))

and ( b1 or ( b2 b3 ))  ==
    or ( and ( b1 b2 ) and ( b1 b3 ))

and ( or ( b1 b2 ) b3 ) ==
    or ( and ( b1 b3 ) and ( b2 b3 ))

exor ( b1 b2 ) ==
not ( or ( and ( b1 b2 ) and ( not ( b1 ) not ( b2 ))))
```

```
CONSTRUCTORS    *true
                *false


DEFINE OPS

not ( vb ) := case vb is *true  :  false
                         *false :  true
              esac

and ( b1 b2 ) := case b1 is *true  :  b2
                            *false :  false
                 esac

or ( b1 b2 ) := case b1 is *true  :  true
                           *false :  b2
                esac

exor ( b1 b2 ) := case b1 is *true  :  not ( b2 )
                             *false :  b2
                  esac


ENDSPEC
```


5.2. Natürliche Zahlen

```
SSPEC   N A T


USE      SSPEC BOOL


PUBLIC SORTS    nat

        OPS             null :  ―> nat
                 pred, succ :  nat ―> nat
                   add, sub :  nat nat ―> nat
                     lt, zq :  nat nat ―> bool


PROPERTIES

na > null   ⇒
succ ( pred ( na )) == pred ( succ ( na )) == na

n1 > null   ⇒
pred ( add ( n1 n2 )) == add ( pred ( n1 ) n2 )

n1 ≥ n2    ⇒
add ( sub ( n1 n2 ) n3 ) == sub ( add ( n1 n3 ) n2 )

n1 ≥ add ( n2 n3 )    ⇒
sub ( sub ( n1 n2 ) n3 ) == sub ( n1 add ( n2 n3 ))

zq ( n1 n1 ) == true
zq ( n1 n2 ) == zq ( n2 n1 )
zq ( succ ( n1 ) succ ( n2 )) == zq ( n1 n2 )
and ( zq ( n1 n2 ) zq ( n2 n3 )) ⇒ zq ( n1 n3 )
```

```
lt ( na na ) == false
lt ( na succ ( na )) == true
lt ( succ ( n1 ) succ ( n2 )) == lt ( n1 n2 )

lt ( n1 n2 )  ⇒  not ( lt ( n2 n1 ))
lt ( n1 n2 )  ⇒  lt ( n1 succ ( n2 ))
```

<u>CONSTRUCTORS</u>

```
*null
*succ
```

<u>DEFINE OPS</u>

```
pred ( na ) :=
case na is *null :  error.nat
           *succ ( na' ) :  na'
esac

add ( n1 n2 ) :=
case n1 is *null :  n2
           *succ ( na' ) : succ ( add ( na' n2 ))
esac

sub ( n1 n2 ) :=
case n2 is *null :  n1
           *succ ( n2' ) :
     case n1 is *null :  error.nat
                *succ ( n1' ) :  sub ( n1' n2' )
     esac
esac
```



```
zq ( n1 n2 ) :=
case n1 is *null
        case n2 is *null :  true
                otherwise   false
        esac
        otherwise
        case n2 is *null :  false
                otherwise
                zq ( pred ( n1 ) pred ( n2 ))
        esac
esac

lt ( n1 n2 ) :=
case n2 is *null : false
        otherwise
        case n1 is *null : true
                otherwise
                lt ( pred ( n1 ) pred ( n2 ))
        esac
esac
```

<u>ENDSPEC</u>

5.3. Erlaubte Zeichen

SSPEC Z E I C H E N

```
USE           SSPEC BOOL

PUBLIC SORTS  zeichen

      OPS     a, b, c, d, e :   ―> zeichen
              f, g, h, i, j :   ―> zeichen
              k, l, m, n, o :   ―> zeichen
              p, q, r, s, t :   ―> zeichen
              u, v, w, x, y :   ―> zeichen
              z, 1, 2, 3, 4 :   ―> zeichen
              5, 6, 7, 8, 9 :   ―> zeichen
              0, ∘1, ∘2, ∘3 :   ―> zeichen
              ∘4, ∘5, ∘6    :   ―> zeichen
              ∘7, ∘8, ∘9, ␣ :   ―> zeichen
              ,, ., ;, :, - :   ―> zeichen
              +, *, ?, /, % :   ―> zeichen
              !, ", ', =    :   ―> zeichen
              (, ), <, >    :   ―> zeichen
              sonz :  zeichen   ―> bool
              eq   :  zeichen zeichen  ―> bool

PROPERTIES

eq ( z1 z1 ) == true
eq ( z1 z2 ) == eq ( z2 z1 )
and ( eq ( z1 z2 ) eq ( z2 z3 )) ⇒ eq ( z1 z3 )
```



```
/*
Die ∘1 bis ∘9 sind interne Sonderzeichen mit folgender
Bedeutung :
∘1 ==  Zeilenende- Zeichen
∘2 ==  Kommentar- Anfang- Zeichen
∘3 ==  Kommentar- Ende- Zeichen
∘4 ==  Includeaufruf- Anfang- Zeichen
∘5 ==  Includeaufruf- Ende- Zeichen
∘6 ==  Includemember- Anfang- Zeichen
∘7 ==  Includemember- Ende- Zeichen
∘8 ==  Parameterkennzeichnung
∘9 ==  Stringkennzeichnung
und dürfen in dem ursprünglich zu bearbeitenden
Quellprogramm nicht vorhanden sein.
␣  ==  Sonderzeichen für Blank
*/

CONSTRUCTORS    *a
                .
                .
                *z
                *1
                .
                .
                *0
                *∘1
                .
                .
                *∘9
                *,
                .
                .
                *>
```

```
DEFINE OPS

sonz ( vz ) :=
case vz is *∘1 :  true
           .
           .
           *∘9 :  true
           *,  :  true
           .
           .
           *>  :  true
           otherwise  false
esac

eq ( z1 z2 ) :=
case z1 is *a
              case z2 is *a :  true
                             otherwise  false
              esac
           *b
              case z2 is *b :  true
                             otherwise  false
              esac
           .
           .
           .
           *>
              case z2 is *> :  true
                             otherwise  false
              esac
esac

ENDSPEC
```


5.4. Quellprogramm

```
SSPEC   Q U E L L P R O G

USE     SSPEC NAT  SSPEC ZEICHEN

PUBLIC SORTS   qprog

        OPS    nil          :  —> qprog
               hd, last     :  qprog —> zeichen
               tl, cut      :  qprog —> qprog
               laenge       :  qprog —> nat
               head, tail   :  qprog   nat    —> qprog
               cat          :  qprog   qprog  —> qprog
               cons         :  zeichen qprog  —> qprog
               mbre         :  zeichen qprog  —> bool
               subst        :  qprog qprog nat —> qprog
               cmpr         :  qprog qprog nat —> bool

PROPERTIES

hd ( cons ( vz qp )) == vz

tl ( cons ( vz qp )) == qp

laenge ( nil ) == null
laenge ( cons ( vz qp )) == succ ( laenge (qp ))
laenge ( tl ( cons ( vz qp )))  == laenge ( qp )
laenge ( cut ( cons ( vz qp ))) == laenge ( qp )

head ( qp null ) == nil
and ( na > null  na ≤ succ ( laenge ( qp )) )    ⇒
head ( cons ( vz qp ) na ) ==
                           cons ( vz head ( qp pred ( na )))
```

```
tail ( qp null ) == qp
tail ( tl ( qp ) na ) == tail ( qp succ ( na ))
na > null   ⇒
exor ( tail ( cons ( vz qp ) na ) == tail ( qp pred ( na ))
       tail ( nil na ) == tail ( nil pred ( na )))

cut ( nil ) == nil
cut ( cons ( vz qp ))  == head ( cons ( vz qp ) laenge ( qp ))

last ( cons ( vz qp )) == hd ( tail ( cons ( vz qp )
                                     laenge ( qp )))

subst ( q1 q2 null )          == cat ( q2 q1 )
subst ( q1 q2 na )            == cat ( q2 tail ( q1 na ))
subst ( q1 q2 laenge ( q1 )) == q2

cmpr ( q1 q2 null ) == true

or ( na > laenge ( q1 ) na > laenge ( q2 ))    ⇒
cmpr ( q1 q2 na )  == false

and ( na > null
      and ( na ≤ laenge ( q1 )  na ≤ laenge ( q2 )) )    ⇒
cmpr ( q1 q2 na ) == and ( eq ( hd ( q1 ) hd ( q2 ))
                           cmpr ( tl ( q1 ) tl ( q2 ) pred ( na )))

na ≤ laenge ( qp )   ⇒
cat ( head ( qp na ) tail ( qp na )) == qp
```



```
/*
cons ( vz qp ) ::   vz wird in qp vorne angefügt
hd ( qp )      ::   erstes Zeichen von qp
tl ( qp )      ::   qp ohne das erste Zeichen
last ( qp )    ::   das letzte Zeichen von qp
cut ( qp )     ::   qp ohne das letzte Zeichen
cat ( q1 q2 )  ::   Konkatenation von q1 und q2
mbre ( vz qp ) ::   ergibt true , wenn vz Element von qp ist
                          false, sonst
head ( qp na ) ::   Liste der ersten na Zeichen von qp
tail ( qp na ) ::   qp ohne die ersten na Zeichen
subst ( q1 q2 na ) ::  Ersetzung der ersten na Zeichen
                       von q1 durch q2
cmpr ( q1 q2 na )  ::  ergibt true, wenn die ersten na
                       Zeichen von q1 und q2 gleich sind
                       false, sonst
laenge ( qp )      ::  Anzahl der Elemente von qp
*/
```

<u>CONSTRUCTORS</u>

```
*nil
*cons
```

<u>DEFINE OPS</u>

```
hd ( qp ) :=
case qp is *nil :  error.zeichen
           *cons ( vz qp' ) :  vz
esac


tl ( qp ) :=
case qp is *nil :  nil
           *cons ( vz qp' ) :  qp'
esac


last ( qp ) :=
case qp is *nil :  error.zeichen
           *cons ( vz qp' ) :
                case qp' is *nil :  vz
                        otherwise  last ( qp' )
                esac
esac
```

```
cut ( qp ) :=
case qp is *nil :  nil
           *cons ( vz qp' ) :
                case qp' is *nil :  nil
                            otherwise  cons ( vz cut ( qp' ))
                esac
esac

cat ( q1 q2 ) :=
case q1 is *nil :  q2
           *cons ( vz q1' ) : cons ( vz cat ( q1' q2 ))
esac

mbre ( vz qp ) :=
case qp is *nil :  false
           *cons ( vr qp' ) :
                 if eq ( vr vz ) then true
                                 else mbre ( vz qp' )
                 fi
esac

head ( qp na ) :=
if zq ( na null )
   then nil
   elseif lt ( laenge ( qp ) na )
          then error.qprog
          else cons ( hd ( qp )
                      head ( tl ( qp ) pred ( na )))
fi

tail ( qp na ) :=
if zq ( na null )
   then qp
   else tail ( tl ( qp ) pred ( na ))
fi
```



```
subst ( q1 q2 na ) :=
if zq ( na null )
   then cat ( q2 q1 )
   else subst ( tl ( q1 ) q2 pred ( na ))
fi

cmpr ( q1 q2 na ) :=
if zq ( na null )
   then true
   elseif or ( zq ( laenge ( q1 ) null )
               zq ( laenge ( q2 ) null ))
        then false
        else and ( eq ( hd ( q1 ) hd ( q2 ))
                   cmpr ( tl ( q1 ) tl ( q2 ) pred ( na )))
fi

laenge ( qp ) :=
case qp is *nil :  null
           *cons ( vz  qp' ) :  succ ( laenge ( qp' ))
esac
```

<u>ENDSPEC</u>

5.5. Paar von Quellprogrammen

```
SSPEC   Q P A A R

USE     SSPEC QUELLPROG

PUBLIC SORTS    qpaar

        OPS     pair   :  qprog qprog ---> qpaar
                s1, s2 :  qpaar ---> qprog

PROPERTIES

s1 ( pair ( q1 q2 )) == q1
s2 ( pair ( q1 q2 )) == q2
pair ( s1 ( qp ) s2 ( qp ) ) == qp

CONSTRUCTORS    *pair

DEFINE OPS

s1 ( qp ) :=  case  qp is *pair ( q1 q2 ) : q1  esac
s2 ( qp ) :=  case  qp is *pair ( q1 q2 ) : q2  esac

ENDSPEC
```


5.6. Tripel von Quellprogrammen

```
SSPEC   Q T R I P E L

USE     SSPEC QUELLPROG

PUBLIC SORTS    qtripel

        OPS     trip :  qprog qprog qprog ---> qtripel
                sq1, sq2, sq3 :  qtripel  ---> qprog

PROPERTIES

sq1 ( trip ( q1 q2 q3 )) == q1
sq2 ( trip ( q1 q2 q3 )) == q2
sq3 ( trip ( q1 q2 q3 )) == q3
trip ( sq1 ( qp ) sq2 ( qp ) sq3 ( qp ) ) == qp

CONSTRUCTORS    *trip

DEFINE OPS

sq1 ( qp ) :=  case  qp is *trip ( q1 q2 q3 ) : q1  esac
sq2 ( qp ) :=  case  qp is *trip ( q1 q2 q3 ) : q2  esac
sq3 ( qp ) :=  case  qp is *trip ( q1 q2 q3 ) : q3  esac

ENDSPEC
```

5.7. Sonderbehandlung von Zeichenkettenkonstanten

SSPEC Z E I K E

```
USE        SSPEC QUELLPROG    SSPEC QPAAR


PUBLIC OPS       postz :  qprog ——> qpaar
                 zaehl :  qprog bool  ——> bool
                 srch  :  qprog qprog ——> qpaar
               s2srch  :  qprog qprog nat ——> qprog


PROPERTIES


zaehl ( nil false ) == false
zaehl ( qp false )  == not ( zaehl ( cons ( ' qp ) false ))
zaehl ( qp true  )  == not ( zaehl ( qp false ))

cat ( s1 ( srch ( q1 q2 )) s2 ( srch ( q1 q2 ))) == q1

laenge ( s2 ( srch ( cut ( s1 ( srch ( q1 q2 ))) q2 )))
                                                   == null

let T1 == s1 ( srch ( q1 q2 )) in
exor ( and ( mbre ( last ( T1 ) q2 )
            not ( mbre ( last ( T1 ) cut ( T1 ))))
        zq ( laenge ( s2 ( srch ( q1 q2 ))) null ))

mbre ( ' qp )    ⇒
cat ( s1 ( postz ( qp )) s2 ( postz ( qp )) ) == qp

mbre ( ' qp )    ⇒
eq ( hd ( s2 ( postz ( qp ))) ' ) == false

eq ( last ( cut ( s1 ( postz ( qp )))) ' )    ⇒
zaehl (cut ( s1 ( postz ( qp ))) false ) == true
```



```
let T2 == srch ( qp cons ( ' nil )) in
and ( mbre ( ' qp ) not ( zaehl ( qp false )) )    ⇒
cat ( s1 ( T2 ) cat ( s1 ( postz ( s2 ( T2 )))
                      s2 ( postz ( s2 ( T2 ))))) == qp

s2srch ( q1 q2 succ ( succ ( null ))) ==
         s2 ( srch ( s2 ( srch ( q1 q2 ) ) q2 ))

s2srch ( q1 q2 succ na ))) ==
s2srch ( s2 ( srch ( q1 q2 )) q2 na )


/*
srch ( q1 q2 ) :: sucht in q1 nach dem ersten Auftreten eines
                  Elementes von q2 und liefert als Ergebnis
                  ein Tupel mit
              1. Komponente : Programmtext bis einschließlich
                              des gefundenen Zeichens
              2. Komponente : Programmtext nach dem gefundenen
                              Zeichen

postz ( qp ) :: wird benutzt, um Zeichenketten-Konstanten zu
                lesen. Als Argument wird immer der Programm-
                text übergeben, der einem gefundenen Quote
                folgt. Als Ergebnis wird ein Tupel geliefert
                mit
              1. Komponente : Zeichenketten-Konstante ein-
                              schließlich des abschließenden
                              Quotes
              2. Komponente : Programmtext, der der Zeichen-
                              ketten-Konstanten folgt

zaehl ( qp vb ) :: "zaehlt" die Quotes in qp und liefert
                   true,  wenn die Anzahl der Quotes
                          ungerade ist
                   false, sonst
                   vb muß immer mit false initialisiert
                   werden
                   zaehl wird nur bei den Properties benutzt

s2srch ( q1 q2 na ) :: rekursives Suchen ( na-mal ) in
                       s2 ( srch ( q1 q2 ))
                       Ergebnis : zweite Komponente des
                                  letzten Suchens
*/
```

DEFINE OPS

```
srch ( q1 q2 ) :=
if zq ( laenge ( q1 ) null )
   then pair ( nil nil )
   elseif mbre ( hd ( q1 ) q2 )
          then pair ( cons ( hd ( q1 ) nil )
                        tl ( q1 ) )
          else pair ( cons ( hd ( q1 )
                               s1 ( srch ( tl ( q1 ) q2 )))
                        s2 ( srch ( tl ( q1 ) q2 )))
fi


postz ( qp ) :=
let  T1 == srch ( qp cons ( ' nil ))  in
if not ( eq ( last ( s1 ( T1 )) ' ))
   then error.qpaar
   elseif eq ( hd ( s2 ( T1 )) ' )
         then pair ( cat ( s1 ( T1 ) cons ( ' s1 (
                              postz ( tl ( s2 ( T1 ))))))
                       s2 ( postz ( tl ( s2 ( T1 )))))
         else pair ( s1 ( t1 ) s2 ( t1 ))
fi


zaehl ( qp vb ) :=
if zq ( laenge ( qp ) null )
   then vb
   elseif eq ( hd ( qp ) ' )
          then zaehl ( tl ( qp ) not ( vb ))
          else zaehl ( tl ( qp ) vb )
fi


s2srch ( q1 q2 na ) :=
if zq ( laenge ( q1 ) null )
   then nil
   elseif zq ( na succ ( null ))
          then s2 ( srch ( q1 q2 ))
          else s2 ( srch ( s2srch ( q1 q2 pred ( na )) q2 ))
fi
```

ENDSPEC

5.8. Ersetzung der Kommentare

SSPEC N O K O M M E N T A R

USE SSPEC ZEIKE

PUBLIC OPS kweg : qprog —> qprog

PROPERTIES

```
let T1 == srch ( kweg ( qp ) cons ( / nil )) in
eq ( hd ( s2 ( T1 )) * )  ⇒  zaehl ( s1 ( T1 ) false ) == true

and ( mbre ( ∘2 kweg ( qp ))
      not ( mbre ( vz cons ( ∘1 cons ( ∘3 nil )))))  ⇒
        not ( mbre ( vz s1 ( srch ( s2 ( srch ( kweg ( qp )
                                            cons ( ∘2 nil )))
                                  cons ( ∘3 nil )))))

/*
kweg(qp) :: qp' mit: qp' ist qp ohne Kommentare.
            Anfang und Ende der entfernten Kommentare
            sind mit ∘2 und ∘3 gekennzeichnet; in den
            Kommentaren enthaltene Zeilenende-Marken werden
            nicht gelöscht.
*/
```

```
PRIVATE OPS      kzeil :  qprog ---> qprog


/*
kzeil :: wird aufgerufen, wenn der Anfang eines Kommentars
         erkannt und mit dem Sonderzeichen °2 markiert worden
         ist.
         kzeil streicht den nun am Anfang stehenden Kommentar,
         behält jedoch evtl. Zeilenende-Marken bei und
         setzt am Kommentarende das Zeichen °3.
*/


PROPERTIES

or ( eq ( hd ( kzeil ( qp )) °1 )
     eq ( hd ( kzeil ( qp )) °3 )) == true


DEFINE OPS

kweg ( qp ) :=
let T1 == srch ( qp cons ( ' cons ( / nil )))
    T2 == postz ( s2 ( T1 ))
    T3 == last ( s1 ( T1 ))                       in

if zq ( laenge ( qp ) null )
   then nil
   elseif not ( mbre ( T3 cons ( ' cons ( / nil ))))
          then qp
          elseif eq ( T3 ' )
                 then cat ( cat ( s1 ( T1 )
                                  s1 ( T2 ))
                            kweg ( s2 ( T2 )))
                 elseif eq ( hd ( s2 ( T1 )) * )
                        then cat ( cut ( s1 ( T1 ))
                                   cons ( °2
                                     kweg ( kzeil (
                                            tl ( s2(
                                            T1 ))))))
                        else cat ( s1 ( T1 )
                                   kweg ( s2 ( T1 )))
fi
```



```
kzeil ( qp ) :=
let  T1 == srch ( qp cons ( °1 cons ( * nil ))) in

if zq ( laenge ( qp ) null )
   then error.qprog
   elseif eq ( last ( s1 ( T1 ) °1 ))
          then cons ( °1 kzeil ( s2 ( T1 )))
          elseif eq ( last ( s1 ( T1 )) * )
                 then if eq ( hd ( s2 ( T1 )) / )
                         then cons ( °3 tl ( s2 ( T1 )))
                         else kzeil ( s2 ( T1 ))
                      fi
                 else error.qprog
fi


ENDSPEC
```

5.9. Entfernung der redundanten Blanks

SSPEC N O R E D B L A N K

```
USE         SSPEC ZEIKE


PUBLIC OPS       bweg :  qprog ---> qprog


PROPERTIES

eq ( hd ( bweg ( qp )) ␣ ) == false

let T1 == srch ( bweg ( qp ) cons ( ␣ nil ))    in
eq ( hd ( s2 ( T1 )) ␣ )  ⇒  zaehl ( s1 ( T1 ) false )
sonz ( last ( cut ( s1 ( T1 )))) ⇒ zaehl ( s1 ( T1 ) false )
sonz ( hd ( s2 ( T1 ))) ⇒ zaehl ( s1 ( T1 ) false )


/*
bweg(qp) :: qp' mit: qp' ist identisch qp ohne
               redundante Blanks ( ␣ == Blank ).
*/


PRIVATE OPS      nob :  qprog ---> qprog


/*
nob :: wird von bweg aufgerufen, nachdem evtl. den Programm-
       text anführende Blanks entfernt worden sind.
       nob entfernt dann alle redundanten Blanks aus dem
       Programmtext.
*/
```



```
DEFINE OPS

bweg ( qp ) :=
if zq ( laenge ( qp ) null )
   then nil
   elseif eq ( hd ( qp ) ␣ )
          then bweg ( tl ( qp ))
          else nob ( qp )
fi

nob ( qp ) :=
let T1 == postz ( tl ( qp ))
    T2 == hd ( qp )
    T3 == tl ( qp )

if zq ( laenge ( qp ) null )
   then nil
   elseif eq ( hd ( T3 ␣ ))
          then if eq ( T2 ␣ )
                  then nob ( T3 )
                  elseif eq ( T2 ' )
                        then cat ( cons ( ' s1 ( T1 ))
                                   nob ( s2 ( T1 )))
                        elseif sonz ( T2 )
                            then cons ( T2 nob ( tl ( T3 )))
                            else cons ( T2 nob ( T3 ))
               fi
          elseif eq ( T2 ' )
                 then cat ( cons ( ' s1 ( T1 ))
                            nob ( s2 ( T1 )))
                 elseif eq ( T2 ␣ )
                        then if sonz ( hd ( T3 ))
                                then nob ( T3 )
                                else cons ( T2 nob ( T3 ))
                             fi
                        else cons ( T2 nob ( T3 ))
fi


ENDSPEC
```

5.10. Entfernung der Compile-Anweisungen

```
SSPEC   N O C O M P A N

USE     SSPEC ZEIKE

PUBLIC OPS      cweg :  qprog ---> qprog

PROPERTIES

let T1 == srch ( qp cons ( % nil ))
    T4 == cons ( e cons ( j cons ( e cons (
                 c cons ( t nil )))))
    T5 == cons ( e cons ( x cons ( p cons (
                 a cons ( n cons ( d nil ))))))
    T6 == cons ( n cons ( o T5 ))
    T7 == cons ( e cons ( t cons ( p cons (
                 n cons ( d nil )))))
    L4 == laenge ( T4 )
    L5 == laenge ( T5 )
    L6 == laenge ( T6 )
    L7 == laenge ( T7 )                         in

or ( or ( cmpr ( s2 ( T1 ) T4 L4 )
          cmpr ( s2 ( T1 ) T5 L5 ))
     or ( cmpr ( s2 ( T1 ) T6 L6 )
          cmpr ( s2 ( T1 ) T7 L7 )))  =>
and ( zaehl ( s1 ( T1 ) false ) zaehl ( s2 ( T1 ) false ))


/*
cweg(qp) :: liefert qp' mit: qp' ist identisch qp ohne die
            Compile- Anweisungen: %eject, %expand, %etpnd,
                                  %noexpand.
*/
```



```
DEFINE OPS

cweg ( qp ) :=
let T1 == srch ( qp cons ( ' cons ( % nil )))
    T2 == postz ( s2 ( T1 ))
    T4 == cons ( e cons ( j cons ( e cons (
                 c cons ( t nil )))))
    T5 == cons ( e cons ( x cons ( p cons (
                 a cons ( n cons ( d nil ))))))
    T6 == cons ( n cons ( o T5 ))
    T7 == cons ( e cons ( t cons ( p cons (
                 n cons ( d nil )))))
    L4 == laenge ( T4 )
    L5 == laenge ( T5 )
    L6 == laenge ( T6 )
    L7 == laenge ( T7 )                         in

if excr ( zq ( laenge ( qp ) null )
          and ( not ( eq ( last ( s1 ( T1 )) ' ))
                zq ( laenge ( s2 ( T2 )) null )))
   then qp
   elseif eq ( last ( s1 ( T1 )) ' )
          then cat ( cat ( s1 ( T1 ) s1 ( T2 ))
                     cweg ( s2 ( T2 )))
          elseif cmpr ( s2 ( T1 ) T4 L4 )
                 then cat ( cut ( s1 ( T1 ))
                            cweg ( tail ( s2 ( T1 ) L4 )))
                 elseif cmpr ( s2 ( T1 ) T5 L5 )
                        then cat ( cut ( s1 ( T1 ))
                                   cweg ( tail ( s2 ( T1 )
                                                 L5 )))
                        elseif cmpr ( s2 ( T1 ) T6 L6 )
                               then cat ( cut ( s1 ( T1 ))
                                          cweg ( tail (
                                                 s2 ( T1 )
                                                 L6 )))
                               elseif cmpr ( s2( T1 ) T7 L7 )
                                      then cat ( cut ( s1 ( T1 ))
                                                 cweg ( tail (
                                                        s2 ( T1 )
                                                        L7 )))
                                      else cat ( s1 ( T1 )
                                                 cweg ( s2 ( T1 )))
fi

ENDSPEC
```

5.11. Bereinigung des Quellprogramms

```
SSPEC   Q P S A U B E R

USE     SSPEC NOKOMMENTAR  SSPEC NOREDBLANK
        SSPEC NOCOMPAN

PUBLIC OPS      rein :  qprog —> qprog

DEFINE OPS

rein ( qp ) := cweg ( kweg ( bweg ( qp )))

ENDSPEC
```


5.12. Include-Auflösung

```
SSPEC   I N C L A U F

USE     SSPEC QPSAUBER  SSPEC QTRIPEL

PUBLIC OPS      aufl :  qprog —> qprog
                incn :  qprog —> qprog

/*
In den nun folgenden Properties wird die Operation aufl opera-
tionell erklärt.
Aus Gründen der besseren Lesbarkeit wird bei Angaben von
Quellcode in SPL' das Blankzeichen ( ␣ ) nicht geschrieben.
z.B.: k␣=␣2␣; entspricht: k = 2 ;
*/

PROPERTIES

let T1 == cons ( i cons ( n cons ( c cons ( l cons ( u
                cons ( d cons ( e nil )))))))                  in

cmpr ( s2 ( srch ( aufl ( qp ) cons ( % nil )))
        T1 laenge ( T1 ))  ⇒  zaehl ( s1 ( T1 ) false )

lmbr ( m1 ) == %m1 ( va = 'i=1;' )
                l = 1; %va; l = 2;
⇒  aufl ( k = 1;' %include m1 '; k = 2; ) ==
   k=1;' %include m1 ';k=2;

lmbr ( m1 ) == %m1 ( va = 'i=1;' )
                l = 1; %va; l = 2;
⇒  aufl ( k = 1; %include m1; k = 2; ) ==
   k=1;∘4m1∘5∘6l=1;i=1;l=2;∘7k=2;
```

```
lmbr ( m1 ) == %m1 ( va = 'i=1;' )
                 l = 1; %va; l = 2;
⇒  aufl ( k = 1; %include m1 ( va = 'n = 3;' );
              k = 2; ) ==
    k=1;∘4m1∘8va∘9n=3∘5∘6l=1;n=3;l=2;∘7k=2;

lmbr ( m1 ) == %m1 ( va = 'i=1;' )
                 l = 1; %va; l = 2;
⇒  aufl ( k = 1;  %include m1; k = 2;
              %include m1 ( va = 't = 5;' ) ; k = 3; ) ==
    cat ( aufl ( k = 1; %include m1; )
                aufl ( k = 2; %include m1 ( va = 't = 5; ' )
                       k = 3; ) )

and ( lmbr ( m1 ) == %m1 ( va = 'i=1;' )
                       l = 1; %va; l = 2;
      lmbr ( m2 ) == %m2 ( vb = 's = 7;' , vc = 't = 2;' );
                       z = 1; %vb; z = 2;  %include m1;
                       z = 3; %vc; z = 4; )
⇒  aufl ( k = 1; %include m2 ( vc = 'j = 5;' ); k = 2; )
              ==
   cat ( cat ( k=1;∘4m2∘8vc∘9j=5;∘5∘6z=1;s=7;z=2;
               aufl ( %include m1; ) )
          z=3;j=5;z=4;∘7k=2; )
```

```
/*
aufl ( qp ) :: liefert qp', in dem alle Include-Aufrufe
               aufgelöst sind,
               d.h. anstelle der Include-Aufrufe von qp
               stehen nun die entsprechenden aktualisierten
               Include-Member

incn ( qp ) :: liefert den Member-Namen eines Include-Aufrufs

lmbr ( m1 ) :: vorgegebene Operation, durch die das komplette
               Include-Member m1 aus der Bibliothek gelesen
               wird.
*/
```


PRIVATE OPS

```
body :  qprog —> qprog
def  :  qprog —> qprog
eins :  qprog —> qprog
entf :  qprog —> qprog
incm :  qprog —> qprog
mark :  qprog —> qprog
para :  qprog —> qprog
rest :  qprog —> qprog
ruf  :  qprog —> qprog
find :  qprog —> qtripel
akt  :  qtripel —> qprog
pana :  qprog qprog —> qpaar
such :  qprog qprog —> qpaar
tau  :  qprog qprog —> qprog
wda  :  qprog qprog —> qprog
ers  :  qprog qprog qprog —> qprog
```

```
/*
body ( qp ) :: liefert den Body eines Include-Members

def  ( qp ) :: liefert die ( Default- ) Parameter-Liste eines
               Include-Mmebers

eins ( qp ) :: setzt im Body eines Include-Members fuer je-
               des Auftreten eines formalen Parameters den
               aktuellen Wert ein

entf ( qp ) :: entfernt bei einem eingegebenen String das
               abschließende Quote und ersetzt alle auf-
               tretenden Doppel-Quotes durch einzelne Quotes
               z.B.: ''STRING''''S''' —> 'STRING''S'

incm ( qp ) :: bereitet einen Include-Aufruf auf und liefert
               ein Include-Member, in dem die Default-Werte
               durch die aktuellen Werte ersetzt und evtl.
               auftretende weitere Include-Aufrufe ebenfalls
               aufgelöst sind

mark ( qp ) :: markiert die Parameter und die Werte einer
               Parameter-Liste eines Include- Aufrufes mit
               ∘8 bzw. ∘9

para ( qp ) :: liefert die Parameter-Liste eines Include-
               Aufrufs einschließlich der ∘5-Markierung
```

```
rest ( qp ) :: liefert den Programmtext nach einem Include-
               Aufruf

ruf  ( qp ) :: liefert einen kompletten Include-Aufruf
               mit °4- und °5-Markierung und der
               markierten Parameter-Liste

find ( qp ) :: findet in qp den nächsten Include-Aufruf,
               bereitet diesen auf und liefert ein Tripel
               mit:
               1. Komponente : Programmtext vor dem Include-
                               Aufruf
               2. Komponente : aufbereiteter Include-Member
               3. Komponente : Programmtext nach dem Include-
                               Aufruf

akt  ( qp ) :: liefert den aktualisierten Include-Member,
               d.h.: im Body des Include-Members wird jeder
                     formale Parameter durch seinen aktuellen
                     Wert ersetzt.
                     Das Argument qp ist ein Tripel
                     ( q1 q2 q3 ) mit :
                     q1 ==  Include-Member-Name
                     q2 ==  Parameterliste des Aufrufs
                     q3 ==  Include-Member, auf den die
                            Operation rein angew. wurde

pana ( qp1 qp2 ) ::
               sucht in der Parameter-Liste qp1 nach dem
               Parameter-Namen qp2 und liefert ein Tupel mit
               1. Komponente : ....qp2=
               2. Komponente : Programmtext nach dem qp2
                               zugeordneten Wert

such ( qp1 qp2 ) ::
               sucht im Body qp1 nach dem Parameter-Namen
               1. Komponente : Programmtext vor qp2
               2. Komponente : Programmtext nach qp2
```



```
tau ( qp1 qp2 ) ::
               in der Parameter-Liste des Include-Members
               werden die Default-Werte gegen die angegebenen
               aktuellen Werte aus der Parameterliste des
               Include-Aufrufs ausgetauscht

wda ( qp1 qp2 ) ::
               wechselt die Default-Werte in der Parameter-
               Liste des Include-Members qp1 gegen die aktu-
               ellen Werte der Parameter-Liste q2 des Include-
               Aufrufs aus durch Anwendung von tau

ers ( qp1 qp2 qp3 ) ::
               ersetzt jedes Auftreten des Parameter-Namens
               qp2 im Include-Body qp1 durch den Wert qp3
*/
```

DEFINE OPS

```
aufl ( qp ) :=
if zq ( laenge ( sq2 ( find ( rein ( qp )))) null )
   then rein ( qp )
   else cat ( cat ( sq1 ( find ( rein ( qp )))
                    rein ( incm ( sq2 ( find ( rein ( qp ))))))
               aufl ( sq3 ( find ( rein ( qp )))))
fi


incn ( qp ) :=
let  T1 == cons ( ∘8 cons ( ∘5 nil ))               in
if zq ( laenge ( s2 ( srch ( qp T1 ))) null )
   then error.qprog
   else tl ( cut ( s1 ( srch ( qp T1 ))))
fi


body ( qp ) :=
let  T1 == srch ( qp cons ( ' cons ( ) nil )))  in
if eq ( last ( s1 ( T1 )) ) )
   then s2 ( T1 )
   elseif eq ( last ( s1 ( T1 )) ' )
          then body ( s2 ( postz ( s2 ( T1 ))))
          else error.qprog
fi


def ( qp ) :=
let  T1 == srch ( qp cons ( ' cons ( ) nil )))  in
if eq ( last ( s1 ( T1 )) ) )
   then s1 ( T1 )
   elseif eq ( last ( s1 ( T1 )) ' )
          then cat ( cat ( s1 ( T1 )
                           s1 ( postz ( s2 ( T1 ))))
                      def ( s2 ( postz ( s2 ( T1 )))))
          else error.qprog
fi
```



```
eins ( qp ) :=
let  Z2 == succ ( succ ( null ))
     Z3 == succ ( succ ( succ ( null )))
     T1 == srch ( def ( qp ) cons ( = nil ))
     T2 == cut ( tail ( s1 ( T1 ) Z2 ))
     T3 == postz ( tl ( s2 ( T1 )))
     T4 == such ( body ( qp ) T2 )               in
if zq ( laenge ( s2 ( T4 )) null )
   then error.qprog
   elseif zq ( laenge ( cut ( tail ( def ( qp ) Z2 ))) null )
          then cons ( ∘6 tail ( qp Z3 ))
          else eins ( cons ( ∘6
                         cat ( cons ( ( s2 ( T3 ))
                               ers ( body ( qp )
                                     T2
                                     entf ( s1 ( T3 ))))))
fi


entf ( qp ) :=
let  T1 == srch ( qp cons ( ' nil ))              in
if zq ( laenge ( s2 ( T1 )) null )
   then cut ( qp )
   else cat ( s1 ( T1 ) entf ( tl ( s2 ( T1 ))))
fi


incm ( qp ) :=
if zq ( laenge ( qp ) null )
   then error.qprog
   else cat ( qp aufl ( akt ( incn ( qp )
                              para ( qp )
                              rein ( lmbr ( incn ( qp ))))))
fi
```

```
mark ( qp ) :=
let   T1 == srch ( qp cons ( = nil ))
      T2 == postz ( tl ( s2 ( T1 )))
      T3 == cons ( ∘8 cut ( s1 ( T1 )))
      T4 == cons ( ∘9 cons ( ' s1 ( T2 )))      in
if zq ( laenge ( qp ) null )
   then error.qprog
   elseif eq ( hd ( qp ) ) )
           then nil
           elseif eq ( hd ( qp ) , )
                   then mark ( tl ( qp ))
                   else cat ( cat ( T3 T4 ) mark ( s2 ( T2 )))
fi


para ( qp ) :=
tail ( qp succ ( laenge ( incn ( qp ))))


rest ( qp ) :=
let   T1 == srch ( qp cons ( ' cons ( ; nil )))   in
if zq ( laenge ( s2 ( T1 )) null )
   then nil
   elseif eq ( last ( s1 ( T1 )) ; )
           then s1 ( T1 )
           else rest ( s2 ( postz ( s2 ( T1 ))))
fi


ruf ( qp ) :=
let   T1 == srch ( qp cons ( ( cons ( ; nil )))
      L1 == succ ( succ ( succ ( succ ( succ (
              succ ( succ ( succ ( null ))))))))   in
if zq ( laenge ( qp ) null )
   then error.qprog
   elseif eq ( last ( s1 ( T1 )) ; )
           then cat ( cut ( subst ( s1 ( T1 ) ∘4 L1 ))
                      cons ( ∘5 nil ))
           else cat ( cat ( cut ( subst ( s1 ( T1 ) ∘4 L1 ))
                            mark ( s2 ( T1 )))
                      cons ( ∘5 nil ))
fi
```



```
find ( qp ) :=
let T1 == srch   ( qp cons ( ' cons ( % nil )))
    T2 == postz  ( s2 ( T1 ))
    T3 == cons   ( i cons ( n cons ( c cons ( l cons
                 ( u cons ( d cons ( e nil )))))))      in
if zq ( laenge ( qp ) null )
   then trip ( nil nil nil )
   elseif eq ( last ( s1 ( T1 )) ' )
       then trip ( cat ( cat ( s1 ( T1 ) s1 ( T2 ))
                         sq1 ( find ( s2 ( T2 ))))
                   sq2 ( find ( s2 ( T2 )))
                   sq3 ( find ( s2 ( T2 ))))
       elseif not ( cmpr ( s2 ( T1 ) T3 laenge ( T3 )))
              then trip ( cat ( s1 ( T1 )
                                sq1 ( find ( s2 ( T1 ))))
                          sq2 ( find ( s2 ( T1 )))
                          sq3 ( find ( s2 ( T1 ))))
              else trip ( cut ( s1 ( T1 ))
                          ruf ( s2 ( T1 ))
                          rest ( s2 ( T1 )))
fi


akt ( qp ) :=
let qp1 == sq1 ( qp )
    qp2 == sq2 ( qp )
    qp3 == sq3 ( qp )
     T1 == laenge ( qp1 )
     T2 == cons ( ∘7 cons ( ∘1 nil ))
     T3 == subst ( qp3 ∘6 succ ( T1 ))                 in
if zq ( laenge ( qp3 ) null )
   then error.qprog
   elseif not ( eq ( hd ( qp3 ) % ))
          then if cmpr ( qp1 qp2 T1 )
                   then cat ( subst ( qp3 ∘6 T1 ) T2 )
                   else cat ( cons ( ∘6 qp3 ) T2 )
               fi
          elseif zq ( laenge ( qp2 ) succ ( null ))
                   then cat ( eins ( T3 ) T2 )
                   else cat ( eins ( wda ( T3 qp2 )) T2 )
fi
```

```
pana ( qp1 qp2 ) :=
let  Z2 == succ ( succ ( null ))          in
if zq ( laenge ( q1 ) null )
   then error.qpaar
   elseif cmpr ( qp1 qp2 laenge ( qp2 ))
          then pair ( cat ( qp2 cons ( = nil ))
                        s2 ( postz ( tail ( qp1
                                            add ( laenge ( qp2 )
                                                  Z2 )))))
          else pair ( cons ( hd ( qp1 )
                              s1 ( pana ( tl ( qp1 ) qp2 )))
                        s2 ( pana ( tl ( qp1 ) qp2 )))
fi


such ( qp1 qp2 ) :=
let  T1 == srch ( qp1 cons ( % cons ( ' nil )))
     T2 == postz ( s2 ( T1 ))                       in
if not ( mbr ( last ( s1 ( T1 ))
               cons ( % cons ( ' nil ))))
   then pair ( qp1 nil )
   elseif eq ( last ( s1 ( T1 )) ' )
         then pair ( cat ( cat ( s1 ( T1 ) s1 ( T2 ))
                                s1 ( such ( s2 ( T2 ) qp2 )))
                          s2 ( such ( s2 ( T2 ) qp2 )))
         elseif cmpr ( s2 ( T1 ) qp2 laenge ( qp2 ))
                then pair ( cut ( s1 ( T1 ))
                                tail ( s2 ( T1 ) succ (
                                        laenge ( qp2 ))))
                else pair ( cat ( s1 ( T1 )
                                      s1 ( such ( s2 ( T1 )
                                                  qp2 )))
                                 s2 ( such ( s2 ( T1 ) qp2 )))
fi
```



```
taus ( q1 q2 ) :=
let  T1 == tl ( s2 ( srch ( q2 cons ( ∘9 nil ))))
     T2 == tl ( cut ( s1 ( srch ( q2 cons ( ∘9 nil ))))) in
if eq ( hd ( q2 ) ∘5 )
   then q1
   else taus ( cat ( cat ( s1 ( pana ( q1 T2 ))
                                 cons ( ' s1 ( postz ( T1 ))))
                            s2 ( pana ( q1 T2 )))
                  cons ( ∘8
                          s2 ( srch ( s2 ( srch ( q2
                                                   cons ( ∘8
                                                          nil )))
                                      cons ( ∘8 nil )))))
fi


wda ( qp1 qp2 ) :=
cat ( taus ( def ( qp1 ) qp2 )
      body ( qp1 ))


ers ( qp1 qp2 qp3 ) :=
let  T1 == such ( qp1 qp2 )          in
if zq ( laenge ( s2 ( T1 )) null )
   then qp1
   else ers ( cat ( cat ( s1 ( T1 ) qp3 ) s2 ( T1 ))
              qp2
              qp3 )
fi
```

<u>ENDSPEC</u>

5.13. Statistische Aussagen

SSPEC S T A T I S T I K

USE SSPEC INCLAUF

```
PUBLIC OPS      bzeil :  qprog ──> nat
                kzahl :  qprog ──> nat
                kzeil :  qprog ──> nat
                hiera :  qprog ──> qprog
```

/*
In den nun folgenden Properties wird die Operation hiera operationell erklärt.
Aus Gründen der besseren Lesbarkeit wird bei Angaben von Quellcode in SPL' das Blankzeichen (␣) nicht geschrieben.
z.B.: k␣=␣2␣; entspricht k = 2 ;
*/

PROPERTIES

```
exor ( zq   ( laenge ( hiera ( qp )) null )
       and  ( eq ( hd ( hiera ( qp )) ( )
                eq ( last ( hiera ( qp )) ) ))) == true

exor ( and ( mbre ( ( hiera ( qp )) mbre ( ) hiera ( qp )))
       and ( not ( mbre ( ( hiera ( qp )))
                not ( mbre ( ) hiera ( qp ))))) == true

mbre ( ∘3 ( s2srch( qp cons ( ∘3 nil ) pred ( kzahl ( qp )))))
            == true

let T1 == s2srch ( qp cons ( ∘3 nil ) succ ( kzahl ( qp ))) in
zq ( laenge ( T1 ) null ) ==true

zq ( kzahl ( qp ) null )    ⇒   zq ( kzeil ( qp ) null )

not ( mbre ( ∘3 qp ) )      ⇒   zq ( kzahl ( qp ) null )
```



```
let T2 == cons ( ∘2 nil )
    T3 == cons ( ∘1 cons ( ∘2 nil ))
    T4 == cons ( ∘3 nil )
    T5 == cons ( ∘3 cons ( ∘1 nil ))
    T6 == add ( kzeil ( q1 ) kzeil ( q3 ))
    V1 == and ( zq ( laenge ( q2 ) na )
                not ( eq ( vz ∘1 ))    ⇒
                      not ( mbre ( vz q2 )))         in
V1   ⇒
kzeil ( cat ( cat ( cat ( cat ( q1 T3 ) q2 ) T5 ) q3 ))
        == add ( T6 succ ( na ))
V1   ⇒
kzeil ( cat ( cat ( cat ( cat ( q1 T3 ) q2 ) T4 ) q3 ))
        == add ( T6 na )
V1   ⇒
kzeil ( cat ( cat ( cat ( cat ( q1 T2 ) q2 ) T5 ) q3 ))
        == add ( T6 na )
V1   ⇒
kzeil ( cat ( cat ( cat ( cat ( q1 T2 ) q2 ) T4 ) q3 ))
        == add ( T6 pred ( na ))

and ( V1 na > null )    ⇒    bzeil ( q2 ) == pred ( na )

and ( V1 na > null )    ⇒
bzeil ( cat ( cat ( q1 cons ( ∘1 q2 )) q3 ))
        == add ( add ( bzeil ( q1 ) bzeil ( q3 )) na )

kzahl ( cat ( q1 cons ( ∘1 q2 )))
        == succ ( add ( kzahl ( q1 ) kzahl ( q2 )))

lmbr ( m1 ) == %m1 ( va = 'i=1;' )
                 l = 1; %va; l = 2;
⇒  hiera ( aufl ( k = 1;' %include m1 '; k = 2; ))
        == nil

lmbr ( m1 ) == %m1 ( va = 'i=1;' )
                 l = 1; %va; l = 2;
⇒  hiera ( aufl ( k = 1; %include m1; k = 2; ))
        == ( m1 )

lmbr ( m1 ) == %m1 ( va = 'i=1;' )
                 l = 1; %va; l = 2;
⇒  hiera ( aufl ( k = 1; %include m1 ( va = 'n = 3;' );
                 k = 2; ))
        == ( m1 )
```

```
lmbr ( m1 ) == %m1 ( va = 'i=1;' )
                l = 1; %va; l = 2;
⇒  hiera ( aufl ( k = 1;  %include m1; k = 2;
          %include m1 ( va = 't = 5;' ) ; k = 3; ))
          == ( m1 ) ( m2 )

and ( lmbr ( m1 ) == %m1 ( va = 'i=1;' )
                          l = 1; %va; l = 2;
      lmbr ( m2 ) = %m2 ( vb = 's = 7;' , vc = 't = 2;' );
                     z = 1; %vb; z = 2;  %include m1;
                     z = 3; %vc; z = 4; )
⇒  hiera ( aufl ( k = 1; %include m2 ( vc = 'j = 5;' );
                k = 2; ))
          == ( m2 ( m1 ))


/*
bzeil ( qp ) :: Anzahl der Zeilen in qp,  die ausschließlich
                mit Blanks beschrieben sind

kzahl ( qp ) :: Anzahl der Kommentare in qp

kzeil ( qp ) :: Anzahl der Kommentarzeilen ( K.Z. ) in qp,
                mit folgender Konvention :

             ...∘1 ∘2 ∘1 ..k.. ∘1 ∘3 ∘1... == k+1  K.Z.
             ...∘1 ∘2 ∘1 ..k.. ∘1 ∘3 z.... == k    K.Z.
             ....z ∘2 ∘1 ..k.. ∘1 ∘3 ∘1... == k    K.Z.
             ....z ∘2 ∘1 ..k.. ∘1 ∘3 z.... == k-1  K.Z.

             mit z  ≠  ∘1
             und ∘1 ..k.. ∘1  steht für k-mal das Zeichen ∘1.

hiera ( qp ) :: Klammerausdruck, der die Include-Aufruf-
                Hierachie wiedergibt
*/
```



```
PRIVATE OPS    zkom   :  qprog ──> nat


/*
zkom ( qp ) :: Anzahl der Kommentar-Zeilen eines Kommentars
*/


DEFINE OPS

bzeil ( qp ) :=
let  T1 == srch ( aufl ( qp ) cons ( ∘1 nil ))  in
if zq ( laenge ( qp null ))
   then null
   elseif not ( zq ( laenge ( s2 ( T1 )) null ))
          then if eq ( hd ( s2 ( T1 )) ∘1 )
                  then succ ( bzeil ( s2 ( T1 )))
                  else bzeil ( s2 ( T1 ))
               fi
          else null
fi


kzahl ( qp ) :=
if zq ( laenge ( qp ) null )
   then null
   elseif eq ( last ( s1 ( srch ( aufl ( qp )
                                   cons ( ∘3 nil ))))
                 ∘3 )
          then succ ( kzahl ( s2 ( srch ( aufl ( qp )
                                          cons ( ∘3 nil )))))
          else null
fi
```

```
kzeil ( qp ) :=
let  T1 == srch ( aufl ( qp ) cons ( ∘2 nil ))
     T2 == s2 ( srch ( aufl ( qp ) cons ( ∘3 nil )))  in
if zq ( laenge ( qp ) null )
   then null
   elseif not ( eq ( last ( s1 ( T1 )) ∘2 ))
            then null
            elseif eq ( last ( cut ( s1 ( T1 ))) ∘1 )
                   then succ ( add ( zkom ( s2 ( T1 ))
                                     kzeil ( T2 )))
                   else add ( zkom ( s2 ( T1 )) kzeil ( T2 ))
fi

hiera ( qp ) :=
let  T1 == srch ( aufl ( qp ) cons ( ∘4 cons ( ∘5 nil )))   in
if zq ( laenge ( qp ) null )
   then nil
   elseif eq ( last ( s1 ( T1 )) ∘4 )
            then cat ( cons ( ( incn ( cons ( ∘4 s2 ( T1 ))))
                       hiera ( s2 ( T1 )))
            elseif eq ( last ( s1 ( T1 )) ∘5 )
                   then cons ( ) hiera ( s2 ( T1 )))
                   else nil
fi

zkom ( qp ) :=
let  Z1 == succ ( succ ( null ))                              in
if and ( eq ( hd ( qp ) ∘1 )
            eq ( hd ( tl ( qp )) ∘1 ))
   then succ ( zkom ( tl ( qp )))
   elseif and ( eq ( hd ( qp ) ∘1 )
                eq ( hd ( tl ( qp ))∘3 ))
            then if eq ( hd ( tail ( qp Z1 )) ∘1 )
                    then succ ( zkom ( tl ( qp )))
                    else zkom ( tl ( qp ))
                 fi
            elseif eq ( hd ( qp ) ∘3 )
                   then null
                   else error.nat
fi
```

ENDSPEC

6. Erläuterungen zur Top-Level-Spezifikation

In Kapitel 5 wurde die erste formale Spezifikation des Problems auf einer hohen Abstraktionsebene gegeben : die Top-Level-Spezifikation. Sie ist vollkommen implementierungsunabhängig.

Im allgemeinen existieren für ein gestelltes Problem viele mögliche Spezifikationen, die dem Problem mehr oder weniger angemessen sind . Die vorliegende Spezifikation ist bestimmt durch die Interpretation eines eingegebenen Programms als Folge von Symbolen, die nach dem LIFO-Prinzip verarbeitet werden. Man hätte auch eine File-Struktur und ein FIFO-Prinzip spezifizieren können. Auf diese Möglichkeit wird in Kapitel 7 noch einmal näher eingegangen.

Die Top-Level-Spezifikation spaltet das gestellte Problem in geeignete Teilprobleme auf und spezifiziert diese als eigenständige abstrakte Datentypen. Dem Wunsch nach größtmöglicher Modularisierung wird Rechnung getragen. Nach dem Prinzip des "Information Hiding" werden nur diejenigen Operationen in das Interface der einzelnen Datentypen aufgenommen, die auch von anderen Datentypen benutzt werden. Das Zusammenwirken der einzelnen Spezifikationen definiert eine Hierarchie auf den abstrakten Datentypen.

Mit der Spezifizierung der einzelnen Datentypen werden die Datenstrukturen, mit deren Hilfe das Problem gelöst werden soll, festgelegt. Auf dieser hohen Abstraktionsebene werden die Datenstrukturen so gewählt, daß das gestellte Problem möglichst "natürlich" beschrieben werden kann. Auf eine spätere Implementierung wird keine Rücksicht genommen. Sollten sich die hier festgelegten Datenstrukturen für eine spätere Implementierung in einer Programmiersprache als nicht ideal erweisen, so ist es die Aufgabe eines Implementierunsschrittes (auf abstrakter Ebene), diese in geeignetere Strukturen umzuwandeln.

Die einzelnen abstrakten Datentypen sind funktional aufgebaut. Die Operationen , die die Wirkung des Datentyps nach außen beschreiben, sind meistens durch weniger komplexe Operationen, die dem Benutzer nicht zugänglich sind, realisiert. Das Zusammenspiel dieser einfachen Operationen ergibt das gewünschte Resultat.

Für die angegebenen Algorithmen gilt dasselbe wie für die in diesem Schritt festgelegten Datenstrukturen : sie sollen die gewünschten Aufgaben möglichst natürlich beschreiben, auf eine spätere Implementierung wird nicht geachtet.
Die Terminierung der Algorithmen wird nicht nachgewiesen. Wegen der Verwendung von Termsprachen zur Spezifizierung sind die Algorithmen in den meisten Fällen rekursiv aufgebaut.

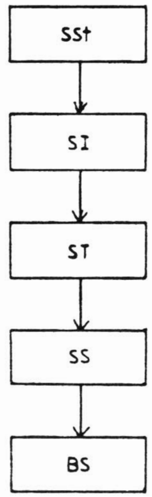
Die von "TRIPLEX" angebotene Möglichkeit der Angabe von PROPERTIES wird in jedem Datentyp in Anspruch genommen. Die Vollständigkeit der angegebenen PROPERTIES ist nicht immer gewährleistet; eine Überspezifizierung, wie sie in [Loe 81b] angesprochen wird, liegt somit nicht vor.
PROPERTIES werden in der vorliegenden Arbeit benutzt, um die Wirkung und das Zusammenspiel der einzelnen Operationen zu verdeutlichen. Die Überprüfung der Konsistenz der angegebenen PROPERTIES mit den entsprechenden Algorithmen verbleibt einer anderen Arbeit.

6.1. Datentyp-Hierarchie

Die in der Top-Level-Spezifikation definierten abstrakten Datentypen sind unterteilt in fünf Gruppen :

1) Basis-Spezifikationen (BS)
2) Struktur-Spzifikationen (SS)
3) Spezifikationen der Textbereinigung (ST)
4) Spezifikationen der Include-Auflösung (SI)
5) Spezifikation der Statistik (SSt)

Diese Gruppen definieren bzgl. der Use-Relation ihrer Elemente folgende grobe Hierarchie :



Datentypen einer Gruppe benutzen über die Use-Relation direkt oder indirekt Datentypen aus untergeordneten Gruppen. Die exakten Hierarchie-Beziehungen zwischen den einzelnen Datentypen sind auf der folgenden Seite dargestellt.

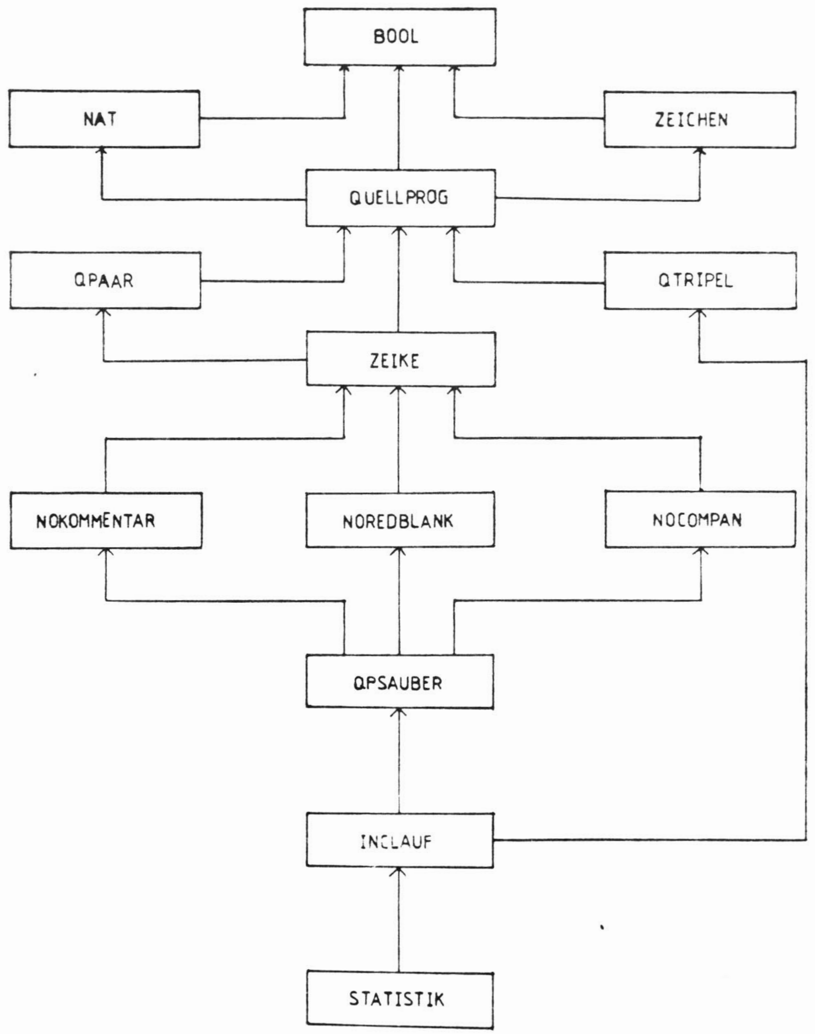


Abb.: Datentyp-Hierarchie

6.2. Basis-Spezifikationen

Die Basis-Spezifikationen BOOL, NAT und ZEICHEN stellen die grundlegenden Datentypen zur Verfügung, um das gestellte Problem spezifizieren zu können.

BOOL ermöglicht das Arbeiten mit aussagenlogischen Konstrukten.

NAT stellt die zahlentheoretischen Grundlagen bzgl. der natürlichen Zahlen zur Verfügung.

ZEICHEN **definiert, aus welchen Symbolen ein zu bearbeitendes** Quellprogramm bestehen darf, und welche internen Sonderzeichen benutzt werden.

6.3. Struktur-Spezifikationen

Die Struktur-Spezifikationen QUELLPROG, QPAAR und QTRIPEL stellen die grundlegenden Datenstrukturen zur Verfügung, auf denen die komplexen Datentypen arbeiten.

QUELLPROG spezifiziert ein eingegebenes Quellprogramm als Folge von Symbolen der Sorte 'zeichen', die nach dem LIFO-Prinzip abgearbeitet wird.

QPAAR spezifiziert ein Paar von Elementen der Sorte 'qprog'.

QTRTIPEL spezifiziert ein Tripel von Elementen der Sorte 'qprog'.

6.4. Spezifikationen der Textbereinigung

Die Spezifikationen der Textbereinigung spezifizieren die

a) Sonderbehandlung von Zeichenkettenkonstanten (ZEIKE)
b) Entfernung von Kommentaren (NOKOMMENTAR)
c) Entfernung von Compile-Anweisungen (NOCOMPAN)
d) Entfernung von redundanten Blanks (NOREDBLANK)
e) Zusammenfassung von b) - d) (QPSAUBER)

Es werden keine neuen Sorten definiert

6.5. Spezifikation der Include-Auflösung

Der Datentyp INCLAUF spezifiziert die Abarbeitung von Include-Aufrufen unter Berücksichtigung sämtlicher erlaubter Aufruf-Varianten :

- Include-Aufruf und Include-Member ohne Parameter
- Include-Aufruf ohne und Include-Member mit Parameter
- Include-Aufruf und Include-Member mit Parameter .

Es wird keine neue Sorte definiert.

6.6. Spezifikation der Statistik

Der Datentyp STATISTIK spezifiziert die Berechnung folgender Angaben :

- Anzahl der Leerzeilen
- Anzahl der Kommentare
- Anzahl der Kommentarzeilen
- Bestimmung der Include-Aufruf-Hierarchie .

Es wird keine neue Sorte definiert.

6.7. Informelle Beschreibung der Datentypen

In diesem Abschnitt werden informelle Erläuterungen zu den in Kapitel 5 formal spezifizierten abstrakten Datentypen gegeben.

Die rekursiven Algorithmen, die die einzelnen Operationen spezifizieren, sind formal in der Regel so aufgebaut, daß von der Aufschreibung her zuerst die Abbruchkriterien (Fehlerausgänge oder Ende des Algorithmus) abgetestet werden und dann die Rekursion ausgeführt wird.

Bei der Erstellung der PROPERTIES in den Datentypen wird weitgehend die Präfix-Notation von "TRIPLEX" beibehalten. Es wird zusätzlich mit Implikationen und Äquivalenzrelationen, die in Infix-Notation dargestellt sind, gearbeitet, um Voraussetzungen und Abhängigkeiten darstellen zu können. Bei den komplexen Datentypen INCLAUF und STATISTIK werden in Gleichungen eingekleidete Beispiele als PROPERTIES aufgeführt, um die Wirkung von komplexen Operationen exemplarisch zu verdeutlichen.

Zum besseren Verständnis werden im weiteren Text die Namen der Datentypen und Operationen, die nicht intuitiv klar sind, erläutert, indem die Erklärungen für die Namen in Klammern hinter dem jeweiligen Namen aufgeführt sind.
Bei Abkürzungen sind die entsprechenden Buchstaben in den Erläuterungen groß geschrieben.

Beispiele : **QPROG** (QuellPROGramm)
BOOL (BOOLsche Algebra)

6.7.1. **BOOL** (BOOLsche Algebra)

Der Datentyp BOOL spezifiziert die Boolsche Algebra mit den Operationen
true und false als Konstruktoren der Sorte 'bool'
und den grundlegenden Operationen not , and und or .

Die Operation exor wurde lediglich zur Vereinfachung von PROPERTIES , die in anderen Datentypen aufgestellt werden, spezifiziert.

6.7.2. **NAT** (NATürliche Zahlen)

Der Datentyp NAT spezifiziert die natürlichen Zahlen einschließlich der Null.
Die neu definierte Sorte 'nat' wird durch die Konstruktoren null und succ aufgebaut.

null (NULL) spezifiziert den Basisterm *null .

Wegen der Unterscheidung zu dem Symbol 0, das in ZEICHEN als Operation 0 definiert ist, wird die 0-Operation in NAT unter dem Namen null spezifiziert.

succ (SUCCessor) ist die Nachfolgerfunktion auf den natürlichen Zahlen.

Die weiteren Operationen in NAT spezifizieren :

die Vorgängerfunktion pred (PREDecessor),
die Addition add ,
die Subtraktion sub und
den Vergleich zweier natürlicher Zahlen zq (Zahlen-ÄQivalenz).

6.7.3. **ZEICHEN** (erlaubte ZEICHEN)

Der Datentyp ZEICHEN spezifiziert als neue Sorte 'zeichen' die Symbole, die als Bestandteile der betrachteten Programme erlaubt sind :

- alle Buchstaben des Alphabets (a ,..., z)
- alle Ziffern (0 ,..., 9)
- Blank (␣)
- Sonderzeichen von "SPL'" (+ , % , " ,...)
- interne Sonderzeichen (°1 ,..., °9)

Die internen Sonderzeichen werden für die weitere Bearbeitung (Kommentarbehandlung,Include-Aufruf-Auflösung,statistische Aussagen) benötigt;
sie dürfen in dem ursprünglichen Quellprogramm nicht auftreten. Die Zeilenende-Markierungen °1 müssen beim Einlesen gesetzt werden.

Zur leichteren Verarbeitung von Symbolen der Sorte 'zeichen' werden zwei Prädikate eingeführt :

- sonz (SONderZeichen)
  zur Bestimmung, ob ein Symbol Sonderzeichen von "SPL'" ist
- eq (EQality)
  zur Bestimmung der Äquivalenz von Symbolen .

Zur Erleichterung der Aufschreibung der Konstruktoren und der Fallunterscheidungen bei den Algorithmen wird ausschließlich in ZEICHEN mit einer informellen, mit "TRIPLEX" nicht konsistenten, aber intuitiv eindeutigen Notation gearbeitet.
Bei Aufzählungen steht eine Folge von Punkten für alle die Operationen, die unter PUBLIC OPS in ZEICHEN definiert, aber nicht explizit unter CONSTRUCTORS aufgeführt sind.

6.7.4. **QUELLPROG** (QUELLPROGramm)

Der Datentyp QUELLPROG spezifiziert als neue Sorte 'qprog', die wichtigste Datenstruktur der Gesamt-Spezifikation, die Liste.
Jedes Programm, das von "INTAKT" verarbeitet werden soll (Quellprogramm), wird interpretiert als eine Liste von Symbolen der Sorte 'zeichen'.

Die Terme dieser Sorte werden durch die Konstruktoren nil und cons aufgebaut.

nil erzeugt die leere Liste.
cons fügt ein Symbol an den Anfang einer Liste hinzu.

Die weiteren Operationen, die in QUELLPROG spezifiziert werden, dienen zur Listenmanipulation oder liefern Aussagen über Listen.
Alle Operationen bearbeiten eine Liste nach dem LIFO-Prinzip, d.h. das Symbol, das zuletzt in die Liste eingefügt wurde, wird als erstes Symbol gelesen und bearbeitet.

Die Operationen stellen folgende Möglichkeiten der Listenverarbeitung zur Verfügung :

- die Bestimmung der Länge einer Liste (laenge)
- das Herausfiltern der ersten n Symbole einer Liste (hd , head , cut)
- das Herausfiltern der letzten n Symbole einer Liste (tl , tail , last)
- die Konkatenation zweier Listen (cat)
- die Substitution der ersten n Symbole einer Liste L durch eine Liste L' (subst)
- den Vergleich der ersten n Symbole zweier Listen (cmpr)
- die Bestimmung des Enthaltenseins eines Symbols in einer Liste (mbre)

Einige Operationen von QUELLPROG lassen sich durch reine Kombination von anderen Operationen direkt ableiten (siehe PROPERTIES von QUELLPROG). Bei der Spezifikation dieser Operationen in der Top-Level-Spezifikation wurde auf diese Konstruktionsmöglichkeit bewußt verzichtet, um die Arbeitsweisen dieser Operationen jeweils separat darstellen zu können.

Durch die umfangreiche Auswahl von Operationen, die QUELLPROG zur Verfügung stellt, wird das Arbeiten mit Listen bei der Textbereinigung, der Include-Auflösung und der Statistik sehr erleichtert.

6.7.5. **QPAAR** (Quellprogramm-PAAR)

Der Datentyp QPAAR spezifiziert als neue Sorte 'qpaar', Zwei-Tupel von Elementen der Sorte 'qprog'.

Die einzigen Operationen sind der Konstrutor
pair , der aus zwei Listen ein Paar (engl. pair) von Listen macht,
und die Selektoren
s1 und s2 , die aus einem Paar von Listen die erste bzw. die zweite Liste herausfiltern.

QPAAR wird eingeführt, um das Arbeiten innerhalb einer Liste zu vereinfachen, indem man eine Liste an der gewünschten Stelle aufbrechen kann und die beiden Teillisten dann zu einem Zwei-Tupel der Sorte 'qpaar' zusammenfaßt.

QPAAR ist wie auch QTRIPEL für die Spezifizierung des Gesamtproblems nicht unbedingt notwendig, erleichtert aber das Suchen und Vergleichen sowie das Suchen und Ersetzen von Symbolfolgen und macht das Arbeiten innerhalb einer Liste überschaubarer.

6.7.6. **QTRIPEL** (Quellprogramm-TRIPEL)

Der Datentyp QTRIPEL spezifiziert als neue Sorte 'qtripel', Tripel von Elementen der Sorte 'qprog'.

Als Operationen werden der Konstruktor
trip , der aus drei Listen ein Tripel von Listen macht,
und die Selektoren
sq1 , sq2 und sq3 , die aus einem Tripel von Listen die erste, zweite oder die dritte Liste herausfiltern,
spezifiziert.

QTRIPEL wird eingeführt, um das Auflösen von Include-Aufrufen übersichtlicher zu machen, indem man einerseits beim Auftreten eines Include-Aufrufs das Quellprogramm in drei Teillisten aufspaltet und zu einem 'qtripel' zusammenfaßt, um leichter an der Stelle des Aufrufs arbeiten und die Zusammengehörigkeit der Teillisten gewährleisten zu können (siehe Operation find in INCLAUF) und andererseits einen aktualisierten Include-Member in seinen drei Hauptbestandteilen leicht darstellen kann (siehe Operation akt in INCLAUF).

QTRIPEL ist für die Spezifizierung des Gesamtproblems nicht unbedingt notwendig, macht aber die Spezifikation der Include-Auflösung erheblich übersichtlicher.

6.7.7. **ZEIKE** (ZEIchenKEtten-Konstante)

Der Datentyp ZEIKE spezifiziert Operationen, die in erster Linie die Sonderbehandlung von Zeichenketten-Konstanten bzgl. der Textbereinigung, der Include-Auflösung und der Statistik erleichtern.

srch (SeaRCH) ist eine Suchoperation, die in einer Liste L nach dem ersten Auftreten eines Elements einer weiteren Liste L' sucht, bei erfolgreichem Suchen die Liste L an dem "Fundort" aufspaltet und aus den beiden Teillisten ein Tupel der Sorte 'qpaar' bildet.
Es ist also möglich, gleichzeitig nach mehreren Symbolen zu suchen. Insbesondere kann nach einem Quote (Häkchen) gesucht werden, das den Anfang einer Zeichenketten-Konstanten kennzeichnet.
Durch eine gekoppelte Anwendung der Operationen srch und cmpr (siehe QUELLPROG) läßt sich ein Pattern-Matching durchführen, das zum Auffinden von Schlüsselwörtern und Parameternamen geeignet ist.

postz (POST Zeichenketten-Konstanten) ist eine Operation, die dem Überlesen bzw. Lesen von Zeichenketten-Konstanten dient.
Nachdem durch Anwendung von srch ein Quote gefunden worden ist, läßt sich durch Anwendung von postz auf die zweite Komponente des von srch gelieferten Ergebnisses die gefundene Zeichenketten-Konstante lesen und von der Liste trennen.

zaehl ("ZAEHLt" Quotes) ist eine Operation, die eine Aussage macht über die "Anzahl" der Quotes in einer Liste der Sorte 'qprog'.
zaehl liefert den Wert true, wenn die Anzahl der Quotes ungerade ist und den Wert false sonst.
zaehl wird ausschließlich in PROPERTIES benutzt, um allgemeine Aussagen über Listen der Sorte 'qprog' machen zu können, die die Sonderstellung der Zeichenketten-Konstanten berücksichtigen.

s2srch (Suche in der 2-ten Komponenten von SRCH) ist eine Operation, die das mehrfache Suchen von Symbolen erlaubt, d.h. es ist möglich, das n-te Auftreten eines Symbols zu bestimmen.
Der Name leitet sich aus dem Algorithmus ab; es wird jeweils in der zweiten Ergebnis-Komponente des vorherigen Suchens weitergesucht.
s2srch wird nur in den PROPERTIES von STATISTIK benutzt, um die Wirkung der Zähloperation kzahl zu beschreiben.

6.7.8. **NOKOMMENTAR** (NO KOMMENTAR)

Der Datentyp NOKOMMENTAR spezifiziert Operationenen, die Kommentare eines Quellprogramms eliminieren und an deren Stelle Kommentaranfang- und Kommentarende-Markierungen setzen.

kweg (Kommentar WEG) entfernt aus einer gegebenen Liste der Sorte 'qprog' sämtliche Kommentare und kennzeichnet Anfang und Ende der entfernten Kommentare mit Kommentaranfang- und Kommentarende-Zeichen.
Die in den Kommentaren enthaltenen Zeilenende-Markierungen bleiben erhalten, um auch nach Entfernung der Kommentare noch die Anzahl der Kommentarzeilen feststellen zu können.

kzeil (KommentarZEILe) wird nach Erkennen eines Kommentaranfangs in kweg aufgerufen und entfernt diesen einen Kommentar bis auf evtl. vorkommende Zeilenende-Markierungen. kzeil ist eine private Operation und kann somit nicht außerhalb von NOKOMMENTAR aufgerufen werden.

6.7.9. **NOREDBLANK** (NO REDundant BLANK)

Der Datentyp NOREDBLANK spezifiziert Operationen, die redundante Blanks in einer Liste der Sorte 'qprog' ersatzlos entfernen.

bweg (Blank WEG) entfernt alle eine Symbolfolge anführenden Blanks und ruft dann die Operation nob auf.

nob (NO Blanks) entfernt alle redundanten Blanks innerhalb einer Liste der Sorte 'qprog'.

6.7.10. **NOCOMPAN** (NO COMPile-ANweisungen)

Der Datentyp NOCOMPAN spezifiziert die Operation

cweg (Compile-Anweisungen WEG), die aus einer Liste der Sorte 'qprog' alle Compile-Anweisungen ersatzlos entfernt.

6.7.11. **QPSAUBER** (QuellProgramm SAUBER)

Der Datentyp QPSAUBER spezifiziert die Operation rein (REINes Quellprogramm), die durch Anwendung der Interface-Operationen von NOKOMMENTAR, NOREDBLANK und NOCOMPAN eine "saubere" Liste der Sorte 'qprog' liefert,
d.h. eine Liste von Symbolen der Sorte 'zeichen' ohne semantisch irrelevante Programmteile.

QPSAUBER stellt somit eine Zusammenfassung der Datentypen NOKOMMENTAR, NOREDBLANK und NOCOMPAN dar und steuert die Reihenfolge der Ausführung der in diesen Datentypen spezifizierten Operationen.

6.7.12. INCLAUF (INCLude-Aufruf-AUFlösung)

Der Datentyp INCLAUF spezifiziert die Auflösung von Include-Aufrufen. Es werden sämtliche Aufruf-Varianten (siehe PROPERTIES von INCLAUF) berücksichtigt.

Auflösen bedeutet das Ersetzen der Include-Aufrufe durch die entsprechenden aktualisierten Include-Member, wobei weitere Include-Aufrufe innerhalb des Include-Members ebenfalls aufgelöst werden.

Include-Aufruf-Anfang und -Ende, Include-Member-Anfang und -Ende, Parameternamen und -werte werden durch die entsprechenden internen Sonderzeichen gekennzeichnet.

Von außen sichtbare Operationen :

- aufl (AUFLösen von Include-Aufrufen)
  löst sämtliche Include-Aufrufe in einem Quellprogramm auf;
  diese Operation wird durch die privaten Operationen realisiert

- incn (INCLude-Member-Name)
  liefert als Ergebnis den Include-Member-Namen bzgl. eines vorgegebenen Include-Aufrufs;
  diese Operation muß von außen sichtbar sein, um in STATISTIK die Include-Aufruf-Hierarchie erstellen zu können

- lmbr (Lies Include-MemBeR)
  liest einen kompletten Include-Member aus der Include-Bibliothek;
  diese Operation ist vorgegeben und wird in dieser Arbeit nicht näher spezifiziert

Da alle Operationen von INCLAUF innerhalb von Kommentaren in der formalen Spezifikation (Kapitel 5.12.) informell erklärt sind, werden in diesem Kapitel keine zusätzlichen Erläuterungen zu den privaten Operationen gegeben.
Stattdessen wird das Zusammenspiel der einzelnen Operationen aufgezeigt.
Dieses geschieht mit Hilfe von zwei verschiedenen Darstellungsarten, die einander ergänzen :

- schematische Darstellung des Zusammenspiels der die Operation aufl realisierenden Operationen (6.7.12.1.)
- Simulation der Abarbeitung eines imaginären Aufrufs der Operation aufl (6.7.12.2.) .

6.7.12.1. Schematische Darstellung

Das hier verwendete Schema entspricht bis auf wenige Abweichungen den bekannten Programmablaufplänen.

Die Rechtecke des Schemas symbolisieren

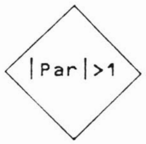
- Operationen
- Aufgabenbeschreibungen
- Ergebnisbeschreibungen .

Rechtecke, die Operationen symbolisieren , können mehrere Ausgänge haben, um die Abarbeitung der in Ausdrücken auftretenden Operationen weiterverfolgen zu können. In diesen Fällen haben die aus dem Rechteck herausführenden Pfeile an ihrem Ende eine Verdickung (●——>) , die sich unter oder über dem Namen der Operation befindet, deren Abarbeitung weiter erklärt werden soll. Solche Verzweigungen enden immer in Rechtecken, die Ergebnisse symbolisieren.

Die Rauten des Schemas symbolisieren Verzweigungen des Algorithmus. Die darin enthaltenen Ausdrücke stehen stellvertretend für die Frage nach dem Vorhandensein der durch sie beschriebenen Objekte oder für die Frage nach der Gültikeit des Ausdrucks.

Beispiele :

== Gibt es Include-Aufrufe ?



== Ist die Anzahl der Parameter größer 1 ?

Die auf den nächsten drei Seiten folgenden Abbildungen bilden in ihrer Gesamtheit eine schematische Darstellung der Arbeitsweise der Operation <u>aufl</u> .

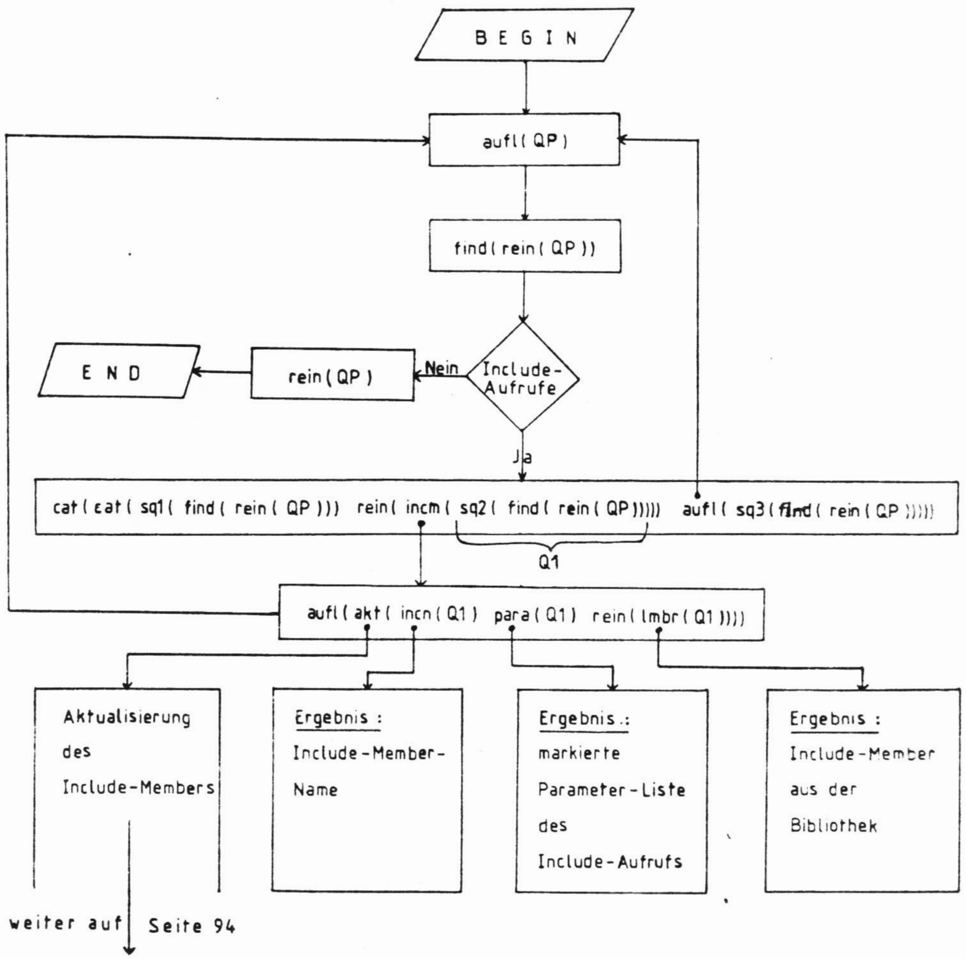
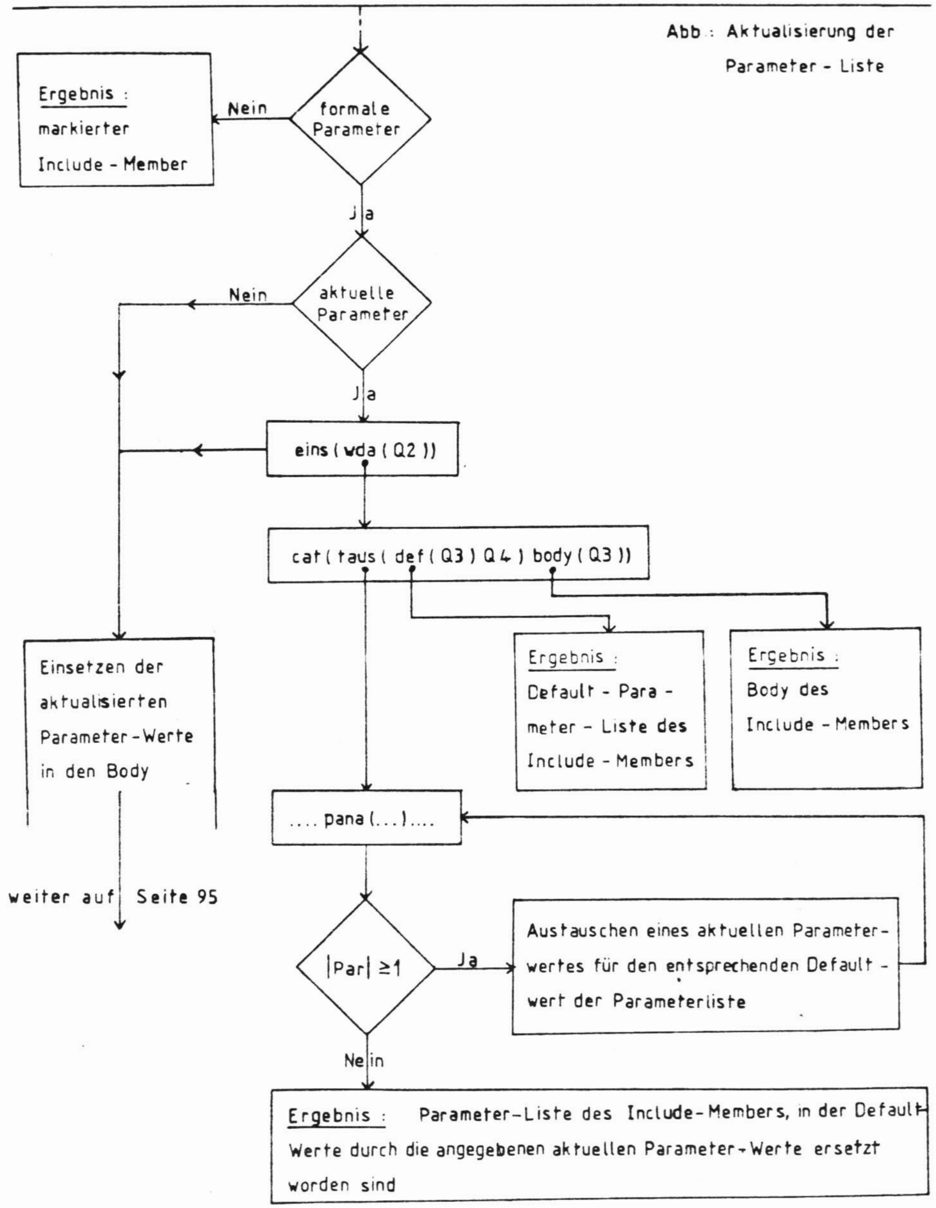


Abb.: Auflösung von Include-Aufrufen

Abb : Aktualisierung der Parameter - Liste



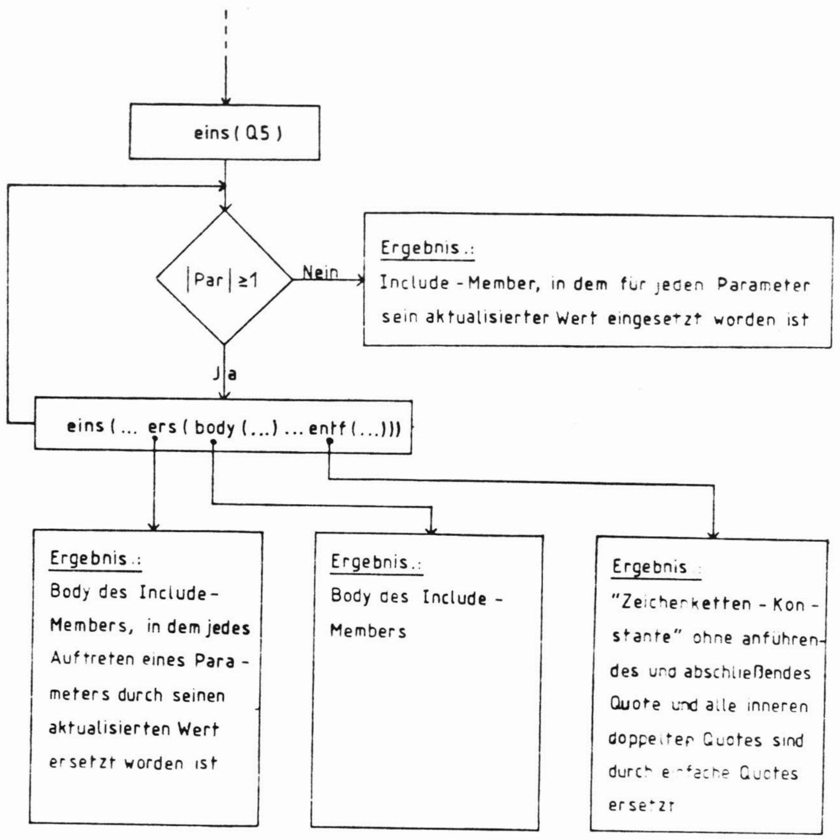


Abb.: Aktualisierung des Include-Members

6.7.12.2. Simulation der Abarbeitung

Ausgehend von einem imaginären Aufruf der Operation <u>aufl</u> werden die Aufrufe und Ergebnisse der zur Abarbeitung benötigten Operationen betrachtet.

Die komplexeste Alternative des Algorithmus von <u>aufl</u> wird betrachtet, die dort aufgeführten Operationsaufrufe werden erklärt, und von den Algorithmen der entsprechenden Operationen wird wie bei <u>aufl</u> wiederum die komplexeste Alternative weiterverfolgt, bis man zu Operationen gelangt, die nicht in INCLAUF spezifiziert sind.

Die weniger komplexen Alternativen der Algorithmen sind entweder Fehlerausgänge oder Abbruchkriterien oder lassen sich leicht aus den komplexeren Alternativen ableiten.

<u>Bemerkungen zur Notation :</u>

Die jeweils ausgewählte Alternative, der Operationsaufruf oder das Ergebnis der Operationsausführung steht, getrennt durch "-:", rechts neben oder unter dem gerade betrachteten Operationsaufruf.

Erklärungen zu Operationen, Argumenten des Operationsaufrufs und /oder zur Wirkung der Algorithmen stehen als "SPL'"-Kommentare unter oder neben dem gerade betrachteten Operationsaufruf.

Wegen der notwendigen Unterscheidung zwischen Klammern, die zu dem zu bearbeitenden Quellprogramm und solchen, die zu Ausdrücken innerhalb der Abarbeitung gehören, wird folgende Konvention getroffen :

- (und) bezeichnen Klammern für Argumente der Operationen aus INCLAUF
- [und] bezeichnen Klammern, die in dem zu bearbeitenden Quellprogramm auftreten.

In Abweichung zu der in Kapitel 1.1. festgelegten Notation gilt :

- die Operationsnamen werden außerhalb der Kommentare nicht unterstrichen.
- innerhalb der erklärenden "SPL'"-Kommentare werden Operationsnamen in "" eingeschlossen.
- Argumente von Operationen werden, wenn sie nicht **selbst** wieder Operationsaufrufe sind, mit großen Buchstaben geschrieben.
- Namen innerhalb der Argumente werden ihrer semantischen Bedeutung nach frei benannt.

<u>Abkürzungen, die im weiteren Text benutzt werden :</u>
(in der Reihenfolge ihres Auftretens aufgelistet)

```
QP      == eingegebenes Quellprogramm
MBNAME  == Include-Member-Name
PVI     == Programmtext vor dem Include-Aufruf
PNI     == Programmtext nach dem Include-Aufruf
PLIS    == Parameter-Liste des Include-Aufrufs
DEFP    == Default-Parameter-Liste des Include-Members
AKTDEFP == aktualisierte Pararmeter-Liste des Include-Members
PARi    == i-ter Parameter einer Parameter-Liste
WERTi   == Wert des i-ten Parameters
BODY    == Body des Include-Members
PNAME   == Parametername
DLV     == Defaultwerte-Liste vor einem Parameternamen
DLN     == Defaultwerte-Liste nach einem Parameternamen
AKTBODY == aktualisierter Include-Member-Body
BVP     == Body vor dem Parameternamen
BNP     == Body nach dem Parameternamen
A1BODY  == Body, der bzgl. genau eines Parameters
           aktualisiert ist
```



```
aufl(QP)  -:
    cat(cat(sq1(find(rein(QP))) incm(sq2(find(rein(QP)))))
        aufl(sq3(find(rein(QP)))))

/* QP ist ein eingegebenes Quellprogramm.
   "sq1","sq2" und "sq3" ermöglichen den Zugriff auf die
   einzelnen Ergebniskomponenten von "find".
   Durch die doppelte Konkatenation werden die einzelnen
   Programmteile wieder zu einem Quellprogramm zusammenge-
   fügt                                                    */


find(QP)  -:  trip(PVI
                   ruf (INCLUDE␣MBNAME[PLIS];PNI)
                   rest(INCLUDE␣MBNAME[PLIS];PNI))

/* "find" findet in einem eingegebenen Quellprogramm den
   ersten Include-Aufruf, falls einer vorhanden ist.
   Als Ergebnis liefert "find" ein Tripel mit :
   i)   PVI
   ii)  aktualisierter Include-Member
   iii) PNI                                                */


rest(INCLUDE␣MBNAME[PLIS];PNI)  -:  PNI

/* "rest" liefert als Ergebnis den Rest eines Programms,
   d.h. den Programmtext, der hinter dem Aufruf steht,
   und das ist PNI                                         */


ruf(INCLUDE␣MBNAME[PLIS];PNI)  -:
   cat(cat(∘4MBNAME mark([PLIS];PNI)) ∘5)

/* "ruf" bearbeitet den Include-Aufruf :
   Das Schlüsselwort INCLUDE wird eliminiert.
   Die Include-Aufruf-Anfang-Markierung ∘4 und die Include-
   Aufruf-Ende-Markierung ∘5 werden gesetzt.
   "mark" wird zum Markieren der Parameter aufgerufen.     */
```

```
mark([PLIS];PNI)  -:  ∘8PAR1∘9WERT1∘8PAR2∘9WERT2....

/* "mark" markiert die Parameternamen und die Parameterwerte
   der Parameterliste des Include-Aufrufs                   */


/* Mit den oben erläuterten Operationen ist "find" erklärt */


incm(∘4MBNAME[∘8PAR1∘9WERT1∘8PAR2∘9WERT2....]∘5)  -:
    aufl(akt(incn(Q1) para(Q1) rein(lmbr(incn(Q1)))))

/* "incm" bearbeitet den Include-Member.
   Q1 steht stellvertretend für das Argument von "incm", d.h.
   Q1 == ∘4MBNAME[∘8PAR1∘9WERT1∘8PAR2∘9WERT2....]∘5 .
   Q1 ist ein aufbereiteter Include-Aufruf.
   Auf das Ergebnis von "incm" muß noch einmal die Operation
   "aufl" angewendet werden, um innere Include-Aufrufe
   aufzulösen                                               */


incn(∘4MBNAME[∘8PAR1∘9WERT1∘8PAR2∘9WERT2....]∘5)  -:  MBNAME

/* "incn" liefert als Ergebnis den Include-Member-Namen
   eines aufbereiteten Include-Aufrufs                      */


para(∘4MBNAME[∘8PAR1∘9WERT1∘8PAR2∘9WERT2....]∘5)  -:
    ∘8PAR1∘9WERT1∘8PAR2∘9WERT2....∘5

/* "para" liefert als Ergebnis die Parameter-Liste eines
   aufbereiteten Include-Aufrufs einschließlich der Mar-
   kierung ∘5                                               */
```



```
lmbr(MBNAME)  -:  %MBNAME[DEFP]BODY

/* "lmbr" liefert als Ergebnis eine Kopie des Include-
   Members mit dem Namen MBNAME aus der Include-Bibliothek */


akt(MBNAME
    ∘8PAR1∘9WERT1∘8PAR2∘9WERT2....∘5
    %MBNAME[DEFP]BODY)  -:
   eins(wda(∘6[DEFP]BODY ∘8PAR1∘9WERT1....∘5))∘7∘1

/* "akt" aktualisiert einen gelesenen Include-Member, d.h.:
   - setzen der Include-Member-Anfang- und -Ende-Marken
   - aktualisieren der Defaultwerte
   - setzen einer Zeilenende-Markierung                    */


wda(∘6[DEFP]BODY ∘8PAR1∘9WERT1....∘5)  -:
   cat(taus(def(∘6[DEFP]BODY) ∘8PAR1∘9WERT1....∘5)
       body(∘6[DEFP]BODY)))

/* "wda" wechselt die Argumente aus, d.h.:
   als Ergebnis wird eine Symbolfolge geliefert, die aus
   der aktualisierten Parameterliste und dem Body des
   Include-Members besteht                                 */


body(∘6[DEFP]BODY)  -:  BODY

/*"body" liefert den Body eines Include-Members            */


def(∘6[DEFP]BODY)  -:  ∘6[DEFP]

/* "def" liefert die Defaultwerte-Liste eines Include-
   Members einschließlich des anführenden ∘6-Zeichens     */
```

```
taus(∘6[DEFP] ∘8PAR1∘9WERT1∘8...∘5)   -:   ∘6[AKTDEFP]

/* "taus" tauscht in der Defaultwerte-Liste eines Include-
   Members die Defaultwerte gegen die angegebenen aktuellen
   Parameter-Werte aus.
   Die Operation "pana" wird zum Auffinden der gesuchten
   Parameter-Namen benutzt                                  */


pana(∘6[DEFP] PNAME)   -:   pair(DLVPNAME= DLN)

/* "pana" sucht den Parameternamen PNAME in der Defaultwerte-
   Liste DEFP.
   Als Ergebnis wird ein Zwei-Tupel geliefert mit :
   - DLVPNAME=
   - DLN                                                    */


eins(∘6[AKTDEFP]BODY)   -:   ∘6AKTBODY

/* "eins" setzt in dem Body eines Include-Members die Werte
   der aktualisierten Defaultwerte-Liste AKTDEFP für die
   formalen Parameter ein.
   Die Operationen "entf", "such" und "ers" werden dafür
   benutzt.
   Das Ergebnis ist der aktualisierte Include-Member-Body
   AKTBODY, der von dem ∘6-Zeichen angeführt wird.          */
```



```
such(BODY PNAME)   -:   pair(BVP BNP)

/* "such" sucht im Body eines Include-Members nach einem
   Parameternamen PNAME.
   Als Ergebnis wird ein Zwei-Tupel geliefert mit :
   - BVP
   - BNP                                                    */


entf(ZEICHENKETTE)   -:   ZEICHENKETTE-Q

/* "entf" entfernt in der eingegebenen Zeichenkette das an-
   führende und das abschließende Quote und ersetzt jedes
   Auftreten von zwei aufeinanderfolgenden Quotes durch ein
   einzelnes Quote (ZEICHENKETTE-Q).
   Dies ist nötig, da die aktuellen Parameterwerte Zeichen-
   ketten-Konstanten sind und die darin vorkommenden Quotes
   nach "SPL'"-Syntax verdoppelt sein müssen. Bei dem Ein-
   setzen der aktuellen Werte in den Body werden diese nicht
   mehr als Zeichenketten-Konstanten betrachtet und die
   überflüßigen Quotes müssen wieder entfernt werden.       */


ers(BODY PNAME PWERT)   -:   A1BODY

/* "ers" ersetzt in einem Body eines Include-Members jedes
   Auftreten des Parameter-Namens PNAME durch den Parameter-
   wert PWERT.
   Das Ergebnis ist der Body A1BODY, der bzgl. dieses einen
   Parameters aktualisiert ist.                             */
```

6.7.13. STATISTIK (STATISTIK)

Der Datentyp STATISTIK spezifiziert Operationen zur Bestimmung statistischer Aussagen über Programme, die schon aufbereitet sind.

bzeil (BlankZEILen) liefert als Ergebnis die Anzahl der Leerzeilen eines aufbereiteten Programms, d.h. die Anzahl der Leerzeilen, die ausschließlich mit Blanks gefüllt waren.
Nach der Aufbereitung charakterisieren zwei aufeinanderfolgende ∘1-Zeichen eine Leerzeile.

kzahl (KommentaranZAHL) liefert als Ergebnis die Anzahl der Kommentare.
Nach der Aufbereitung bestimmt die Anzahl der ∘3-Zeichen die Kommentar-Anzahl.

kzeil (KommentarZEILen-Anzahl) liefert als Ergebnis die Anzahl der Kommentarzeilen, wobei folgende Konventionen gemacht werden :

```
- ...∘1∘2∘1 ..k.. ∘1∘3∘1... == k+1 Kommentarzeilen
- ...∘1∘2∘1 ..k.. ∘1∘3z.... == k   Kommentarzeilen
- ....z∘2∘1 ..k.. ∘1∘3∘1... == k   Kommentarzeilen
- ....z∘2∘1 ..k.. ∘1∘3z.... == k-1 Kommentarzeilen
```

mit z ≠ ∘1
und ∘1 ..k.. ∘1 steht für k-mal das Zeichen ∘1.

kzeil benutzt die private Operation zkom (Zeilen pro KOMmentar), die die Anzahl der Zeilen eines Kommentars bestimmt.

hiera (HIERAchie) liefert als Ergebnis einen Klammerausdruck, der die Include-Aufruf-Hierarchie wiedergibt.
Der Klammerausdruck ist so aufgebaut, daß ein Include-Member M, der innerhalb eines anderen Include-Members M' aufgerufen wird, in der nächst tieferen zu M' gehörenden Schachtelungstiefe steht. Include-Member, die nacheinander aufgerufen werden, gehören zur selben Schachtelungstiefe.

Beispiele :
(M1)(M2) == M1 und M2 werden nacheinander aufgerufen
(M2(M1)) == M1 wird im Body von M2 aufgerufen

. Die Beispiele, die in den PROPERTIES in 5.7.13. verwendet werden, sind identisch den Beispielen in INCLAUF (5.7.12.).

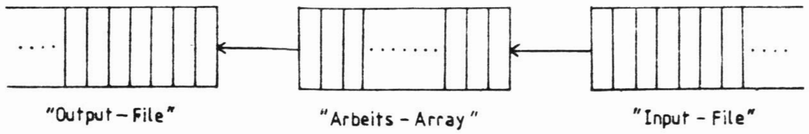
7. Implementierungsschritt

In diesem Kapitel wird der abstrakte Datentyp QUELLPROG, der in 5.5.4. spezifiziert wurde, durch einen neuen Datentyp IMPL_QUELLPROG, der auf einem niedrigeren Abstraktionsniveau spezifiziert wird, implementiert.

Die Implementierung I_QUELLPROG ermöglicht es dem implementierenden Datentyp IMPL_QUELLPROG, die Semantik des implementierten Datentyps QUELLPROG zu realisieren; d.h. der konkretere abstrakte Datentyp realisiert über die Implementierungs-Funktion den auf einer höheren Abstraktionsebene spezifizierten abstrakten Datentyp.

Dieser Implementierungsschritt zielt auf eine spätere Implementierung der Gesamtspezifikation in "PASCAL".
Die Listenstruktur, die in QUELLPROG spezifiziert wurde, in "PASCAL" aber nicht zur Verfügung steht, wird deshalb durch eine Stackkonstruktion, die in IMPL_QUELLPROG unter Zuhilfenahme des neu spezifizierten abstrakten Datentyps Z_ARRAY spezifiziert wird, realisiert.
Die in IMPL_QUELLPROG benutzten Unterstrukturen Array und "impliziter Pointer" (es wird kein abstrakter Datentyp POINTER explizit spezifiziert) sind in "PASCAL" standardmäßig vorhanden.

Unsere ursprüngliche Idee, die Listenstruktur durch eine kombinierte File-Array-Struktur



zu ersetzen, wurde verworfen, da in QUELLPROG eine

Zeichenfolge nach dem "LIFO"-Prinzip abgearbeitet wird, die "PASCAL"-Files aber nach dem "FIFO"-Prinzip arbeiten. Dieser grundlegende Unterschied erschien uns zur Implementierung zu aufwendig, sodaß wir die einfachere Implementierung durch ein Array mit Pointer realisieren.

In den folgenden Unterkapiteln werden zunächst die zur Implementierung benötigten abstrakten Datentypen Z_ARRAY und IMPL_QUELLPORG spezifiziert. Dann wird die eigentliche Implementierung I_QUELLPROG spezifiziert, die in Kapitel 8 schließlich als korrekt bewiesen wird.

7.1. Formale Spezifikation der Datentypen

7.1.1. Array von Zeichen

```
SSPEC Z _ A R R A Y

USE      SSPEC BOOL  SSPEC NAT  SSPEC ZEICHEN

PUBLIC SORTS    zarray

        OPS    leerarray :  —> zarray
                 schreibe :  zarray nat zeichen —> zarray
               vertausche :  zarray nat nat —> zarray
                 subarray :  zarray nat nat —> zarray
                     lese :  zarray nat —> zeichen
                   is_def :  zarray nat —> bool
                  is_leer :  zarray —> bool
                   ug, og :  zarray —> nat
                   ararg2 :  zarray —> nat

PROPERTIES

lese ( schreibe ( ar na ze ) na ) == ze
is_def ( schreibe ( ar na ze ) na ) == true

zq ( n1 n2 ) == false    ⇒
lese ( schreibe ( ar n1 ze ) n2 ) == lese ( ar n2 )

and ( is_def ( ar n1 ) is_def ( ar n2 ))    ⇒
vertausche ( vertausche ( ar n1 n2 ) n2 n1 ) == ar

ar ≠ *leerarray    ⇒
subarray ( ar  ug ( ar )  og ( ar )) == ar
```



```
and ( is_def ( ar na )  and ( na ≥ n1  na ≤ n2 ))    ⇒
is_def ( subarray ( ar n1 n2 ) na )

and ( is_def ( ar na )  and ( na ≥ n1  na ≤ n2 ))    ⇒
lese ( subarray ( ar n1 n2 ) na ) == lese ( ar na )

CONSTRUCTORS     *leerarray
                 *schreibe

AUXILIARIES      $arg_2_*schreibe

DEFINE CARRIER

is_array ( ar ) :=
case ar is *leerarray : true
           *schreibe ( a1 n1 z1 ) :
           case a1 is *leerarray : true
                      *schreibe ( a2 n2 z2 ) :
                                 and ( is_array ( a1 )
                                       lt ( n2 n1 ))
           esac
esac

DEFINE CONSTRUCTORS

leerarray := *leerarray

schreibe ( ar na ze ) :=
case ar is *leerarray : *schreibe (ar na ze )
           *schreibe (a1 n1 z1 ) :
     if lt ( n1 na )
        then *schreibe (*schreibe (a1 n1 z1 ) na ze )
        elseif zq ( na n1 )
           then schreibe ( a1 na ze )
           else schreibe ( schreibe (a1 na ze ) n1 z1 )
     fi
esac
```

DEFINE OPS

```
vertausche ( ar n1 n2 ) :=
schreibe ( schreibe ( ar n1 lese ( ar n2 ))
           n2 lese ( ar n1 ))


subarray ( ar n1 n2 ) :=
if or ( lt ( n2 n1 ) not ( and ( is-def ( ar n1 )
                                 is_def ( ar n2 ))))
   then error.array
   else case ar is *schreibe ( aa na ze ) :
              if lt ( n2 na )
                 then subarray ( aa n1 n2 )
                 elseif zq ( n1 na )
                        then schreibe ( leerarray na ze )
                        elseif zq ( n2 na )
                               then schreibe ( subarray (
                                      aa n1 ararg2 ( aa ))
                                               na ze )
                               else error.zarray
              fi
        esac
fi


lese ( ar na ) :=
case ar is *leerarray :  error.zeichen
           *schreibe ( a1 n1 z1 )
           if zq ( na n1 ) then z1
              elseif lt ( na n1 ) then lese ( a1 na )
                                  else error.zeichen
           fi
esac


is_def ( ar na ) :=
case ar is *leerarray :  false
           *schreibe ( a1 n1 z1 ) :
           if zq ( na n1 )
              then true
              elseif lt ( na n1 ) then is_def ( a1 na )
                                  else error.bool
           fi
esac
```



```
ug ( ar ) :=
case ar is *leerarray :  error.nat
           *schreibe (a1 n1 z1 ) :
           case a1 is *leerarray :  z1
                      otherwise     ug ( a1 )
           esac
esac


og ( ar ) :=
case ar is *leerarray :  error.nat
           *schreibe ( a1 n1 z1 ) :  n1
esac


is_leer ( ar ) :=
case ar is *leerarray :  true
           otherwise     false
esac


ararg2 ( ar ) :=
case ar is *leerarray :  error.nat
           otherwise     $arg-2-*schreibe ( ar )
esac
```

ENDSPEC

7.1.2. QUELLPROG implementierenden Datentyp

```
SSPEC   I M P L _ Q U E L L P R O G

USE       SSPEC Z_ARRAY

PUBLIC SORTS  imq

       OPS     paar      :  zarray nat—> imq
               imqarg1   : imq —> zarray
               imqarg2   : imq —> nat

               imnil     :  —> imq
               imhd      :  imq —> zeichen
               imtl      :  imq —> imq
               imcut     :  imq —> imq
               imlast    :  imq —> zeichen
               imlaenge  :  imq —> nat
               imhead    :  imq nat —> imq
               imtail    :  imq nat —> imq
               imcat     :  imq imq —> imq
               imcons    :  zeichen imq —> imq
               immbre    :  zeichen imq —> bool
               imsubst   :  imq imq nat —> imq
               imcmpr    :  imq imq nat —> bool


PROPERTIES

/*
alle PROPERTIES von QUELLPROG gelten auch für
IMPL-QUELLPROG entsprechend, wenn man den Operationsnamen aus
QUELLPROG jeweils die Silbe "im" voransetzt.
*/
```



```
CONSTRUCTORS      *paar

AUXILIARIES       $arg_1_*paar
                  $arg_2_*paar

DEFINE OPS

imnil := paar ( leerarray null )

imcons ( ze iq ) :=
case iq is *paar ( ar na ) :
     if is_leerarray ( ar )
        then paar ( schreibe ( leerarray  succ ( null ) ze )
                    succ ( na ))
        else paar ( schreibe ( ar plus ( ug ( ar ) na ) ze )
                    succ ( na ))
     fi
  otherwise  error.imq
esac

imhd ( iq ) :=
case iq is *paar ( ar na ) :
     if zq ( na null )
        then error.imq
        else lese ( ar pred ( plus ( ug ( ar ) na )))
     fi
esac

imtl ( iq ) :=
case iq is *paar ( ar na ) :
     if zq ( na null )
       then iq
       else paar ( ar pred ( na ))
     fi
  otherwise error.imq
esac
```

```
imlast ( iq ) :=
case iq is *paar ( ar na ) :
     if zq ( na null ) then error.zeichen
                       else lese ( ar ug ( ar ))
     fi
esac

imcut ( iq ) :=
case iq is *paar ( ar na ) :
     if zq ( na null)
        then iq
        elseif zq ( na succ ( null ))
               then paar ( ar null )
               else paar ( subarray ( ar succ ( ug ( ar ))
                           og ( ar )) pred ( na ))
     fi
esac

imcat ( i1 i2 ) :=
case i1 is *paar ( ar na ) :
     if zq ( na null )
        then i2
        else imcons ( imhd ( i1 ) imcat ( imtl ( i1 ) i2 ))
     fi
esac

immbre ( ze iq ) :=
case iq is *paar ( ar na ) :
     if zq ( na null )
        then false
        elseif eq ( ze imhd ( iq ))
               then true
               else immbre ( ze imtl ( iq ))
     fi
esac

imsubst ( i1 i2 na ) := imcat ( i2 imtail ( i1 na ))
```



```
imhead ( iq na ) :=
case iq is *paar ( ar nb ) :
     if zq ( na null )
        then imnil
        elseif lt ( nb na )
           then error.qprog
           else paar ( subarray ( ar
                                  sub ( plus ( ug ( ar )
                                               nb ) na )
                                  og ( ar ))
     fi
esac

imtail ( iq na ) :=
case iq is *paar ( ar nb ) :
     if lt ( nb na )
        then paar ( ar null )
        else paar ( ar sub ( nb na ))
     fi
esac

imcmpr ( i1 i2 na ) :=
case i1 is *paar ( a1 n1 ) :
    case i2 is *paar ( a2 i2 ) :
        if zq ( na null )
          then true
          elseif or ( zq ( n1 null ) zq ( n2 null ))
                then false
                else and ( eq ( imhd ( i1 ) imhd ( i2 ))
                           imcmpr ( imtl ( i1 ) imtl ( i2 )
                                    pred ( na )))
        fi
    esac
esac

imlaenge ( iq ) :=
case iq is *paar ( ar na ) : na    esac
```

<u>ENDSPEC</u>

7.2. Erläuterungen zu den Datentypen

7.2.1. SSPEC Z_ARRAY (Zeichen-ARRAY)

Der Datentyp Z_ARRAY spezifiziert als neue Sorte 'zarray', die grundlegen Datenstruktur der Implementierung.
Jedes Element der Sorte 'zarray' ist zu interpretieren als Array im bekannten Sinn, dessen Elemente von der Sorte 'zeichen' sind.

Die Operationen leerarray und schreibe sind die Konstruktoren in Z_ARRAY.

leerarray (LEERes ARRAY) erzeugt ein leeres Array.

schreibe (SCHREIBE in Array) schreibt ein bestimmtes Zeichen an eine bestimmte Position in ein Array.

vertausche (VERTAUSCHE zwei Array-Elemente) vertauscht zwei bestimmte Array-Elemente.

subarray (SUB-ARRAY) liefert als Ergebnis ein genau abgegrenztes Teilarray des betrachteten Arrays.

lese (LiES Ein Zeichen) liest das Zeichen, das an einer bestimmten Stelle in dem Array steht.

is_def (ISt die Array-Position DEFiniert ?) überprüft, ob an einer bestimmten Position des Arrays ein beliebiges Zeichen steht.

is_leer (ISt das Array LEER ?) überprüft, ob das Array an mindestens einer Position definiert ist.

ug (Untere Grenze) liefert den Index der niedrigsten Position, die definiert ist.

og (Obere Grenze) liefert den Index der höchsten Position, die definiert ist.

ararg2 (ARray-ARGument-2) liefert als Ergebnis die zweite Komponente eines Terms der Sorte 'zarray', d.h. den Index der höchsten schreibe-Operation. ararg2 wird durch die Hilfsoperation $arg-2-schreibe realisiert.

Da nur "wohlgeordnete" Arrays zu dem Carrier der Sorte 'zarray' gehören sollen, wird das Herbrand-Universum über das Prädikat is_array auf die gewünschte Untermenge eingeschränkt (siehe DEFINE CARRIER in Z-ARRAY).
Wohlgeordnet sind der Term *leerarray und alle Terme, die mit einer streng monotonen Indexfolge aufgebaut sind.

Beispiele :
a) *schreibe(*schreibe(*leerarray 3 *a) 7 *b) ∊ 'zarray'
b) *schreibe(*schreibe(*leerarray 7 *a) 3 *b) ∉ 'zarray'
c) *schreibe(*schreibe(*leerarray 5 *a) 5 *b) ∉ 'zarray'

mit
- 3, 5 und 7 repräsentieren die entsprechenden *succ-Terme aus NAT
- Indexfolge(a) == 3,7
- Indexfolge(b) == 7,3
- Indexfolge(c) == 5,5

Da der Carrier von 'zarray' explizit definiert ist, müssen auch die Konstruktoren explizit angegeben werden (siehe DEFINE CONSTRUCTORS in Z_ARRAY).

Die Operation leerarray liefert den Term *leerarray. Die Definition von schreibe erzwingt den Aufbau wohlgeordneter Arrays. Soll an eine Position geschrieben werden, deren Index kleiner als der höchste Index des aktuellen Arrays ist, so wird der Term rekursiv durchlaufen bis die richtige Position erreicht ist; dort wird dann der Term um den neuen Eintrag erweitert.
Soll an eine Position geschrieben werden, die schon definiert ist, so wird der alte Eintrag durch das neue Zeichen ersetzt; es können also keine doppelten Einträge vorkommen.

7.2.2. IMPL_QUELLPROG (IMPLementiert QUELLPROG)

Der Datentyp IMPL_QUELLPROG soll den Datentyp QUELLPROG implementieren. Es wird die neue Sorte 'imq' (IMplementiert 'Qprog') definiert.

Elemente von 'imq' sind Zwei-Tupel, deren erste Komponenten Elemente von 'zarray' und deren zweite Komponenten Elemente von 'nat' sind.
Bildlich gesprochen, repräsentiert die zweite Komponente eines Elementes der Sorte 'imq' einen Pointer auf eine Position des Arrays in der ersten Komponente. Elemente aus 'imq' können also in einem eingeschränkten Sinn als Stacks interpretiert werden.

Hierbei ist zu beachten, daß bei dem Aufbau der Sorte 'imq' durch den Konstruktor <u>paar</u> keinerlei Anforderungen an die Korrelation zwischen Array und Pointer gestellt werden. Die Idee, die dem Implementierungsschritt zugrunde liegt, nämlich das Ersetzen der Listenstruktur durch eine Stackstruktur, die durch ein Array und einen Pointer, der auf das oberste Element des Arrays zeigt, realisiert ist, ist in IMPL_QUELLPROG noch nicht vollständig verwirklicht. Erst bei der Implementierung durch I_QUELLPROG wird mit Hilfe der Repräsentationsfunktion <u>rep.qprog</u> dieses Ziel erreicht.
Die Operationen von IMPL-QUELLPROG liefern deshalb in Abhängigkeit ihrer Argumente teilweise Resultate, die nicht der Intuition gerecht werden. Die Operation <u>imlaenge</u> , die später die Operation <u>laenge</u> aus QUELLPROG implementieren soll, kann z.B. als Resultat eine Zahl größer Null liefern, obwohl das zugehörige Array leer ist, da z.B. der Term *paar(*leerarray *succ(*null)) ein Element der Sorte 'imq' ist.

Die Operation <u>paar</u> ist der Konstruktor von 'imq'.

<u>paar</u> (erzeuge ein PAAR) macht aus einem Array der Sorte 'zarray' und einer Zahl der sorte 'nat' ein Paar bzw. ein Zwei-Tupel.
Eine Beziehung zwischen dem Wert der zweiten Komponenten (Pointer) und dem Array in der ersten Komponente wird nicht erzwungen. Der Term
*paar(*schreibe(*leerarray *succ(*null) *z) *null)
ist zum Beispiel Element der Sorte 'imq'.

<u>imqarg1</u> ('IMQ'-ARGument-1) liefert als Ergebnis die erste Komponente eines Elementes der Sorte 'imq', das Array des Paares.

<u>imqarg2</u> ('IMQ'-ARGument-2) liefert als Ergebnis die zweite Komponente eines Elementes der Sorte 'imq', den Pointer des Paares.

Alle anderen Operationen entsprechen in ihrer Leistung den jeweiligen Operationen aus QUELLPROG (5.4.).
Die Namen dieser Operationen sind aus den Operationsnamen von QUELLPROG abgeleitet, um diese Beziehung deutlich zu machen.
Beispiele : <u>imcut</u> soll <u>cut</u> implementieren
<u>imhead</u> soll <u>head</u> implementieren

Den üblichen Stackoperationen <u>push</u> und <u>pop</u> entsprechen also hier die Operationen <u>imcons</u> und <u>imtl</u> .

Die Wirkungen der Operationen in IMPL_QUELLPROG entsprechen den Wirkungen der Operationen in QUELLPROG, wenn man die in 6.7.4. angegebenen Beschreibungen nicht auf Listen sondern auf Stacks mit den oben beschriebenen Einschränkungen bezieht.

Einige Definitionen dieser Operationen haben wegen der geänderten zugrundeliegenden Datenstruktur nur wenig oder gar nichts mehr gemeinsam mit den Algorithmen der entsprechenden Operationen in QUELLPROG.
Der Grund liegt in der Benutzung des Pointers und der Operation <u>subarray</u> von Z-ARRAY.
Bei einigen Operationen wird nur noch der Wert des Pointers verändert (<u>imtl</u> , <u>imtail</u>).
Das Ergebnis von <u>imlaenge</u> ist eindeutig bestimmt durch den Wert des Pointers.

7.3. Formale Spezifikation der Implementierung I_QUELLPROG

```
ISPEC   I _ Q U E L L P R O G

of    QUELLPROG
by    IMPL_QUELLPROG

base    SSPEC NAT   SSPEC ZEICHEN


implement

    sorts  qprog    by   imq

      ops    nil    by   imnil
              hd    by   imhd
            last    by   imlast
              tl    by   imtl
             cut    by   imcut
          laenge    by   imlaenge
            head    by   imhead
            tail    by   imtai
             cat    by   imcat
            cons    by   imcons
            mbre    by   immbre
           subst    by   imsubst
            cmpr    by   imcmpr
```

```
define representation

/*
Für alle Elemente iq der Sorte imq, mit iq ≠ error.imq, gilt :
*/

rep.qprog ( iq ) :=

case iq is *paar ( ar null ) : nil
           *paar ( ar na )   :
    if not ( lt ( succ ( minus ( og ( ar ) ug ( ar ))) na ))
      then
        if is_def ( ar pred ( plus ( ug ( ar ) na )))
          then cons ( imhd ( iq ) rep.qprog ( imtl ( iq )))
          else error.qprog
        fi
      else error.qprog
    fi
esac

ENDSPEC
```


7.4. Erläuterungen zur Implementierung

I_QUELLPROG (Implementierung von QUELLPROG) spezifiziert die Implementierung des abstrakten Datentyps QUELLPROG durch den abstrakten Datentyp IMPL_QUELLPROG.

Die Basis der Implementierung wird durch NAT und ZEICHEN gebildet, den beiden Datentypen, die sowohl von QUELLPROG alsauch von IMPL_QUELLPROG benutzt werden.

Die Syntax der Implementierung ist durch <u>ISPEC</u> I_QUELLPROG gegeben.

Die Semantik ist gegeben durch das Implementierungs-Tripel IT, das durch I_QUELLPROG folgendermaßen bestimmt ist :

IT = (QUELLPROG,(<u>implement</u> , <u>rep.qprog</u>),IMPL_QUELLPROG)

QUELLPROG ist die abstrakte Spezifikation.

IMPL_QUELLPROG ist die konkrete Spezifikation, die QUELLPROG implementieren soll.

<u>implement</u> ist der Morphismus von der abstrakten Sorte und den abstrakten Operationen in QUELLPROG in die konkrete Sorte und die konkreten Operationen in IMPL_QUELLPROG.

<u>rep.qprog</u> ist eine Funktion, durch die mit Ausnahme des Fehlerelements error.qprog jedem Element der implementierenden Sorte 'imq' eindeutig ein Element der Sorte 'qprog' zugeordnet wird :

<u>rep.qprog</u> : <u>implement</u> ('qprog') ——> 'qprog'

oder einfacher

<u>rep.qprog</u> : 'imq' ——> 'qprog' .

rep.qprog ordnet genau den Elementen aus 'imq' das Fehlerelement error.qprog der Sorte 'qprog' zu, für die mindestens eine der folgenden Aussagen gilt :

- der Wert des Pointers ist größer als die Differenz von oberer und unterer Grenze des Arrays plus Eins
- es gibt nicht-definierte Positionen in dem Array, deren Index größer ist als die untere Grenze des Arrays und kleiner gleich ist der Summe des Pointerwerts und der unteren Grenze des Arrays .

Die teilweisen Unzulänglichkeiten der Wirkungen einiger Operationen von IMPL_QUELLPROG, auf die in 7.2.2. hingewiesen wurde, sind nun ausgeschaltet, denn für alle Elemente von 'imq', die nicht auf error.qprog abgebildet werden, sind die Bedingungen, die an einen Stack, der durch ein Array und einen Pointer realisiert ist, gestellt werden, erfüllt :

- die Arrays sind wohlgeordnet
- die Arrays haben keine Lücken
- alle über den Pointer erreichbaren Positionen des Arrays sind definiert .

8. KORREKTHEITSBEWEIS DER IMPLEMENTIERUNG I_QUELLPROG

8.1. Notation und Schreibweise

Im allgemeinen wird bei der Umformung von Termen während der einzelnen Beweisschritte die Präfixnotation der im Datentyp QUELLPROG bzw. der im Datentyp IMPL_QUELLPROG benutzten Operationen beibehalten. Es werden auch teilweise in den Datentypen nicht eingeführte Operationen verwendet, auf deren Definition hier verzichtet wird, da diese allgemein bekannt sind (z.B. "le" in Bedeutung von ≤).

Bei allgemeinen Betrachtungen , die sich nicht ausschließlich auf Term- Algebren beziehen, kann die Infixnotation benutzt werden (z.B. "a+b" statt "plus(a b)").

Kurze Bemerkungen , die zu einzelnen Ausdrücken innerhalb der Beweisschritte gemacht werden, stehen jeweils rechts neben dem Gleichheitszeichen, das zu dem entsprechenden Ausdruck führt

```
(z.B.: expression A
            =            / A führt zu B
       expression B                          ).
```

Längere Erläuterungen zu einzelnen Beweisschritten werden als Kommentar gekennzeichnet

```
(z.B.: expression A
       =
       /* A geht über in B wegen Lemma xyz ... */
       =
       expression B                                        ).
```

8.2. Generelle Voraussetzung

Für alle Lemmata und Beweise, die sich im folgenden auf die Implementierung I_QUELLPROG beziehen, gelte :

1.) $$\varrho(x) = \begin{cases} \text{rep.qprog}(x) & \text{, falls } x \in \text{imq} \\ x & \text{,sonst} \end{cases}$$

2.) $\varrho(x) \neq$ error.qprog ,
es sei denn, daß diese Alternative explizit in die Betrachtung mit aufgenommen wird.

3.) Sei x ein "fehler", dann ist $\varrho(x)$ wie folgt definiert :

ϱ(error.bool) = error.bool
ϱ(error.nat) = error.nat
ϱ(error.zeichen) = error.zeichen
ϱ(error.imq) = error.qprog

4.) Der Sortenname eines Datentypes bezeichnet auch die Menge aller Elemente dieser Sorte.
(z.B.: x ist von der Sorte y $\Leftrightarrow$ x $\in$ y)

8.3. Einführung benötigter Lemmata

8.3.1. Lemma 1

$\forall$iq$\in$imq\{error.imq}. $\forall$ar$\in$zarray\{error.zarray, *leerarray}.
iq = *paar(ar na).rep.qprog(iq) $\neq$ error.qprog.:

le(na succ(sub(og(ar) ug(ar))))

/*Das Lemma sagt aus, daß der "Pointer" eines nichttrivialen Elementes der Sorte imq nicht größer als 1 + (Differenz zwischen Ober- und Untergrenze des Arrays) sein kann.
Diese Summe gibt die Maximal- Anzahl der definierten Arraypositionen an. */

<u>Beweis :</u>
(Der Beweis wird indirekt geführt)

Annahme:

Es gebe ein iq$\in$imq\{error.imq}. iq = *paar(ar na)
$\wedge$ ar $\neq$ *leerarray. rep.qprog(iq) $\neq$ error.qprog , für das gelte :
not(le(na succ(sub(og(ar) ug(ar))))).

not(le(na(succ(sub(og(ar) ug(ar)))))

= /nach arithm. Umformung

lt(succ(sub(og(ar) ug(ar)))na)

Nach Definition von rep.qprog werden jedoch alle iq $\in$ imq , für die die Aussage "lt(succ(sub(og(ar) ug(ar)))na)" gilt, auf error.qprog abgebildet.

Dies ist ein Widerspruch zur Annahme. qed

8.3.2. Lemma 2

```
∀iq∈imq\{error.imq}. ∀ar∈zarray\{error.zarray, *leerarray}.
∀ze∈zeichen\{error.zeichen}. iq = *paar(ar na).
rep.qprog(iq) ≠ error.qprog.:

le(succ(na) succ(sub(og(schreibe(ar plus(ug(ar) na) ze))
                     ug(schreibe(ar plus(ug(ar) na) ze)))))
```

/*Lemma 2 ist eine Erweiterung von Lemma1. */

Beweis :

Es sei: nb = plus(ug(ar) na),
dann gilt aus der Definition von ug(ar) und og(ar) :

```
ug(ar) = ug(schreibe(ar nb ze))
og(ar) ≤ og(schreibe(ar nb ze)).
```

Daraus folgt direkt:

```
sub(og(ar) ug(ar)) ≤ sub(og(schreibe(ar nb ze))
                         ug(schreibe(ar nb ze)))
  ⇔

succ(sub(og(ar) ug(ar))) ≤
  succ(sub(og(schreibe(ar nb ze)) ug(schreibe(ar nb ze))))
```

/*Es werden nun folgende zwei Fälle unterschieden : */

Fall 1:

```
succ(sub(og(ar) ug(ar))) =
  succ(sub(og(schreibe(ar nb ze)) ug(schreibe(ar nb ze))))
```

Da ug(ar)=ug(schreibe(ar nb ze)) folgt für diesen Fall:

```
og(ar) = og(schreibe(ar nb ze))

  ⇒                                         /da og(ar) ≥ nb

og(ar) ≥ plus(ug(ar) na)

  ⇒                                           /nach Lemma1

lt(na succ(sub(og(ar) ug(ar))))

  ⇔

lt(na succ(sub(og(schreibe(ar nb ze))
               ug(schreibe(ar nb ze)))))

  ⇒

le(succ(na) succ(sub(og(schreibe(ar nb ze))
                     ug(schreibe(ar nb ze)))))
```

/*Ende des Beweises zu Fall 1. */

<u>Fall 2</u>:

```
succ(sub(og(ar) ug(ar))) <
  succ(sub(og(schreibe(ar nb ze)) ug(schreibe(ar nb ze))))
```

Aus dieser ("Kleiner-") Relation ergibt sich nach Umformungen und unter Anwendung von Lemma1:

```
og(ar) = pred(plus(ug(ar) na))


og(schreibe(ar nb ze)) = plus(ug(ar) na) = nb = succ(og(ar)).
```

Nach Lemma 1 gilt:
le(na succ(sub(og(ar) ug(ar)))).

In dem hier betrachteten Fall gilt sogar die Gleichheit (zq) :

```
zq(na succ(sub(og(ar) ug(ar))))

  ⇔                                          /Arithmetik

zq(succ(na) succ(succ(sub(og(ar) ug(ar)))))

  ⇔                                          /Arithmetik

zq(succ(na) succ(sub(succ(og(ar)) ug(ar))))

  ⇔                                /siehe obige Umformung

zq(succ(na) succ(sub(og(schreibe(ar nb ze))
                     ug(schreibe(ar nb ze)))))          qed
```

/*Dieses Lemma hat seine Bedeutung bei der Anwendung der Operation imcons, da dann an die "Position" plus(ug(ar) na) "geschrieben" und der "Pointer" um 1 erhöht wird. */

8.3.3. Lemma 3

```
∀iq∈imq\{error.imq}. ∀ar∈zarray\{error.zarray}.
∀ze∈zeichen\{error.zeichen}. iq = *paar(ar na).
na ≠ *null. rep.qprog(iq) ≠ error.qprog.:

is_def(schreibe(ar plus(ug(ar) na) ze)
       pred(plus(ug(schreibe(ar plus(ug(ar) na) ze))
                 succ(na))))
  = true
```

/*Das Lemma sagt aus, daß, wenn ein Element iq der Sorte imq um ein Element "vorne" erweitert wird (z.B. durch eine Anwendung von imcons), dann auch der zu iq gehörende Array an der entsprechend aktuellen "Position" definiert ist. */

<u>Beweis</u> :

Es gilt :

```
is_def(ar pred(plus(ug(ar) na)))  =  true
```

/*Der Beweis dafür folgt direkt aus der Definition von rep.qprog, denn für
is_def(ar pred(plus(ug(ar) na))) = false
ergibt sich
rep.qprog(iq) = error.qprog, was ein Widerspruch zur Voraussetzung zu diesem Lemma ist. */

Ebenso gilt:

```
is_def(schreibe(ar plus(ug(ar) na) ze) plus(ug(ar) na))

   =                      /nach Definition von is_def in ARRAY

true
```

```
/*Da ug(ar) = ug(schreibe(ar plus(ug(ar) na) ze))   folgt:    */

is_def(schreibe(ar plus(ug(ar) na) ze)
       plus(ug(schreibe(ar plus(ug(ar) na) ze)) na)) = true

  ⇔                                                /Arithmetik

is_def(schreibe(ar plus(ug(ar) na) ze)
       pred(plus(ug(schreibe(ar plus(ug(ar) na) ze))
                 succ(na))))                         = true

                                                           qed
```

/*Äquivalent zu Lemma 2 findet Lemma 3 seine Bedeutung bei Anwendung der Operation imcons.
Die Lemmata 1, 2 und 3 entsprechen den einzelnen Alternativen von rep.qprog. */

Die Elemente der Sorte imq kann man sich als Stacks vorstellen und die Elemente der Sorte qprog als lineare Listen.
Ein iq∈imq\{error.imq} wird beschrieben durch ein Paar, dessen erste Komponente ein Element der Sorte Array und dessen zweite Komponente ein Element der Sorte nat ist.
Dieses Tupel kann als Implementierung eines Stacks durch einen Array mit Pointer betrachtet werden.

Die Repräsentationsfunktion rep.qprog ordnet nun jedem "stack" eine entsprechende Liste derart zu, daß alle Elemente des Arrays (von der Sorte zeichen), die über den Pointer erreichbar sind, zu einer Liste mit der gleichen Ordnung wie innerhalb des Arrays zusammengefaßt werden.

Daraus folgt nun, daß verschiedene iq, iq' ∈ imq, deren Arrays in allen relevanten "Positionen" übereinstimmen, auf dasselbe Element der Sorte qprog abgebildet werden, auch wenn sich die Elemente iq und iq' in allen anderen Positionen unterscheiden (Belegungen relevanter "Positionen" sind Elemente der Sorte zeichen, auf die in einem Element iq, bzw. iq' über den "Pointer" zugegriffen werden kann).

Der folgende Satz trägt dieser Tatsache Rechnung:

8.4. Hauptsatz der Implementierung

Vorraussetzung VS1 :

```
-  Es sei iq  = *paar(ar na)
      und iq' = *paar(ar' na)   mit na∈nat\{error.nat}.
-  Es sei ϱ(iq)  ≠ error.x
      und ϱ(iq') ≠ error.x
                    mit x∈{bool, nat, zeichen, qprog}.
```

Aussagen des Satzes :

```
a.) VS1 ∧ na = *null.:    ϱ(iq) = ϱ(iq')

b.) VS1 ∧ na > null ∧ ∀k∈[0 : na-1].
        lese(ar plus(ug(ar) k)) = lese(ar' plus(ug(ar') k)).:

    ϱ(iq) = ϱ(iq')
```

/*Teil a.) des Satzes sagt aus, daß alle Elemente aus imq, deren zweite Komponente 0 ist (sozusagen alle "leeren stacks"), auf dasselbe Element aus qprog abgebildet werden, nämlich auf nil.

Teil b.) des Satzes sagt aus, daß alle "korrekten" Elemente (Bedingung VS1 ist erfüllt!) der Sorte imq, die in allen "relevanten" (über den "Pointer" erreichbaren) "Positionen" mit jeweils identischen Elementen der Sorte zeichen "belegt" sind, und deren "Pointer" (zweite Komponente) ebenfalls identisch sind, auf genau dasselbe Element aus qprog unter ϱ abgebildet werden (d.h.: "gleiche stacks" werden auf "einunddieselbe Liste" abgebildet). */

<u>Beweis</u> :

zu a.)

```
VS1 ∧ na = *null.:    (iq) =  (iq').

d.h.:
iq  = *paar(ar *null)
iq' = *paar(ar' *null).


ϱ(iq) = ϱ(*paar(ar *null)

  =                             /nach Def. von ϱ und wegen VS1

rep.qprog(*paar(ar *null))  =  nil    /nach Def. von rep.qprog



ϱ(iq') = ϱ(*paar(ar' *null))

  =                             /nach Def. von ϱ und wegen VS1

rep.qprog(*paar(ar' *null)) =  nil    /nach Def. von rep.qprog
```


zu b.)

Induktionsbeweis über na

<u>Induktionsanfang</u> (na = *succ(*null))

Es sei na = *succ(*null), dann ist zu zeigen:

```
VS1 ∧ lese(ar ug(ar)) = lese(ar' ug(ar')).: ϱ(iq) = ϱ(iq')


ϱ(iq) = ϱ(*paar(ar *succ(*null)))

  =                                  /nach Def. von ϱ und wegen VS1

rep.qprog(*paar(ar *succ(*null)))

  =      /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cons(imhd(*paar(ar *succ(*null)))
     rep.qprog(imtl(*paar(ar *succ(*null)))))

  =                                              /nach Def. von imhd

cons(lese(ar pred(plus(ug(ar) *succ(*null))))
     rep.qprog(imtl(*paar(ar *succ(*null)))))

  =

/*Sei [redacted]1[redacted]:
  ze = lese(ar pred(plus(ug(ar) *succ(*null))))              */

  =

cons(ze rep.qprog(imtl(*paar(ar *succ(*null)))))

  =                                              /nach Def. von imtl

cons(ze rep.qprog(*paar(ar *null)))

  =                                         /nach Def. von rep.qprog

cons(ze nil)
```

```
ϱ(iq') = ϱ(*paar(ar' *succ(*null)))

  =                                    /nach Def. von ϱ und wegen VS1

rep.qprog(*paar(ar' *succ(*null)))

  =       /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cons(imhd(*paar(ar' *succ(*null)))
     rep.qprog(imtl(*paar(ar' *succ(*null)))))

  =                                               /nach Def. von imhd

cons(lese(ar' pred(plus(ug(ar') *succ(*null))))
     rep.qprog(imtl(*paar(ar' *succ(*null)))))

  =

/*Arithmetische Umformungen ergeben:

  pred(plus(ug(ar') *succ(*null))) = ug(ar')
  und
  pred(plus(ug(ar) *succ(*null)))  = ug(ar)

  nach [redacted] gilt :
  ze = lese(ar pred(plus(ug(ar) *succ(*null)))) =
       lese(ar ug(ar))

  Nach der Hypothese aus der Konklusion von Teil b.) des Satz-
  es gilt:
  lese(ar ug(ar))  =  lese(ar' ug(ar'))

  Somit ergibt sich also auch:
  lese(ar' pred(plus(ug(ar') *succ(*null)))) = ze .
  Dies ermöglicht die Ableitung zu folgendem Ausdruck :       */

  =

cons(ze rep.qprog(imtl(*paar(ar' *succ(*null)))))

  =                                               /nach Def. von imtl

cons(ze rep.qprog(*paar(ar' *null)))

  =                                          /nach Def. von rep.qprog

cons(ze nil)
```


<u>Induktionsannahme</u>   (*null < na ≤ nb)

Folgendes sei bewiesen für *null < na ≤ nb:

```
VS1 ∧ ∀naεnat\{error.nat}. *null < na ≤ nb. ∀kε[0 : nb-1].
lese(ar plus(ug(ar) k)) = lese(ar' plus(ug(ar') k)).
∀iq,iq'εimq\{error.imq}. iq = *paar(ar na).
iq' = *paar(ar' na).:
                  ϱ(iq) = ϱ(iq')
```

<u>Induktionsschritt</u>   (na = *succ(nb))

Für na = *succ(nb) ist zu zeigen :

```
VS1 ∧ ∀arεzarray\{error.zarray, *leerarray}.
∀naεnat\{error.nat}. na = *succ(nb). ∀iε[0 : nb].
lese(ar plus(ug(ar) i)) = lese(ar' plus(ug(ar') i)).
∀iq,iq'εimq\{error.imq}. iq = *paar(ar na).
iq' = *paar(ar' na).:
                   ϱ(iq) = ϱ(iq')


ϱ(iq) = ϱ(*paar(ar na)) =

ϱ(*paar(ar *succ(nb)))

  =                                    /nach Def. von ϱ und wegen VS1

rep.qprog(*paar(ar *succ(nb)))

  =       /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cons(imhd(*paar(ar *succ(nb)))
     rep.qprog(imtl(*paar(ar *succ(nb)))))

  =                                               /nach Def. von imhd

cons(lese(ar pred(plus(ug(ar) *succ(nb))))
     rep.qprog(imtl(*paar(ar *succ(nb)))))

  =
```

```
/*Nach VS1 gilt :  ϱ(iq) ≠ error.x; daraus folgt dann:
  is_def(ar pred(plus(ug(ar) *succ(nb)))) = true , d.h.,
  daß an der "Position" pred(plus(ug(ar) *succ(nb))) ein
  Element der Sorte zeichen "steht".Dieses sei ze'.

  Im folgenden sei ■■2■■ definiert:
  ze' = lese(ar pred(plus(ug(ar) *succ(nb))))
  Somit läßt sich folgender Ausdruck ableiten:             */

  =

cons(ze' rep.qprog(imtl(*paar(ar *succ(nb)))))

  =                                          /nach Def. von imtl

cons(ze' rep.qprog(*paar(ar nb)))


ϱ(iq') = ϱ(*paar(ar' na)) =

ϱ(*paar(ar' *succ(nb)))

  =                               /nach Def. von ϱ und wegen VS1

rep.qprog(*paar(ar' *succ(nb)))

  =     /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cons(imhd(*paar(ar' *succ(nb)))
     rep.qprog(imtl(*paar(ar' *succ(nb)))))

  =                                          /nach Def. von imhd

cons(lese(ar' pred(plus(ug(ar') *succ(nb))))
     rep.qprog(imtl(*paar(ar' *succ(nb)))))

  =
```



```
/*arithmetische Umformungen ergeben :

  pred(plus(ug(ar') *succ(nb))) = plus(ug(ar' nb))
  und
  pred(plus(ug(ar) *succ(nb)))  = plus(ug(ar nb))

  Nach der Hypothese aus der Konklusion des Induktions-
  schrittes gilt :
  ∀k∈[0 : nb]. lese(ar plus(ug(ar) k))
                     = lese(ar' plus(ug(ar') k))
  und damit insbesonders :

  lese(ar plus(ug(ar) nb))  =  lese( ar' plus(ug(ar') nb))

  Aus ■■2■■ und dieser Betrachtung ergibt sich :

  ze' = lese(ar plus(ug(ar) nb))
      = lese(ar' plus(ug(ar') nb))
  Somit läßt sich folgender Ausdruck ableiten :              */

  =

cons(ze' rep.qprog(imtl(*paar(ar' *succ(nb)))))

  =                                          /nach Def. von imtl

cons(ze' rep.qprog(*paar(ar' nb)))
```

Somit ergibt sich also bis hierhin:

```
ϱ(iq)  = cons(ze' rep.qprog(*paar(ar nb)))
ϱ(iq') = cons(ze' rep.qprog(*paar(ar' nb)))
```

Aufgrund der Induktionsannahme gilt jedoch :

```
ϱ(*paar(ar nb)) = ϱ(*paar(ar' nb))
```

unter der Bedingung

```
VS1 ∧ na = nb ∧ ∀k∈[0 : nb-1].
        lese(ar plus(ug(ar) k)) = lese(ar' plus(ug(ar') k)).
```

Diese Bedingung ist für iq und iq' in der Konklusionshypothese der Induktionsannahme vorausgesetzt (sollte die Bedingung nicht erfüllt sein, dann ist die Konklusion trivialerweise "wahr").

Somit ergeben sich folgende Äquivalenzen :

```
rep.qprog(*paar(ar nb))
  =                                 /da *paar(ar nb) ∊ imq
ϱ(*paar(ar nb))
  =                                 /nach Induktionsannahme
ϱ(*paar(ar' nb))
  =                      /nach Def. von ϱ und wegen VS1
rep.qprog(*paar(ar'nb))
```

Somit gilt dann auch :

```
ϱ(iq) = ϱ(*paar(ar *succ(nb)))
  =                                 /als Ableitungsergebnis
cons(ze' rep.qprog(*paar(ar nb)))
  =                        /nach oben gezeigter Äquivalenz
cons(ze' rep.qprog(*paar(ar' nb)))
  =                                 /als Ableitungsergebnis
ϱ(*paar(ar' *succ(nb)))
  =
ϱ(iq')
                                                      qed
```

/*Ende des Beweises zum Hauptsatz der Implementierung. */

8.5. Beweisskizze

Mit Hilfe der drei Lemmata und des einen Satzes wird nun im folgenden für jede Operation f des Datentyps IMPL_QUELLPROG gezeigt, daß gilt :

$\forall iq \in imq.\ iq = *paar(ar\ na) \wedge \varrho(iq) \neq error.x$ mit $x \in \{bool, nat, zeichen, qprog\}$:

$\varrho(f(iq)) = f'(\varrho(iq))$,

mit f' ist die Operation des Datentyps QUELLPROG, die bei der Implementierung I_QUELLPROG f zugeordnet wird.

Der Beweis wird nun für jede einzelne Operation f explizit durchgeführt und der Name der jeweiligen Operation f dem entsprechenden Beweis als Überschrift vorangestellt.

Wenn für alle Operationen f des Datentyps IMPL_QUELLPROG und den ihnen zugeordneten Operationen f' des Datentyps QUELLPROG gezeigt ist, daß folgendes Diagramm kommutiert, dann ist auch die Implementierung I_QUELLPROG als korrekt bewiesen.

Diagramm

$$\begin{tikzcd} iq \arrow[r, "f"] \arrow[d, "\varrho"'] & iq' \arrow[d, "\varrho"] \\ qp \arrow[r, "f'"'] & qp' \end{tikzcd}$$

8.6. Korrektheitsbeweis der Operationen

8.6.1. IMNIL

zu zeigen:

ϱ(imnil) = nil

<u>Beweis :</u>

```
ϱ(imnil)
  =                                     /nach Def. von imnil
ϱ(paar(leerarray null))
  =           /null evaluiert zu *null, leerarray zu *leerarray
ϱ(paar(*leerarray *null))
  =                             /äquivalent zum letzten Schritt
ϱ(*paar(*leerarray *null))
   =                                         /nach Def. von ϱ
rep.qprog(*paar(*leerarray *null))
  =                                  /nach Def. von rep.qprog
nil                                                       qed
```

/*Auf eine so ausführliche Betrachtungsweise der Evaluierung von Ausdrücken zu den entsprechenden Termen wird bei den folgenden Beweisen teilweise verzichtet, jedoch nur dann, wenn die entsprechenden Ableitungsschritte unmittelbar einsichtig sind.

Ende des Beweises bzgl. imnil. */

8.6.2. IMCONS

zu zeigen :

```
∀iq∊imq\{error.imq}. ∀ze∊zeichen\{error.zeichen}.
ϱ(iq) ≠ error.qprog.:
                 ϱ(imcons(ze iq)) = cons(ϱ(ze) ϱ(iq))   .
```

<u>Beweis :</u>

o.B.d.A. sei iq = *paar(ar na)

Induktionsbeweis über na

<u>Induktionsanfang</u> (na = *null)

```
zu zeigen :
∀iq∊imq\{error.imq}. iq = *paar(ar na). na = *null.
∀ze∊zeichen\{error.zeichen}.:

ϱ(imcons(ze iq)) = cons(ϱ(ze) ϱ(iq))

ϱ(imcons(ze iq)) =

ϱ(imcons(ze *paar(ar *null)))

=
```

/*Um imcons ausführen zu können, müssen nun erst folgende beiden Fälle unterschieden werden :

- Fall 1 : ar = *leerarray
- Fall 2 : ar = schreibe(ar′ st ze′) mit :

```
ze′ ∊ zeichen \ {error.zeichen}
st  ∊ nat \ {error.nat}∧ ug(ar) ≤ st ≤ og(ar)
ar′ ∊ array \ {error.array}                        */
```

<u>Fall 1</u>:

```
ϱ(imcons(ze *paar(ar *null)))

  =                              /wegen der Fallunterscheidung

ϱ(imcons(ze *paar(*leerarray *null)))

  =                                      /nach Def. von imcons

ϱ(paar(schreibe(*leerarray *succ(*null) ze) *succ(*null)))

  =                                                /Evaluierung

ϱ(*paar(*schreibe(*leerarray *succ(*null) ze) *succ(*null)))

  =                                            /nach Def. von ϱ

rep.qprog(*paar(*schreibe(*leerarray *succ(*null) ze)
                *succ(*null)))

  =      /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cons(imhd(*paar(*schreibe(*leerarray *succ(*null) ze)
                *succ(*null)))
     rep.qprog(imtl(*paar(*schreibe(*leerarray *succ(*null)
                                    ze) *succ(*null)))))

  =                                        /nach Def. von imhd

cons(lese(*schreibe(*leerarray *succ(*null) ze)
          pred(plus(ug(*schreibe(*leerarray *succ(*null) ze))
                    *succ(*null))))
     rep.qprog(imtl(*paar(*schreibe(*leerarray *succ(*null)
                                    ze) *succ(*null)))))

  =
```



```
/*Betrachtungen:

  ug(*schreibe(*leerarray *succ(*null) ze))
    =                                                 /trivial
  *succ(*null)

  pred(plus(ug(*schreibe(*leerarray *succ(*null) ze))
            *succ(*null)))
    =                                              /Arithmetik
  pred(plus(*succ(*null) *succ(*null)))
    =
  *succ(*null)

  Diese Umformungen auf den letzten Ausdruck angewendet füh-
  ren zu :                                                  */

  =

cons(lese(*schreibe(*leerarray *succ(*null) ze) *succ(*null))
     rep.qprog(imtl(*paar(*schreibe(*leerarray *succ(*null)
                                    ze) *succ(*null)))))

  =           /nach einer Property bzgl. der Operation lese

cons(ze rep.qprog(imtl(*paar(*schreibe(*leerarray *succ(*null)
                                       ze) *succ(*null)))))

  =                                        /nach Def. von imtl

cons(ze rep.qprog(*paar(*schreibe(*leerarray *succ(*null) ze)
                  *null)))

  =                                   /nach Def. von rep.qprog

cons(ze nil)


/*Ende der Betrachtung von Fall 1  (ar =*leerarray)         */
```

Fall 2:

```
/*In diesem Fall ist ar ein von dem Konstruktor *schreibe auf-
  gebauter Term. Daher hat auch ug(ar) einen wohldefinierten
  Wert größer *null.
  Wie ar explizit "aussieht", ist für die Betrachtung unwe-
  sentlich; es wird die Bezeichnung ar weitergeführt.      */

ϱ(imcons(ze *paar(ar *null)))

  =                                      /nach Def. von imcons

ϱ(paar(schreibe(ar plus(ug(ar) *null) ze) *succ(*null)))

  =                                 /nach Evaluierung von paar

ϱ(*paar(schreibe(ar plus(ug(ar) *null) ze) *succ(*null)))

  =                                            /nach Def. von ϱ

rep.qprog(*paar(schreibe(ar plus(ug(ar) *null) ze)
                *succ(*null)))

  =     /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cons(imhd(*paar(schreibe(ar plus(ug(ar) *null) ze)
                *succ(*null)))
     rep.qprog(imtl(*paar(schreibe(ar plus(ug(ar) *null) ze)
                          *succ(*null)))))

  =                                        /nach Def. von imhd

cons(lese(schreibe(ar plus(ug(ar) *null) ze)
          pred(plus(ug(ar) *succ(*null))))
     rep.qprog(imtl(*paar(schreibe(ar plus(ug(ar) *null) ze)
                          *succ(*null)))))

  =                                                /Arithmetik
```



```
cons(lese(schreibe(ar ug(ar) ze) ug(ar))
     rep.qprog(imtl(*paar(schreibe(ar ug(ar) ze)
                          *succ(*null)))))

  =                          /Property bzgl. der Operation lese

cons(ze rep.qprog(imtl(*paar(schreibe(ar ug(ar) ze)
                             *succ(*null)))))

  =                                        /nach Def. von imtl

cons(ze rep.qprog(*paar(schreibe(ar ug(ar) ze) *null)))

  =                                   /nach Def. von rep.qprog

cons(ze nil)

/*Ende der Behandlung von Fall 2.                           */
```

Somit haben wir bis hierhin gezeigt (aus Fall 1 und Fall 2) :

■■1■■ ϱ(imcons(ze *paar(ar *null))) = cons(ze nil)

Als nächstes wird nun cons(ϱ(ze) ϱ(iq)) abgeleitet:

cons(ϱ(ze) ϱ(iq)) =

cons(ϱ(ze) ϱ(*paar(ar *null)))

= /nach Def. von ϱ, da ze ∉ imq

cons(ze ϱ(*paar(ar *null)))

= /nach Def. von ϱ

cons(ze rep.qprog(*paar(ar *null)))

= /nach Def. von rep.qprog

cons(ze nil)

Somit ist also gezeigt:

cons(ϱ(ze) ϱ(iq)) = cons(ze nil)

und

ϱ(imcons(ze iq)) = cons(ze nil) /siehe ■■1■■

Also gilt:

ϱ(imcons(ze iq)) = cons(ϱ(ze) ϱ(iq))

/*Somit ist der Induktionsanfang bewiesen. */

<u>Induktionsannahme</u> (*null ≤ na ≤ nb)

Folgendes sei bewiesen für *null ≤ na ≤ nb :

∀iq∈imq\{error.imq}. iq = *paar(ar na). *null ≤ na ≤ nb.
∀ze∈zeichen\{error.zeichen}.:

ϱ(imcons(ze iq)) = cons(ϱ(ze) ϱ(iq))

<u>Induktionsschritt</u> (na = *succ(nb))

Für na = *succ(nb) ist zu zeigen:

∀iq∈imq\{error.imq}. iq = *paar(ar na). na = *succ(nb).
∀ze∈zeichen\{error.zeichen}.:
ϱ(imcons(ze iq)) = cons(ϱ(ze) ϱ(iq))

/*Es sei nochmals darauf hingewiesen, daß
iq ≠ error.imq
aufgrund der am Anfang der Beweise gemachten generellen Vorraussetzung erfüllt ist.

Es wird nun wie bei dem Beweis des Induktiosanfanges erst die linke und dann die rechte Seite der Gleichung abgeleitet. /*

```
ϱ(imcons(ze iq)) =

ϱ(imcons(ze *paar(ar na)))

  =                  /da nach Induktionsschritt na = *succ(nb)

ϱ(imcons(ze *paar(ar *succ(nb))))

  =                                    /nach Def von imcons

ϱ(paar(schreibe(ar plus(ug(ar) *succ(nb)) ze)
        *succ(*succ(nb))))

  =

/*Der "schreibe"-Ausdruck evaluiert zu einem korrekten Term
  der Sorte zarray und nicht zu error.zarray.
  Somit evaluiert auch der "paar"-Ausdruck zu einem korrekten
  Term der Sorte imq ungleich error.imq.
  Daher kommt unter Anwendung von ϱ rep.qprog zur Ausführ-
  ung. Somit ergibt sich folgender Ausdruck:               */

  =

rep.qprog(*paar(schreibe(ar plus(ug(ar) *succ(nb)) ze)
                *succ(*succ(nb))))

  =     /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cons(imhd(*paar(schreibe(ar plus(ug(ar) *succ(nb)) ze)
                *succ(*succ(nb))))
     rep.qprog(imtl(*paar(schreibe(ar plus(ug(ar) *succ(nb))
                              ze) *succ(*succ(nb))))))

  =                                     /nach Def. von imhd

cons(lese(schreibe(ar plus(ug(ar) *succ(nb)) ze)
          pred(plus(ug(ar) *succ(*succ(nb)))))
     rep.qprog(imtl(*paar(schreibe(ar plus(ug(ar) *succ(nb))
                              ze) *succ(*succ(nb))))))

  =                                             /Arithmetik
```



```
cons(lese(schreibe(ar plus(ug(ar) *succ(nb)) ze)
          plus(ug(ar) *succ(nb)))
     rep.qprog(imtl(*paar(schreibe(ar plus(ug(ar) *succ(nb))
                              ze) *succ(*succ(nb))))))

  =                                    /Property bzgl. lese

cons(ze rep.qprog(imtl(*paar(schreibe(ar plus(ug(ar)
                                        *succ(nb)) ze)
                        *succ(*succ(nb))))))

  =                                     /nach Def. von imtl

cons(ze rep.qprog(*paar(schreibe(ar plus(ug(ar) *succ(nb)) ze)
                  *succ(nb))))

  =      /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cons(ze cons(imhd(*paar(schreibe(ar plus(ug(ar) *succ(nb)) ze)
                  *succ(nb)))
             rep.qprog(imtl(*paar(schreibe(ar plus(ug(ar)
                                                  *succ(nb))
                                        ze)
                         *succ(nb))))))

  =                                     /nach Def. von imhd

cons(ze cons(lese(schreibe(ar plus(ug(ar) *succ(nb)) ze)
                  pred(plus(ug(ar) succ(nb))))
             rep.qprog(imtl( *paar(schreibe(ar plus(ug(ar)
                                                   *succ(nb))
                                        ze)
                          *succ(nb))))))

  =
```

```
/*Laut der generellen Voraussetzung gilt:
  ϱ(iq) ≠ error.qprog;
  daraus folgt dann direkt:

  ∀i∈nat. 0 ≤ i ≤ nb.: is_def(ar plus(ug(ar) i)) = true.
  Damit gilt dann insbesondere:

  is_def(ar plus(ug(ar) nb)) = true.
  Somit gilt natürlich auch :

  is_def(schreibe(ar plus(ug(ar) *succ(nb)) ze)
         plus(ug(ar) nb))                        =  true

  Im folgenden sei definiert:

  ■■2■■: ze' = lese(schreibe(ar plus(ug(ar) *succ(nb)) ze)
                plus(ug(ar) nb))                            */

  =

cons(ze cons(ze'
             rep.qprog(imtl(*paar(schreibe(ar plus(ug(ar)
                                                *succ(nb))
                                          ze) *succ(nb))))))

  =                                           /nach Def. von imtl

cons(ze cons(ze'
             rep.qprog(*paar(schreibe(ar plus(ug(ar)
                                                *succ(nb)) ze)
                             nb))))


/*cons(ze' rep.qprog(*paar(schreibe(ar plus(ug(ar) *succ(nb))
                                     ze) nb)))

  ist ein Ausdruck, für den die Induktionsannahme zutrifft.
  Daher ist eine weitere Ableitung nicht nötig.

  Es wird nun  cons(ϱ(ze) ϱ(iq))  abgeleitet:                */
```



```
cons(ϱ(ze) ϱ(iq))  =

cons(ϱ(ze) ϱ(*paar(ar na)))  =

cons(ϱ(ze) ϱ(*paar(ar *succ(nb))))

  =                                 /nach Def. von ϱ, für ze ∉ imq

cons(ze ϱ(*paar(ar *succ(nb))))

  =                                                /nach Def. von ϱ

cons(ze rep.qprog(*paar(ar *succ(nb))))

  =      /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cons(ze cons(imhd(*paar(ar *succ(nb)))
             rep.qprog(imtl(*paar(ar *succ(nb))))))

  =                                             /nach Def. von imhd

cons(ze cons(lese(ar pred(plus(ug(ar) *succ(nb))))
             rep.qprog(imtl(*paar(ar *succ(nb))))))

  =                                                    /Arithmetik

cons(ze cons(lese(ar plus(ug(ar) nb))
             rep.qprog(imtl(*paar(ar *succ(nb))))))

  =

/*Es sei zb∈zeichen\{error.zeichen}.
  Eine Property bzgl. der Operation lese ist:

  lese(schreibe(ar plus(ug(ar) *succ(nb)) zb) nb)
    =
  lese(ar plus(ug(ar) nb))

  Diese Property auf den letzten Ausdruck unter Berücksich-
  tigung der Festlegung  ■■2■■  angewendet, ergibt:

  lese(ar plus(ug(ar nb)) = ze'.

  Somit läßt sich ableiten:                                  */

  =
```

```
cons(ze cons(ze'
          rep.qprog(imtl(*paar(ar *succ(nb))))))

  =                                        /nach Def. von imtl

cons(ze cons(ze' rep.qprog(*paar(ar nb))))
```

An dieser Stelle fassen wir zusammen, was bis hierhin erreicht ist:

```
ϱ(imcons(ze iq))

  =                                     /als Ableitungsergebnis

cons(ze cons(ze'
          rep.qprog(*paar(schreibe(ar plus(ug(ar)
                                      *succ(nb)) ze)
                      nb))))
```

und

```
cons(ϱ(ze) ϱ(iq))

  =                                     /als Ableitungsergebnis

cons(ze cons(ze' rep.qprog(*paar(ar nb))))
```

Nach einer Property aus Z_ARRAY gilt:

```
∀k∈[0 : nb-1].:
lese(ar plus(ug(ar) k))
  =
lese(schreibe(ar plus(ug(ar) *succ(nb)) ze)
     plus(ug(schreibe(ar plus(ug(ar) *succ(nb)) ze)) k))
```

Nach Definition von rep.qprog ist auch erfüllt, daß der "rep.-qprog"-Ausdruck der beiden Ableitungsergebnisse zu jeweils einem Wert we mit :
we ≠ error.x mit x ∈ {bool, nat, zeichen, qprog} evaluiert.

Diese Eigenschaften entsprechen den Voraussetzungen des Satzes, sodaß er zur Anwendung kommen kann und folgendes Ergebnis liefert:


```
rep.qprog(*paar(ar nb))

  =                                        /nach Satz und s.o.

rep.qprog(*paar(schreibe(ar plus(ug(ar) *succ(nb))ze) nb))


/*Damit lassen sich nun folgende Äquivalenzen aufstellen:    */

ϱ(imcons(ze iq))

  =                                     /als Ableitungsergebnis

cons(ze cons(ze'
          rep.qprog(*paar(schreibe(ar plus(ug(ar)
                                      *succ(nb)) ze)
                      nb))))

  =                                                  /nach Satz

cons(ze cons(ze'
          rep.qprog(*paar(ar nb))))

  =                                     /als Ableitungsergebnis

cons(ϱ(ze) ϱ(iq))
                                                            qed

/*Ende des Beweises bzgl. "imcons".                          */
```

8.6.3. imhd

zu zeigen:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).:
          ϱ(imhd(iq))  =  hd(ϱ(iq))
```

Beweis :

Fallunterscheidung

Fall 1: (na = *null)

```
hd(ϱ(iq)) =
hd(ϱ(*paar(ar na))) =
hd(ϱ(*paar(ar *null)))
  =                                   /nach Def. von ϱ
hd(rep.qprog(*paar(ar *null)))
  =                            /nach Def. von rep.qprog
hd(nil)
  =                                  /nach Def. von hd
error.qprog


ϱ(imhd(iq)) =
ϱ(imhd(*paar(ar na))) =
ϱ(imhd(*paar(ar *null)))
  =                                /nach Def. von imhd
ϱ(errror.imq) =
error.qprog
```


Fall 2: (na > null)

zu zeigen:

```
hd(ϱ(*paar(ar na)))  = ϱ(imhd(*paar(ar na)))

hd(ϱ(iq)) =
hd(ϱ(*paar(ar na)))
  =                                      /nach Def. von ϱ
hd(rep.qprog(*paar(ar na)))
  =      /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3
hd(cons(imhd(*paar(ar na))
        rep.qprog(imtl(*paar(ar na)))))
  =                                     /nach Def. von hd
imhd(*paar(ar na))
  =                  /nach Def. von ϱ für Elemente ∉ imq
ϱ(imhd(*paar(ar na)))  =
ϱ(imhd(iq))                                          qed

/*Ende des Beweises für Fall 2 und somit für imhd          */
```

8.6.4. imlast

zu zeigen :

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).:
          ϱ(imlast(iq))  =  last(ϱ(iq))
```

<u>Beweis</u> :

Induktionsbeweis über na

<u>Induktionsanfang</u> (na = *null)

zu zeigen für na = *null :

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).:
        ϱ(imlast(iq))  =  last(ϱ(iq)).
```

```
ϱ(imlast(iq)) =

ϱ(imlast(*paar(ar na))) =

ϱ(imlast(*paar(ar *null))

  =                                         /nach Def. von imlast

ϱ(error.imq)  =

error.qprog

/*Es wird nun last(ϱ(iq)) abgeleitet.                        */
```



```
last(ϱ(iq)) =

last(ϱ(*paar(ar na))) =

last(ϱ(*paar(ar *null))

  =                                              /nach Def. von ϱ

last(rep.qprog(*paar(ar na)))

  =                                          /nach Def. von rep.qprog

last(nil) =

error.qprog                                                   qed


/*Es wird hier im Induktionsanfang noch ein zweiter Fall,
  für na = *succ(*null), abgehandelt, da für den eben be-
  trachteten Fall imlast zu error.imq und last zu error.qprog
  als Ergebnis führt, und dieser Fall für die korrekte Veran-
  kerung des Induktionsbeweises zwar berücksichtigt werden muß
  , aber dafür alleine nicht ausreicht.                      */

na = *succ(*null)

ϱ(imlast(iq)) =

ϱ(imlast(*paar(ar na))) =

ϱ(imlast(*paar(ar *succ(*null))))

  =                                           /nach Def. von imlast

ϱ(lese(ar ug(ar)))

  =                    /nach Def. von ϱ, da lese(ar ug(ar)) ∉ imq

lese(ar ug(ar))        ■■1■■


/*Es wird nun  last(ϱ(iq))  abgeleitet :                     */
```

```
last(ϱ(iq)) =

last(ϱ(*paar(ar na))) =

last(ϱ(*paar(ar *succ(*null))))

  =                                         /nach Def. von ϱ

last(rep.qprog(*paar(ar *succ(*null))))

  =      /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

last(cons(imhd(*paar(ar *succ(*null)))
          rep.qprog(imtl(*paar(ar *succ(*null))))))

  =                                       /nach Def. von imhd

last(cons(lese(ar pred(plus(ug(ar) *succ(*null))))
          rep.qprog(imtl(*paar(ar *succ(*null))))))

  =                                              /Arithmetik

last(cons(lese(ar ug(ar))
          rep.qprog(imtl(*paar(ar *succ(*null))))))

  =                                       /nach Def. von imtl

last(cons(lese(ar ug(ar))
          rep.qprog(*paar(ar *null))))

  =                                  /nach Def. von rep.qprog

last(cons(lese(ar ug(ar)) nil)))

  =                                       /nach Def. von last

lese(ar ug(ar))

  =                      /als Ableitungsergebnis, siehe ■■1■■

ϱ(imlast(*paar(ar *succ(*null))))

  =

ϱ(imlast(iq))

/*Ende des Beweises des Induktionsanfanges.                  */
```


Induktionsannahme (*null < na ≤ nb)

Folgendes sei bewiesen für *null < na ≤ nb :

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).:
         ϱ(imlast(iq))  =  last(ϱ(iq))
```

Induktionsschritt (na = *succ(nb))

Für na = *succ(nb) ist zu zeigen:

```
∀iq∈imq \{error.imq}. iq = *paar(ar na).:
         ϱ(imlast(iq))  =  last(ϱ(iq))

ϱ(imlast(iq)) =

ϱ(imlast(*paar(ar na))) =

ϱ(imlast(*paar(ar *succ(nb))))

  =                                     /nach Def. von imlast

ϱ(lese(ar ug(ar))

  =              /nach Def von ϱ, da lese(ar ug(ar)) ∉ imq

lese(ar ug(ar))       ■■2■■

/*Als nächstes wird nun last(ϱ(iq)) abgeleitet.              */
```

```
last(ϱ(iq)) =

last(ϱ(*paar(ar na)))

  =

last(ϱ(*paar(ar *succ(nb))))

  =                                               /nach Def. von ϱ

last(rep.qprog(*paar(ar succ(nb))))

  =      /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

last(cons(imhd(*paar(ar succ(nb)))
          rep.qprog(imtl(*paar(ar succ(nb))))))

  =

/*Nach Voraussetzung gilt:  nb > *null.
  daraus folgt direkt:
  imtl(*paar(ar succ(nb))) = *paar(ar nb) ≠ *paar(ar *null)

  Demnach ergibt die Anwendung von last im obigen
  Ausdruck nach Definition folgende Ableitung:               */

  =

last(rep.qprog(imtl(*paar(ar succ(nb)))))

  =                                            /nach Def. von imtl

last(rep.qprog(*paar(ar nb)))

  =                               /Def. von ϱ umgekehrt angewendet

last(ϱ(*paar(ar nb)))

  =                                         /nach Induktionsannahme

ϱ(imlast(*paar(ar nb)))

  =                                           /nach Def. von imlast
```



```
ϱ(lese(ar ug(ar)))

  =                /nach Def. von ϱ, da lese(ar ug(ar)) ∉ imq

lese(ar ug(ar))

  =                                  /Ableitungsergebnis ■■2■■

ϱ(imlast(iq))                                               qed

/*Ende des Beweises zu imlast                                */
```

8.6.5. imtl

zu zeigen:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).:
            ϱ(imtl(iq))  =  tl(ϱ(iq))
```

<u>Beweis</u> :

<u>Fallunterscheidung</u>

<u>Fall 1</u>: (na = *null)

```
ϱ(imtl(iq)) =

ϱ(imtl(*paar(ar na))) =

ϱ(imtl(*paar(ar *null)))

  =                                   /nach Def. von imtl

ϱ(*paar(ar *null))

  =                                   /nach Def. von ϱ

rep.qprog(*paar(ar *null))

  =                                   /nach Def. von rep.qprog

nil
```

/*Es wird nun tl(ϱ(iq)) abgeleitet. */


```
tl(ϱ(iq)) =

tl(ϱ(*paar(ar na))) =

tl(ϱ(*paar(ar *null)))

  =                                   /nach Def. von ϱ

tl(rep.qprog(*paar(ar *null)))

  =                                   /nach Def. von rep.qprog

tl(nil)

  =                                   /nach Def. von tl

nil
```

/*Ende des Beweises zu Fall 1 */

<u>Fall 2</u>: (na > *null)

Für na > *succ(nb) ist zu zeigen:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na). :
          ϱ(imtl(iq))  =  tl(ϱ(iq))


tl(ϱ(iq)) =

tl(ϱ(*paar(ar na)))

  =                                             /nach Def. von ϱ

tl(rep.qprog(*paar(ar na)))

  =      /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

tl(cons(imhd(*paar(ar na))
       rep.qprog(imtl(*paar(ar na)))))

  =                                            /nach Def. von tl

rep.qprog(imtl(*paar(ar na)))

  =             /nach Def. von ϱ, da imtl(*paar(ar na)) ∈ imq

ϱ(imtl(*paar(ar na)))

  =

ϱ(imtl(iq))
                                                             qed

/*Ende des Beweises bzgl. imtl                                */
```


8.6.6. imlaenge

zu zeigen:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).:
            ϱ(imlaenge(iq) = laenge(ϱ(iq))
```

<u>Beweis</u> :

Induktionsbeweis über na

<u>Induktionsanfang</u> (na = *null)

```
ϱ(imlaenge(iq)) =

ϱ(imlaenge(*paar(ar na))) =

ϱ(imlaenge(*paar(ar *null)))

  =                                     /nach Def. von imlaenge

ϱ(*null) =

*null


laenge(ϱ(iq)) =

laenge(ϱ(*paar(ar na))) =

laenge(ϱ(*paar(ar *null)))

  =                                              /nach Def. von ϱ

laenge(rep.qprog(*paar(ar *null)))

  =                                     /nach Def. von rep.qprog

laenge(nil) = *null
```

<u>Induktionsannahme</u> (*null ≤ na ≤ nb)

Folgendes sei bewiesen für *null ≤ na ≤ nb:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).:
          ϱ(imlaenge(iq))  =  laenge(ϱ(iq))
```

<u>Induktionsschritt</u> (na = *succ(nb))

Für na = *succ(nb) ist zu zeigen:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).:
          ϱ(imlaenge(iq))  =  laenge(ϱ(iq))
```

```
ϱ(imlaenge(iq)) =

ϱ(imlaenge(*paar(ar na))) =

ϱ(imlaenge(*paar(ar *succ(nb))))

  =                                    /nach Def. von imlaenge

ϱ(*succ(nb))

  =                                           /nach Def. von ϱ

*succ(nb)
```

/*Es wird nun laenge(ϱ(iq)) abgeleitet: */


```
laenge(ϱ(iq)) =

laenge(ϱ(*paar(ar na))) =

laenge(ϱ(*paar(ar *succ(nb))))

  =                                           /nach Def. von ϱ

laenge(rep.qprog(*paar(ar *succ(nb))))

  =     /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

laenge(cons(imhd(*paar(ar *succ(nb))))
            rep.qprog(imtl(*paar(ar *succ(nb)))))

  =                                      /nach Def. von laenge

succ(laenge(rep.qprog(imtl(*paar(ar *succ(nb))))))

  =                                        /nach Def. von imtl

succ(laenge(rep.qprog(*paar(ar nb))))

  =                   /nach Def. von ϱ, da *paar(ar nb) ∈ imq

succ(laenge(ϱ(*paar(ar nb))))

  =                                    /nach Induktionsannahme

succ(ϱ(imlaenge(*paar(ar nb))))

  =                                    /nach Def. von imlaenge

succ(ϱ(nb))

  =                                           /nach Def. von ϱ

succ(nb)

  =                                               /Evaluierung

*succ(nb)

                                                           qed
```

/*Ende des Beweises bzgl. imlaenge. */

8.6.7. imcat

zu zeigen:

```
∀iq,iq'∈imq\{error.imq}. iq = *paar(ar na).
iq' = *paar(ar' na').:
      ϱ(imcat(iq iq'))  =  cat(ϱ(iq) ϱ(iq'))
```

<u>Beweis</u> :

Induktionsbeweis über na

<u>Induktionsanfang</u> (na = *null)

zu zeigen für na = *null:

```
∀iq,iq'∈imq\{error.imq}. iq = *paar(ar na).
iq' = *paar(ar'na').:
      ϱ(imcat(iq iq'))  =  cat(ϱ(iq) ϱ(iq'))
```

```
ϱ(imcat(iq iq')) =

ϱ(imcat(*paar(ar na) *paar(ar' na'))) =

ϱ(imcat(*paar(ar *null) *paar(ar' na')))

  =                                          /nach Def. von imcat

ϱ(*paar(ar' na'))
```

/*Die Ableitung wird an dieser Stelle abgebrochen, und es
  wird nun cat(ϱ(iq) ϱ(iq')) abgeleitet:


```
cat(ϱ(iq) ϱ(iq')) =

cat(ϱ(*paar(ar na)) ϱ(*paar(ar' na'))) =

cat(ϱ(*paar(ar *null)) ϱ(*paar(ar' na')))

  =                                          /nach Def. von ϱ

cat(rep.qprog(*paar(ar *null)) ϱ(*paar(ar' na')))

  =                                    /nach Def. von rep.qprog

cat(nil ϱ(*paar(ar' na')))

  =                                          /nach Def. von cat

ϱ(*paar(ar' na'))
```

/*Ende des Induktionsanfangs */

<u>Induktionsannahme</u> (*null ≤ na ≤ nb)

Folgendes sei bewiesen für *null ≤ na ≤ nb:

```
∀iq,iq'∈imq\{error.imq}. iq = *paar(ar na).
iq' = *paar(ar' na').:
      ϱ(imcat(iq iq'))  =  cat(ϱ(iq) ϱ(iq'))
```

<u>Induktionsschritt</u> (na = *succ(nb))

Für na = *succ(nb) ist zu zeigen:

```
∀iq,iq'∈imq\{error.imq}. iq = *paar(ar na).
iq' = *paar(ar' na').:
       ϱ(imcat(iq iq'))  =  cat(ϱ(iq) ϱ(iq'))



ϱ(imcat(iq iq')) =

ϱ(imcat(*paar(ar na) *paar(ar' na'))) =

ϱ(imcat(*paar(ar *succ(nb)) *paar(ar' na')))

  =                                   /nach Def. von imcat

ϱ(imcons(imhd(*paar(ar *succ(nb)))
         imcat(imtl(*paar(ar *succ(nb))) *paar(ar' na'))))

  =                                   /nach Def. von imtl

ϱ(imcons(imhd(*paar(ar *succ(nb)))
         imcat(*paar(ar nb) *paar(ar'na'))))

  =                                   /nach Def. von ϱ

rep.qprog(imcons(imhd(*paar(ar *succ(nb)))
                 imcat(*paar(ar nb) *paar(ar' na'))))

  =     /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cons(imhd(imcons(imhd(*paar(ar *succ(nb))))
                 imcat(*paar(ar nb) *paar(ar' na'))))
     rep.qprog(imtl(imcons(imhd(*paar(ar *succ(nb)))
                     imcat(*paar(ar nb)
                           *paar(ar' na'))))))

  =                                   /nach Def. von imhd
```



```
cons(imhd(*paar(ar *succ(nb)))
     rep.qprog(imtl(imcons(imhd(*paar(ar *succ(nb)))
                           imcat(*paar(ar nb)
                                 *paar(ar' na'))))))

  =                                   /Property bzgl. imtl

cons(imhd(*paar(ar *succ(nb)))
     rep.qprog(imcat(*paar(ar nb) *paar(ar' na'))))

  =                                   /nach Def. von ϱ

cons(imhd(*paar(ar *succ(nb)))
     ϱ(imcat(*paar(ar nb) *paar(ar' na'))))

/*An dieser Stelle wird die Ableitung nicht weitergeführt;
  es ist direkt erkennbar, daß auf das zweite Argument des
  cons- Ausdruckes die Induktionsannahme zutrifft.

  Im weiteren wird nun  cat(ϱ(iq) ϱ(iq'))  abgeleitet:     */



cat(ϱ(iq) ϱ(iq')) =

cat(ϱ(*paar(ar na)) ϱ(*paar(ar' na'))) =

cat(ϱ(*paar(ar *succ(nb))) ϱ(*paar(ar' na')))

  =                                   /nach Def. von ϱ

cat(rep.qprog(*paar(ar *succ(nb))) ϱ(ar' na' )))

  =     /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cat(cons(imhd(*paar(ar *succ(nb))))
         rep.qprog(imtl(*paar(ar *succ(nb)))))
    ϱ(*paar(ar' na')))

  =                                   /nach Def. von cat
```

```
cons(imhd(*paar(ar *succ(nb)))
     cat(rep.qprog(imtl(*paar(ar *succ(nb))))
         ϱ(*paar(ar' na'))))

  =                                               /nach Def. von imtl

cons(imhd(*paar(ar *succ(nb)))
     cat(rep.qprog(*paar(ar nb)) ϱ(*paar(ar' na'))))

  =                         /nach Def. von ϱ, da *paar(ar na) ∈ imq

cons(imhd(*paar(ar *succ(nb)))
     cat(ϱ(*paar(ar nb)) ϱ(*paar(ar' na'))))

  =                                         /nach Induktionsannahme

cons(imhd(*paar(ar *succ(nb)))
     ϱ(imcat(*paar(ar nb) *paar(ar' na'))))

  =                                         /als Ableitungsergebnis

ϱ(imcat(*paar(ar *succ(nb)) *paar(ar' na')))

  =

ϱ(imcat(iq iq'))
                                                                qed
```

/*Ende des Beweises bzgl. imcat. */

8.6.8. immbre

zu zeigen:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).
∀ze∈zeichen\{error.zeichen}.:
    ϱ(immbre(ze iq))  =  mbre(ϱ(ze) ϱ(iq))
```

Beweis :

Induktionsbeweis über na

Induktionsanfang (na = *null)

Für na = *null ist zu zeigen:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).
∀ze∈zeichen\{error.zeichen}.:
    ϱ(immbre(ze iq))  =  mbre(ϱ(ze) ϱ(iq))

ϱ(immbre(ze iq)) =

ϱ(immbre(ze *paar(ar na))) =

ϱ(immbre(ze *paar(ar *null)))

  =                                           /nach Def. von immbre

ϱ(false)

  =                                                /nach Def. von ϱ

false
```

/*Es wird nun mbre(ϱ(ze) ϱ(iq)) abgeleitet: */

```
mbre(ϱ(ze) ϱ(iq)) =

mbre(ϱ(ze) ϱ(*paar(ar na))) =

mbre(ϱ(ze) ϱ(*paar(ar *null)))

  =                             /nach Def. von ϱ, da ze ∉ imq

mbre(ze ϱ(*paar(ar *null)))

  =                                          /nach Def. von ϱ

mbre(ze rep.qprog(*paar(ar *null)))

  =                                    /nach Def. von rep.qprog

mbre(ze nil)

  =                                        /nach Def. von mbre

false

/*Ende des Induktionsanfanges                                */
```

<u>Induktionsannahme</u>   (*null ≤ na ≤ nb)

Folgendes sei bewiesen für *null ≤ na ≤ nb:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).
∀ze∈zeichen\{error.zeichen}.:
        ϱ(immbre(ze iq))  =  mbre(ϱ(ze) ϱ(iq))
```


<u>Induktionsschritt</u>   (na = *succ(nb))

Für ar = *succ(nb) ist zu zeigen:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).
∀ze∈zeichen\{error.zeichen}.:
        ϱ(immbre(ze iq))  =  mbre(ϱ(ze) ϱ(iq))


ϱ(immbre(ze iq)) =

ϱ(immbre(ze *paar(ar na))) =

ϱ(immbre(ze *paar(ar *succ(nb))))


/*An dieser Stelle müssen nun analog zur Definition von
  immbre die beiden Alternativen berücksichtigt werden,
  und zwar:

  1.) imhd(*paar(ar *succ(nb))) = ze

  2.) imhd(*paar(ar *succ(nb))) ≠ ze                          */
```

<u>Fallunterscheidung</u> :

<u>Fall 1</u>:

```
imhd(*paar(ar *succ(nb))) = ze     ⇒


ϱ(immbre(ze *paar(ar *succ(nb))))

  =                    /nach Def. von immbre und wegen Fall 1

ϱ(true) = true
```

Fall 2:

```
imhd(*paar(ar *succ(nb))) ≠ ze            ⇒

ϱ(immbre(ze *paar(ar *succ(nb))))
  =                                   /nach Def. von immbre
ϱ(immbre(ze imtl(*paar(ar *succ(nb)))))
  =                                     /nach Def. von imtl
ϱ(immbre(ze *paar(ar nb)))
  =                                 /nach Induktionsannahme
mbre(ϱ(ze) ϱ(*paar(ar nb)))
  =                          /nach Def. von ϱ, da ze ∉ imq
mbre(ze ϱ(*paar(ar nb)))
  =                                        /nach Def. von
mbre(ze rep.qprog(*paar(ar nb)))
```

/*Es wird nun mbre(ϱ(ze) ϱ(iq)) abgeleitet: */


```
mbre(ϱ(ze) ϱ(iq)) =
mbre(ϱ(ze) ϱ(*paar(ar na))) =
mbre(ϱ(ze) ϱ(*paar(ar *succ(nb))))
  =                          /nach Def. von ϱ, da ze ∉ imq
mbre(ze ϱ(*paar(ar *succ(nb))))
  =                                        /nach Def. von ϱ
mbre(ze rep.qprog(*paar(ar *succ(nb))))
  =     /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3
mbre(ze cons(imhd(*paar(ar *succ(nb)))
             rep.qprog(imtl(*paar(ar *succ(nb))))))
```

/*Es werden nun auch an dieser Stelle die gleichen Fälle
  unterschieden wie oben, nämlich:

  1. imhd(*paar(ar *succ(nb))) = ze

  2. imhd(*paar(ar *succ(nb))) = ze */

Fall 1:

```
imhd(*paar(ar *succ(nb))) = ze            ⇒

mbre(ze cons(imhd(*paar(ar *succ(nb)))
             rep.qprog(imtl(*paar(ar *succ(nb))))))
  =                                      /nach Def. von mbre
true
```

/*Somit ist für Fall 1 die zu zeigende Gleichheit bewiesen */

Fall 2:

```
imhd(*paar(ar *succ(nb))) ≠ ze        ⇒

mbre(ze cons(imhd(*paar(ar *succ(nb)))
         rep.qprog(imtl(*paar(ar *succ(nb))))))

  =                                   /nach Def. von mbre

mbre(ze rep.qprog(imtl(*paar(ar *succ(nb)))))

  =                                   /nach Def. von imtl

mbre(ze rep.qprog(*paar(ar nb)))

/*Somit ist nun auch die Äquivalenz für den zweiten Fall
  gezeigt und damit der Beweis bzgl. immbre abgeschlossen.  */
```


8.6.9. imcmpr

zu zeigen:

```
∀iq,iq'∈imq\{error.imq}. iq = *paar(ar na).
iq' = *paar(ar' na'). ∀nb∈nat\{error.nat}.:

   ϱ(imcmpr(iq iq' nb))  =  cmpr(ϱ(iq) ϱ(iq') ϱ(nb))
```

Beweis :

Induktionsbeweis über nb

Induktionsanfang (nb = *null)

zu zeigen:

```
∀iq,iq'∈ imq\{error.imq}. iq = *paar(ar na).
iq' = *paar(ar' na'). nb = *null.:
   ϱ(imcmpr(iq iq' nb))  =  cmpr(ϱ(iq) ϱ(iq') ϱ(nb))


ϱ(imcmpr(iq iq' nb)) =

ϱ(imcmpr(iq iq' *null))

  =                                   /nach Def. von imcmpr

ϱ(true)

  =                         /nach Def. von ϱ, da true ∉ imq

*true


/*Als nächstes wird  cmpr(ϱ(iq) ϱ(iq') ϱ(nb)) abgeleitet    */
```

```
cmpr(ϱ(iq) ϱ(iq') ϱ(nb)) =

cmpr(ϱ(iq) ϱ(iq') ϱ(*null))

  =                             /nach Def. von ϱ, da *null ∉ imq

cmpr(ϱ(iq) ϱ(iq') *null)

  =                                        /nach Def. von cmpr

true

/*Ende des Beweises zum Induktionsanfang                    */
```

<u>Induktionsannahme</u> (*null ≤ nb ≤ nn)

Folgendes sei bewiesen für *null ≤ nb ≤ nn:

```
∀iq,iq'∈imq\{error.imq}. iq = *paar(ar na).
iq' = *paar(ar' na'). ∀nb∈[0 : nn].:

   ϱ(imcmpr(iq iq' nb))  =  cmpr(ϱ(iq) ϱ(iq') ϱ(nb))
```


<u>Induktionsschritt</u> (nb = *succ(nn))

Für nb = *succ(nn) ist zu zeigen:

```
∀iq,iq'∈imq\{error,imq}. iq = *paar(ar na).
iq' = *paar(ar' na'). nb = *succ(nn).:

   ϱ(imcmpr(iq iq' nb))  =  cmpr(ϱ(iq) ϱ(iq') ϱ(nb))


ϱ(imcmpr(iq iq' nb)) =

ϱ(imcmpr(*paar(ar na) *paar(ar' na') nb))

/*An dieser Stelle müssen nun analog zur Definition von
  imcmpr die beiden Alternativen berücksichigt werden,
  und zwar:

  1.) na = *null  v  na' = *null

  2.) na ≠ *null  ∧  na' ≠ *null                           */
```

<u>Fallunterscheidung</u>

<u>Fall 1</u>:

```
o.B.d.A. sei na = *null                    ⇒

ϱ(imcmpr(*paar(ar na) *paar(ar' na') nb)) =

ϱ(imcmpr(*paar(ar *null) *paar(ar' na') nb))

  =                                     /nach Def. von imcmpr

ϱ(false) =

false
```

Fall 2:

```
na ≠ *null  ∧  na' ≠ *null                    ⇒

ϱ(imcmpr(*paar(ar na) *paar(ar' na') nb))

  =                    /nach Induktionsschritt: nb = *succ(nn)

ϱ(imcmpr(*paar(ar na)*paar(ar'na') *succ(nn)))

  =                                   /nach Def. von imcmpr

ϱ(and(eq(imhd(*paar(ar na)) imhd(*paar(ar' na')))
      imcmpr(imtl(*paar(ar na)) imtl(*paar(ar' na')) nn)))

  =

/*Da das Ergebnis der Ausführung der and- Operation ein Term
  der Sorte bool ist, kommt bei Anwendung von ϱ die identi-
  sche Abbildung zur Ausführung. Somit ergibt sich folgende
  Ableitung:                                             */

  =

and(eq(imhd(*paar(ar na)) imhd(*paar(ar' na')))
    imcmpr(imtl(*paar(ar na)) imtl(*paar(ar' na')) nn)))

/*An dieser Stelle wird diese Ableitung nicht mehr weiterge-
  führt,sondern es wird nun die rechte Seite der zu beweisen-
  den Äquivalenz abgeleitet.                              */
```



```
cmpr(ϱ(iq) ϱ(iq') ϱ(nb)) =

cmpr(ϱ(*paar(ar na)) ϱ(*paar(ar' na')) ϱ(nb)) =

cmpr(ϱ(*paar(ar na)) ϱ(*paar(ar' na')) ϱ(*succ(nn)))

/*Es werden nun auch äquivalent zu eben die beiden Alterna-
  tiven berücksichtigt, nämlich:

  1.) na = *null  v  na' = *null

  2.) na ≠ *null  v  na' ≠ *null                          */
```

Fall 1:

```
o.B.d.A. sei na = *null          ⇒


cmpr(ϱ(*paar(ar na)) ϱ(*paar(ar' na')) ϱ(*succ(nn)))

    =                /nach Def. von ϱ, da *succ(nn) ∉ imq

cmpr(ϱ(*paar(ar na)) ϱ(*paar(ar' na')) *succ(nn))

   =                                        /nach Def. von ϱ

cmpr(rep.qprog(*paar(ar na)) rep.qprog(*paar(ar' na'))
     *succ(nn))

   =                                         /da na = *null

cmpr(rep.qprog(*paar(ar *null)) rep.qprog(*paar(ar' na'))
     *succ(nn))

   =                                /nach Def. von rep.qprog

cmpr(nil rep.qprog(*paar(ar' na')) *succ(nn))

   =                                      /nach Def. von cmpr

false
```

Fall 2:

```
na ≠ *null  ∧  na' ≠ *null            ⇒

cmpr(ϱ(*paar(ar na)) ϱ(*paar(ar' na')) ϱ(*succ(nn)))

  =                        /nach Def. von ϱ, da *succ(nn) ∉ imq

cmpr(ϱ(*paar(ar na)) ϱ(*paar(ar' na')) *succ(nn))

  =                                         /nach Def. von cmpr

and(eq(hd(ϱ(*paar(ar na))) hd(ϱ(*paar(ar' na'))))
    cmpr(tl(ϱ(*paar(ar na))) tl(ϱ(*paar(ar' na'))) nn))

  =                                         /siehe 8.6.3.  imhd

and(eq(ϱ(imhd(*paar(ar na))) ϱ(imhd(*paar(ar' na'))))
    cmpr(tl(ϱ(*paar(ar na))) tl(ϱ(*paar(ar' na'))) nn))

  =                                         /siehe 8.6.5.  imtl

and(eq(ϱ(imhd(*paar(ar na))) ϱ(imhd(*paar(ar' na'))))
    cmpr(ϱ(imtl(*paar(ar na))) ϱ(imtl(*paar(ar' na'))) nn))

  =

/*Das Ergebnis einer Ausführung der Operation imhd ist von
  der Sorte zeichen, sodaß bei Anwendung von ϱ auf einen imhd-
  Ausdruck die identische Abbildung zur Ausführung kommt. So-
  mit ergibt sich folgende Ableitung:                       */

  =

and(eq(imhd(*paar(ar na)) imhd(*paar(ar' na')))
    cmpr(ϱ(imtl(*paar(ar na))) ϱ(imtl(*paar(ar' na'))) nn))

  =

/*Bzgl. der zweiten Komponente des and- Ausdrucks trifft die
  Induktionsannahme zu, sodaß sich dieser Teilausdruck ablei-
  ten läßt zu:
  ϱ(imcmpr(imtl(*paar(ar na)) imtl(*paar(ar' na')) nn))
  Daraus ergibt sich folgender Ausdruck:                    */

  =
```



```
and(eq(imhd(*paar(ar na)) imhd(*paar(ar' na')))
    ϱ(imcmpr(imtl(*paar(ar na)) imtl(*paar(ar' na')) nn)))

  =

/*Das Ergebnis der Ausführung der Operation imcmpr ist von der
  Sorte bool, sodaß bei Anwendung von ϱ auf solch einen Aus-
  druck die identische Abbildung zur Ausführung kommt.
  Somit ergibt sich folgende Ableitung:                       */

  =

and(eq(imhd(*paar(ar na)) imhd(*paar(ar' na')))
    imcmpr(imtl(*paar(ar na)) imtl(*paar(ar' na')) nn)))
                                                            qed
```

Wenn man nun die je zwei Ableitungsergebnisse der rechten und der linken Seite der zu beweisenden Äquivalenz jeweils miteinander vergleicht, so sieht man direkt, daß die Ableitungsergebnisse sowohl in Fall 1 alsauch in Fall 2 identisch sind. Somit ist also die zu beweisende Äequivalenz gezeigt und damit der Induktionsbeweis bzgl. imcmpr abgeschlossen.

/*Ende des Beweises bzgl. imcmpr. */

8.6.10. imcut

zu zeigen:

∀iq∈imq\{error.imq}. iq = *paar(ar na).:
    ϱ(imcut(iq)) = cut(ϱ(iq))

<u>Beweis</u>:

/*Es werden folgende zwei Fälle unterschieden:

1.) na = *null

2.) na ≠ *null */

<u>Fall 1</u>:

ϱ(imcut(iq)) =

ϱ(imcut(*paar(ar na))) =

ϱ(imcut(*paar(ar *null)))

= /nach Def. von imcut

ϱ(*paar(ar *null))

= /nach Def. von ϱ

rep.qprog(*paar(ar *null))

= /nach Def. von rep.qprog

nil

/*Es wird nun cut(ϱ(iq)) abgeleitet: */

cut(ϱ(iq)) =

cut(ϱ(*paar(ar na))) =

cut(ϱ(*paar(ar *null)))

= /nach Def. von ϱ

cut(rep.qprog(*paar(ar *null)))

= /nach Def. von rep.qprog

cut(nil)

= /nach Def. von cut

nil

/*Ende des Beweises für Fall 1 */

<u>Fall 2</u>:

Induktionsbeweis über na

<u>Induktionsanfang</u> (na = *succ(*null))

zu zeigen:

∀iq∈imq\{error.imq}. iq = *paar(ar na).
na = *succ(*null).:
    ϱ(imcut(iq)) = cut(ϱ(iq))

ϱ(imcut(iq)) =

ϱ(imcut(*paar(ar na))) =

ϱ(imcut(*paar(ar *succ(*null))))

= /nach Def. von imcut

```
ϱ(*paar(ar *null))

  =                                   /nach Def. von ϱ

rep.qprog(*paar(ar *null))

  =                            /nach Def. von rep.qprog

nil

/*Es wird nun cut(ϱ(iq)) abgeleitet:                */


cut(ϱ(iq)) =

cut(ϱ(*paar(ar na))) =

cut(ϱ(*paar(ar *succ(*null))))

  =                                   /nach Def. von ϱ

cut(rep.qprog(*paar(ar *succ(*null))))

  =     /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cut(cons(imhd(*paar(ar *succ(*null)))
        rep.qprog(imtl(*paar(ar *succ(*null))))))

  =                                /nach Def. von imtl

cut(cons(imhd(*paar(ar *succ(*null)))
        rep.qprog(*paar(ar *null))))

  =                            /nach Def. von rep.qprog

cut(cons(imhd(*paar(ar *succ(*null))) nil))

  =                                 /nach Def. von cut

nil


/*Ende des Beweises zum Induktionsanfang            */
```


<u>Induktionsannahme</u>   (*null < na ≤ nb)

Folgendes sei bewiesen für *null < na ≤ nb

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).
na∈[1 : nb].:
            ϱ(imcut(iq))  =  cut(ϱ(iq))
```

<u>Induktionsschritt</u>   (na = *succ(nb))

Für na = *succ(nb) ist zu zeigen:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na). na = *succ(nb).:

         ϱ(imcut(iq))  =  cut(ϱ(iq))


ϱ(imcut(iq)) =

ϱ(imcut(*paar(ar na))) =

ϱ(imcut(*paar(ar *succ(nb))))

  =                                  /nach Def. von imcut

ϱ(paar(subarray(ar succ(ug(ar)) og(ar)) nb))

  =                                         /Evaluierung

ϱ(*paar(subarray(ar *succ(ug(ar)) og(ar)) nb))

  =                                      /nach Def. von ϱ

rep.qprog(*paar(subarray(ar *succ(ug(ar)) og(ar)) nb))

  =      /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3
```

```
cons(imhd(*paar(subarray(ar *succ(ug(ar)) og(ar)) nb))
     rep.qprog(imtl(*paar(subarray(ar *succ(ug(ar)) og(ar))
                                         nb))))

  =                                         /nach Def. von imhd

cons(lese(subarray(ar *succ(ug(ar)) og(ar))
          pred(plus(ug(subarray(ar *succ(ug(ar)) og(ar)))
                                        nb)))
     rep.qprog(imtl(*paar(subarray(ar *succ(ug(ar)) og(ar))
                                         nb))))

  =  /da ug(subarray(ar *succ(ug(ar)) og(ar))) = *succ(ug(ar))

cons(lese(subarray(ar *succ(ug(ar)) og(ar))
          pred(plus(*succ(ug(ar)) nb)))
     rep.qprog(imtl(*paar(subarray(ar *succ(ug(ar)) og(ar))
                                         nb))))

  =                                                 /Arithmetik

cons(lese(subarray(ar *succ(ug(ar)) og(ar)) plus(ug(ar) nb))
     rep.qprog(imtl(*paar(subarray(ar *succ(ug(ar)) og(ar))
                                         nb))))

  =                                         /nach Def. von imtl

cons(lese(subarray(ar *succ(ug(ar)) og(ar)) plus(ug(ar) nb))
     rep.qprog(*paar(subarray(ar *succ(ug(ar)) og(ar))
                                    pred(nb))))


/*An dieser Stelle wird diese Ableitung nicht mehr weiterge-
  führt, sondern es wird nun die rechte Seite der zu bewei-
  senden Äquivalenz abgeleitet.                             */
```



```
cut(ϱ(iq)) =

cut(ϱ(*paar(ar na))) =

cut(ϱ(*paar(ar *succ(nb))))

  =                                            /nach Def. von ϱ

cut(rep.qprog(*paar(ar *succ(nb))))

  =      /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cut(cons(imhd(*paar(ar *succ(nb)))
         rep.qprog(imtl(*paar(ar *succ(nb))))))

  =                                         /nach Def. von imhd

cut(cons(lese(ar pred(plus(ug(ar) *succ(nb))))
         rep.qprog(imtl(*paar(ar *succ(nb))))))

  =                                                 /Arithmetik

cut(cons(lese(ar plus(ug(ar) nb))
         rep.qprog(imtl(*paar(ar *succ(nb))))))

  =                                          /nach Def. von cut

cons(lese(ar plus(ug(ar) nb))
     cut(rep.qprog(imtl(*paar(ar *succ(nb))))))

  =                                         /nach Def. von imtl

cons(lese(ar plus(ug(ar) nb))
     cut(rep.qprog(*paar(ar nb))))

  =           /wegen rep.qprog(*paar(ar nb)) = ϱ(*paar(ar nb))

cons(lese(ar plus(ug(ar) nb)) cut(ϱ(*paar(ar nb))))

  =                                     /nach Induktionsannahme

cons(lese(ar plus(ug(ar) nb)) ϱ(imcut(*paar(ar nb))))

  =                                        /nach Def. von imcut
```

```
cons(lese(ar plus(ug(ar) nb))
     ϱ(paar(subarray(ar ug(ar) og(ar)) pred(nb))))

  =                                        /Evaluierung

cons(lese(ar plus(ug(ar) nb))
     ϱ(*paar(subarray(ar ug(ar) og(ar)) pred(nb))))

  =                                        /nach Def. von ϱ

cons(lese(ar plus(ug(ar) nb))
     rep.qprog(*paar(subarray(ar ug(ar) og(ar)) pred(nb))))
```

/*Wenn man nun an dieser Stelle die beiden Ableitungsergebnisse von
ϱ(imcut(*paar(ar *succ(bn))))
und
cut(ϱ(*paar(ar *succ(nb))))
vergleicht, stellt man direkt die textuelle Gleichheit der jeweils zweiten Argumente der cons- Ausdrücke fest.
Es bleibt somit noch zu zeigen, daß gilt:

```
lese(subarray(ar *succ(ug(ar)) og(ar)) plus(ug(ar) nb))
  =
lese(ar plus(ug(ar) nb))                                 */
```

Aus der Definition der Operation subarray ist direkt ersichtlich, daß zutrifft:

∀i∈[1 : nb].:

```
lese(subarray(ar *succ(ug(ar)) og(ar)) plus(ug(ar) i))

  =                                  /nach Def. von subarray

lese(ar plus(ug(ar) i))
```

Damit gilt natürlich dann auch insbesondere:


```
lese(subarray(ar *succ(ug(ar)) og(ar)) plus(ug(ar) nb))

  =

lese(ar plus(ug(ar) nb))                                 qed
```

Damit ist also auch gezeigt, daß die jeweils ersten Argumente der cons- Ausdrücke der Ableitungsergebnisse von
ϱ(imcut(*paar(ar *succ(nb))))
und
cut(ϱ(*paar(ar *succ(bn)))) gleich sind.

Somit ist der Induktionsschritt bewiesen und damit der Beweis bzgl. imcut beendet.

/*Ende des Beweises bzgl. imcut */

8.6.11. imtail

zu zeigen:

∀iq∈imq\{error.imq}. iq = *paar(ar na).
∀nb∈nat\{error.nat}.:

ϱ(imtail(iq nb)) = tail(ϱ(iq) ϱ(nb))

<u>Beweis</u>

Induktionsbeweis über nb

<u>Induktionsanfang</u> (nb = *null)

zu zeigen:

∀iq∈imq\{error.imq}. iq = *paar(ar na). nb = *null.:
ϱ(imtail(iq nb)) = tail(ϱ(iq) ϱ(nb))

ϱ(imtail(iq nb)) =

ϱ(imtail(iq *null))=

ϱ(imtail(*paar(ar na) *null))

= /nach Def. von imtail

ϱ(paar(ar sub(na *null)))

= /Arithmetik

ϱ(paar(ar na))

= /Evaluierung

ϱ(*paar(ar na))

tail(ϱ(iq) ϱ(nb)) =

tail(ϱ(iq) ϱ(*null)) =

tail(ϱ(*paar(ar na)) ϱ(*null))

= /nach Def. von ϱ, da *null ∉ imq

tail(ϱ(*paar(ar na)) *null)

= /nach Def. von tail

ϱ(*paar(ar na))

/*Ende des Beweises zum Induktionsanfang */

<u>Induktionsannahme</u> (*null ≤ nb ≤ nn)

Folgendes sei bewiesen für *null ≤ nb ≤ nn:

∀iq∈imq\{error.imq}. iq = *paar(ar na). nb∈[0 : nn].
ϱ(imtail(iq nb)) = tail(ϱ(iq) ϱ(nb)).

<u>Induktionsschritt</u> (nb = *succ(nn))

Für nb = *succ(nn) ist zu zeigen:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na). nb = *succ(nn).:
      ς(imtail(iq nb))  =  tail(ς(iq) ς(nb))


ς(imtail(iq nb)) =
ς(imtail(iq *succ(nn))) =
ς(imtail(*paar(ar na) *succ(nn)))
  =                                   /nach Def. von imtail
ς(*paar(ar sub(na *succ(nn))))
  =                                             /Arithmetik
ς(*paar(ar sub(pred(na) nn)))


/*Es wird nun tail(ς(iq) ς(nb)) abgeleitet:              */


tail(ς(iq) ς(nb)) =
tail(ς(iq) ς(*succ(nn))) =
tail(ς(*paar(ar na)) ς(*succ(nn)))
  =                      /nach Def. von ς, da *succ(nn) ∉ imq
tail(ς(*paar(ar na)) *succ(nn))
  =                                      /nach Def. von tail
```



```
tail(tl(ς(*paar(ar na))) nn)
  =                                      /siehe 8.6.5. imtl
tail(ς(imtl(*paar(ar na))) nn)
  =                                          /da nn = ς(nn)
tail(ς(imtl(*paar(ar na))) ς(nn))
  =                                  /nach Induktionsannahme
ς(imtail(imtl(*paar(ar na)) nn))
  =                                     /nach Def. von imtl
ς(imtail(*paar(ar pred(na)) nn))
  =                                   /nach Def. von imtail
ς(*paar(ar sub(pred(na) nn)))
                                                        qed


/*Somit ist gezeigt, daß sich sowohl
  ς(imtail(iq nb))   alsauch
  tail(ς(iq) ς(nb))  zu
  ς(*paar(ar sub(pred(na) nn))))  ableiten lassen.

  Damit ist dann der Induktionsschritt bewiesen und somit
  der Beweis bzgl. imtail beendet.

  Ende des Beweises bzgl. imtail.                        */
```

8.6.12. **imsubst**

zu zeigen:

∀iq,iq'∈imq\{error.imq}. iq = *paar(ar na).
iq' = *paar(ar' na'). ∀nb∈nat\{error.nat}.:

ϱ(imsubst(iq iq' nb)) = subst(ϱ(iq) ϱ(iq') ϱ(nb))

<u>Beweis</u> :

Induktionsbeweis über nb

<u>Induktionsanfang</u> (nb = *null)

zu zeigen:

∀iq,iq'∈imq\{error.imq}. iq = *paar(ar na).
iq' = *paar(ar' na'). nb = *null.:

ϱ(imsubst(iq iq' nb)) = subst(ϱ(iq) ϱ(iq') ϱ(nb))

ϱ(imsubst(iq iq' nb)) =

ϱ(imsubst(iq iq' *null))

= /nach Def. von imsubst

ϱ(imcat(iq' imtail(iq *null)))

= /nach Def. von imtail

ϱ(imcat(iq' iq))

subst(ϱ(iq) ϱ(iq') ϱ(nb)) =

subst(ϱ(iq) ϱ(iq') ϱ(*null))

= /nach Def. von ϱ, da *null ∉ imq

subst(ϱ(iq) ϱ(iq') *null)

= /nach Def. von subst

cat(ϱ(iq') ϱ(iq))

= /siehe 8.6.7. imcat

ϱ(imcat(iq' iq))

qed

/*Ende des Beweises zum Induktinsanfang */

<u>Induktionsannahme</u> (*null ≤ nb ≤ nn)

Folgendes sei bewiesen für *null ≤ nb ≤ nn:

∀iq,iq'∈imq\{error.imq}. iq = *paar(ar na).
iq' = *paar(ar' na'). nb∈[0 : nn].

ϱ(imsubst(iq iq' nb)) = subst(ϱ(iq) ϱ(iq') ϱ(nb))

Induktionsschritt (nb = *succ(nn))

Für nb = *succ(nn) ist zu zeigen:

```
∀iq,iq'∈imq\{error.imq}. iq = *paar(ar na).
iq' = *paar(ar' na'). nb = *succ(nn).:

  ϱ(imsubst(iq iq' nb))  =  subst(ϱ(iq) ϱ(iq') ϱ(nb)).


ϱ(imsubst(iq iq' nb)) =

ϱ(imsubst(iq iq' *succ(nn)))

  =                              /nach Def. von imsubst

ϱ(imcat(iq' imtail(iq *succ(nn))))

/*Es wird nun subst(ϱ(iq) ϱ(iq') ϱ(nb)) abgeleitet:      */


subst(ϱ(iq) ϱ(iq') ϱ(nb)) =

subst(ϱ(iq) ϱ(iq') ϱ(*succ(nn)))

  =                /nach Def. von ϱ, da *succ(nn) ∉ imq

subst(ϱ(iq) ϱ(iq') *succ(nn))

  =                              /nach Def. von subst

subst(tl(ϱ(iq)) ϱ(iq') nn)

  =                                /siehe 8.6.5. imtl

subst(ϱ(imtl(iq)) ϱ(iq') nn)

  =                   /nach Def. von ϱ, da nn = ϱ(nn)

subst(ϱ(imtl(iq)) ϱ(iq') ϱ(nn))

  =                           /nach Induktionsannahme
```



```
ϱ(imsubst(imtl(iq) iq' nn))

  =                              /nach Def. von imsubst

ϱ(imcat(iq' imtail(imtl(iq) nn)))

  =                   /nach einer Property bzgl. imtail

ϱ(imcat(iq' imtail(iq succ(nn))))

  =                                       /Evaluierung

ϱ(imcat(iq' imtail(iq *succ(nn))))                 qed



/*Ende des Beweises zum Induktionsschritt und somit auch

  Ende des Beweises bzgl. imsubst                     */
```

8.6.13. imhead

zu zeigen:

∀iq∈imq\{error.imq}. ∀nb∈nat\{error.nat}.
iq = *paar(ar na).:
    ϱ(imhead(iq nb)) = head(ϱ(iq) ϱ(nb))

Für nb > na gilt:
imhead(iq nb) = error.imq, und somit ergibt sich auch:
ϱ(imhead(iq nb)) = error.qprog.

Ebenso gilt für nb > na = laenge(qp) auch:
head(qp nb) = error.qprog

womit also auch für nb > na gewährleistet, daß gilt:
    ϱ(imhead(iq nb)) = head(ϱiq) ϱ(nb))

Für die nun folgende Beweisführung wird nb ≤ na vorausgesetzt:

Beweis :

Induktionsbeweis über nb

Induktionsanfang (nb = *null)

zu zeigen:

∀iq∈imq\{error.imq}. iq = *paar(ar na). nb = *null.:

    ϱ(imhead(iq nb)) = head(ϱ(iq) ϱ(nb))

ϱ(imhead(iq nb)) =

ϱ(imhead(iq *null))

= /nach Def. von imhead

ϱ(imnil) =

ϱ(*paar(*leerarray *null))

= /nach Def. von ϱ

rep.qprog(*paar(*leerarray *null))

= /nach Def. von rep.qprog

nil

/*Es wird nun head(ϱ(iq) ϱ(nb)) abgeleitet: */

head(ϱ(iq) ϱ(nb)) =

head(ϱ(iq) ϱ(*null))

= /nach Def. von ϱ, da *null ∉ imq

head(ϱ(iq) *null)

= /nach Def. von head

nil

/*Ende des Beweises zum Induktionsanfang */

<u>Induktionsannahme</u> (*null < nb ≤ nn ≤ na)

Folgendes sei bewiesen für *null < nb ≤ nn ≤ na:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na).
∀nb∈[1 : nn].:
     ϱ(imhead(iq nb))  =  head(ϱ(iq) ϱ(nb)).
```

<u>Induktionsschritt</u> (nb = *succ(nn) ≤ na)

Für nb = *succ(nn) ≤ na ist zu zeigen:

```
∀iq∈imq\{error.imq}. iq = *paar(ar na). nb = *succ(nn).:

     ϱ(imhead(iq nb))  =  head(ϱ(iq) ϱ(nb))
```



```
ϱ(imhead(iq nb)) =

ϱ(imhead(iq *succ(nn))) =

ϱ(imhead(*paar(ar na) *succ(nn)))

  =                                        /nach Def. von imhead

ϱ(*paar(subarray(ar sub(plus(ug(ar) na) *succ(nn))
                pred(plus(ug(ar) na))) *succ(nn)))

  =                                                  /Arithmetik

ϱ(*paar(subarray(ar pred(sub(plus(ug(ar) na) nn))
                pred(plus(ug(ar) na))) *succ(nn)))

  =                                               /nach Def. von ϱ

rep.qprog(*paar(subarray(ar pred(sub(plus(ug(ar) na) nn))
                     pred(plus(ug(ar) na))) *succ(nn)))

  =      /nach Def. von rep.qprog und wegen Lemmata 1, 2 und 3

cons(imhd(*paar(subarray(ar pred(sub(plus(ug(ar) na) nn))
                     pred(plus(ug(ar) na))) *succ(nn)))
     rep.qprog(imtl(*paar(subarray(ar pred(sub(plus(ug(ar) na)
                                         nn))
                              pred(plus(ug(ar) na)))
                     *succ(nn)))))

  =                                          /nach Def. von imtl

cons(imhd(*paar(subarray(ar pred(sub(plus(ug(ar) na) nn))
                     pred(plus(ug(ar) na))) *succ(nn)))
     rep.qprog(*paar(subarray(ar pred(sub(plus(ug(ar) na) nn))
                           pred(plus(ug(ar) na))) nn)))
```

/*Auf dieses Ableitungsergebnis wird später weiter eingegangen; es wird nun head(ϱ(iq) ϱ(nb)) abgeleitet: */

```
head(ϱ(iq) ϱ(nb)) =

head(ϱ(iq) ϱ(*succ(nn))) =

head(ϱ(*paar(ar na)) ϱ(*succ(nn)))

  =                           /nach Def. von ϱ, da *succ(nn) ∉ imq

head(ϱ(*paar(ar na)) *succ(nn))

  =                                          /nach Def. von head

cons(hd(ϱ(*paar(ar na))) head(tl(ϱ(*paar(ar na))) nn))

  =                                           /siehe 8.6.3. imhd

cons(ϱ(imhd(*paar(ar na))) head(tl(ϱ(*paar(ar na))) nn))

  =                                           /siehe 8.6.5. imtl

cons(ϱ(imhd(*paar(ar na))) head(ϱ(imtl(*paar(ar na))) nn))

  =                                          /nach Def. von imtl

cons(ϱ(imhd(*paar(ar na))) head(ϱ(*paar(ar pred(na))) nn))

  =                                /nach Def. von ϱ, da nn ∉ imq

cons(ϱ(imhd(*paar(ar na))) head(ϱ(*paar(ar pred(ar))) ϱ(nn)))

  =                                      /nach Induktionsannahme

cons(ϱ(imhd(*paar(ar na))) ϱ(imhead(*paar(ar pred(na)) nn)))

  =                                        /nach Def. von imhead

cons(ϱ(imhd(*paar(ar na)))
     ϱ(*paar(subarray(ar sub(plus(ug(ar) pred(na)) nn)
                      pred(plus(ug(ar) pred(na)))) nn)))

  =                                                 /Arithmetik
```



```
cons(ϱ(imhd(*paar(ar na)))
     ϱ(*paar(subarray(ar pred(sub(plus(ug(ar) na) nn))
                         pred(pred(plus(ug(ar) na)))) nn)))

  =                                               /nach Def. von ϱ

cons(ϱ(imhd(*paar(ar na)))
     rep.qprog(*paar(subarray(ar pred(sub(plus(ug(ar) na)
                                           nn))
                                  pred(pred(plus(ug(ar) na))))
                  nn)))
```

/*Im folgenden werden nun die beiden Ableitungsergebisse von ϱ(imhead(iq nb)) und head(ϱ(iq) ϱ(nb)) auf Äquivalenz überprüft.
Das als erstes abgeleitete Ergebnis (vorletzte Seite) wird im folgenden als A1, das zuletzt abgeleitete als A2 angesprochen.

Die zu zeigende Äquivalenz wird in zwei Schritten nachgewiesen, nämlich:

1.) Nachweis der Gleichheit der jeweils ersten Argumente der cons- Ausdrücke von A1 und A2.

2.) Nachweis der Gleichheit der jeweils zweiten Argumente der cons- Ausdrücke von A1 und A2.

*/

zu 1.)

zu zeigen:

```
imhd(*paar(subarray(ar pred(sub(plus(ug(ar) na) nn))
                       pred(plus(ug(ar) na))) *succ(nn)))
  =

ϱ(imhd(*paar(ar na)))
```

```
imhd(*paar(subarray(ar pred(sub(plus(ug(ar) na) nn))
                   pred(plus(ug(ar) na))) *succ(nn)))

  =                                          /nach Def. von imhd

lese(subarray(ar pred(sub(plus(ug(ar) na) nn))
              pred(plus(ug(ar) na)))
     pred(plus(ug(subarray(ar pred(sub(plus(ug(ar) na) nn))
                              pred(plus(ug(ar) na))))
          *succ(nn))))

  =

/*Nach Definition der Operation subarray gilt:

  ug(subarray(ar n1 n2))  =  n1

  Diese Eigenschaft führt zur folgenden Ableitung:          */

  =

lese(subarray(ar pred(sub(plus(ug(ar) na) nn))
              pred(plus(ug(ar) na)))
     pred(plus(pred(sub(plus(ug(ar) na) nn)) *succ(nn))))

  =                                              /Arithmetik

lese(subarray(ar pred(sub(plus(ug(ar) na) nn))
              pred(plus(ug(ar) na)))
     pred(plus(ug(ar) na)))

  =

/*Aufgrund der Definition von subarray besteht ein über die
  Ausführung von subarray(ar n1 n2) , mit n1 ≥ ug(ar) und
  n2 ≤ og(ar), entstandener Term ausschließlich aus Subtermen
  von ar.
  Daraus folgt trivialerweise, daß alle Subterme von
  subarray(ar n1 n2)  auch Subterme von ar sind.

  Somit gilt offensichtlich:
  ∀nb∈[n1 : n2 ].:
      lese(subarray(ar n1 n2) nb)  =  lese(ar nb)
  Nach dieser Überlegung läßt sich folgender Ausdruck ab-
  leiten:                                                    */
```



```
  =

lese(ar pred(plus(ug(ar) na )))


/*Es wird nun das erste Argument des cons- Ausdrucks von A2
  abgeleitet:
                                                             */
ς(imhd(*paar(ar na )))

  =                                          /nach Def. von imhd

ς(lese(ar pred(plus(ug(ar) na ))))

  =              /nach Def. von ς, da lese(ar pred(....)) ∉ imq

lese(ar pred(plus(ug(ar) na )))

/*Somit ist nun die Äquivalenz der jeweils ersten Argumente
  der beiden cons- Ausdrücke von A1 und A2 nachgewiesen.

  Es wird nun die Äquivalenz der jeweils zweiten Argumente
  der beiden cons- Ausdrücke von A1 und A2 gezeigt:          */


zu 2.)

zu zeigen:

rep.qprog(*paar(subarray(ar pred(sub(plus(ug(ar) na) nn))
                            pred(plus(ug(ar) na))) nn))
  =

rep.qprog(*paar(subarray(ar pred(sub(plus(ug(ar) na) nn))
                            pred(pred(plus(ug(ar) na)))) nn))
```

Abkürzungen:

Es sei im folgenden:

```
ar' = subarray(ar pred(sub(plus(ug(ar) na) nn))
                  pred(plus(ug(ar) na)))

ar*= subarray(ar pred(sub(plus(ug(ar) na) nn))
                  pred(pred(plus(ug(ar) na))))

iq' = *paar(ar' nn)

iq* = *paar(ar* nn)
```

/*Der nun folgende **Beweis** besteht darin, zu zeigen, daß für iq' und iq* die Voraussetzungen des Hauptsatzes der Implementierung (siehe 8.4.) erfüllt sind, sodaß damit die Aussage dieses Satzes für iq' und iq* zutrifft, die genau das besagt, was hier bewiesen werden soll, nämlich:
ϱ(iq') = ϱ(iq*) . */

Nach Festlegung gilt:

a.) iq' = *paar(ar' nn)

iq* = *paar(ar* nn) mit nn∈nat\{error.nat}

Außerdem wird gezeigt:

b.) 1.) ϱ(iq') ≠ error.qprog

2.) ϱ(iq*) ≠ error.qprog

zu b.) 1.):

i.) Nach Voraussetzung gilt:
nn∈nat\{error.nat}.

ii.) Nach Definition von ug und og gilt:

ug(ar') = pred(sub(plus(ug(ar) na) nn))

og(ar') = pred(plus(ug(ar) na))

Daraus ergibt sich:

og(ar') - ug(ar') = nn

und daraus wiederum folgt:

not(lt(succ(sub(og(ar') ug(ar'))) nn)) = true

iii.) Es gilt:

```
∀i∈[0 : nn-1].:
           is_def(ar' plus(ug(ar') i)) = true
```

Diese Aussage folgt direkt aus der Definition von ar':
ar' = subarray(ar ug(ar') og(ar'))
und aus der generellen Voraussetzung für alle Beweise, die besagt:
ϱ(iq) ≠ error.qprog mit iq = *paar(ar na),
woraus das vollständige "Definiertsein" von ar und damit auch von ar' folgt (als Umkehrung von Lemma 3).

iv.) Nach Definition von ϱ gilt:

ϱ(*paar(ar' nn)) = rep.qprog(*paar(ar' nn))

und wegen i.), ii.) und iii.) gilt:

rep.qprog(*paar(ar' nn)) ≠ error.qprog.

zu b.) 2.):

i.) Nach Voraussetzung gilt:
nn∈nat\{error.nat}.

ii.) Nach Definition gilt:

ug(ar*) = pred(sub(plus(ug(ar) na) nn))

og(ar*) = pred(pred(plus(ug(ar) na)))

Somit ergibt sich:

og(ar*) - ug(ar*) = pred(nn)

und daraus folgt:

not(lt(succ(sub(og(ar*) ug(ar*))) nn)) = true

iii.) Es gilt:
∀i∈[0 : nn-1].:
is_def(ar* plus(ug(ar*) i)) = true
(Argumentation äquivalent zu b.) 1.) iii.))

iv.) ϱ(*paar(ar* nn)) = rep.qprog(*paar(ar* nn))

und wegen i.), ii.) und iii.) gilt:

rep.qprog(*paar(ar* nn)) ≠ error.qprog

/*Die Vereinigung der Aussagen zu b.)1.) und b.)2.) entspricht der Voraussetzung VS1 des Hauptsatzes der Implementierung.

Es müssen nun noch äquivalent zum Hauptsatz die beiden Fälle unterschieden werden: */

Fall 1:

nn = *null

Aussage des Hauptsatzes:

VS1 ∧ nn = *null.: ϱ(*paar(ar nn)) = ϱ(*paar(ar′ nn)

ϱ(*paar(ar nn)) =

ϱ(*paar(ar *null)) =

rep.qprog(ar *null)) =

nil

Äquivalent dazu läßt sich auch ableiten:

ϱ(*paar(ar nn)) = nil

Damit ist also für nn = *null gezeigt, daß gilt:

ϱ(iq′) = ϱ(iq*)

Fall 2:

```
nn > null

Aussage des Hauptsatzes:

VS1 ∧ na > null ∧ ∀k∈[0 : na-1 ].
lese(ar plus(ug(ar) k))  =  lese(ar' plus(ug(ar') k)).:

            ς(*paar(ar na))  =  ς(*paar(ar' na))
```

VS1 ist wie schon gezeigt für iq' und iq* erfüllt, und ebenso trifft nn > null zu.
Es bleibt nur zu zeigen:

```
∀k∈[0 : nn-1].

lese(ar' plus(ug(ar') k))
  =
lese(ar* plus(ug(ar*) k))
```

Nach Definition gilt:

```
ug(ar')  =  pred(sub(plus(ug(ar) na) nn))

ug(ar*)  =  pred(sub(plus(ug(ar) na) nn))

also gilt:
ug(ar')  =  ug(ar*).
```


Nach Definition von ar' und ar* als subarray(ar x y) gilt aufgrund der schon früher erwähnten Eigenschaft des Enthaltenseins aller Subterme von subarray(ar x y) in ar:

```
∀i∈[0 : nn-1].:

lese(ar' plus(ug(ar') i)) = lese(ar plus(ug(ar') i))
und
lese(ar* plus(ug(ar*) i))  =  lese(ar plus(ug(ar*) i))

wegen ug(ar')  =  ug(ar*)  gilt:
lese(ar plus(ug(ar') i))  =  lese(ar plus(ug(ar*) i)).
```

Somit ist dann bewiesen, daß gilt:

```
∀k∈[0 : nn-1].:

lese(ar' plus(ug(ar') k))  =  lese(ar* plus(ug(ar*) k))  qed
```

Damit ist also auch für Fall 2 nachgewiesen, daß für iq' und iq* die zur Anwendung des Hauptsatzes notwendigen Bedingungen erfüllt sind.

Fall 1 und Fall 2 zusammengenommen bringen dann das gewünschte Ergebnis: ς(iq') = ς(iq*).

Damit ist dann auch der Induktionsschritt bewiesen und somit der Beweis bzgl. imhead beendet.

```
/*Ende des Beweises bzgl. imhead
  und damit auch:


  Ende des Beweises der Korrektheit von I_QUELLPROG.         */
```

9. Schlußbetrachtung

Die vorliegende Arbeit sollte als eine der ersten Anwendungen der im Rahmen des Projektes "Programmverifikation" an der Universität Bonn entwickelten Methoden die Praktikabilität dieser Verfahren zeigen.

Die formale Spezifikation in Kapitel 5 ist auf den ersten Blick hin im Verhältnis zu dem gestellten Problem sehr umfangreich. **Es ist allerdings zu beachten,** daß wir keine vorgegebenen abstrakten Datentypen zur Verfügung hatten und daher alle benötigten ADT's spezifizieren mußten. Geht man davon aus, daß bei späteren Arbeiten auf umfangreiche Datentyp- Bibliotheken zurückgegriffen werden kann, so wäre dann der Aufwand für diese Spezifikationen im Vergleich zu der vorliegenden wesentlich geringer. Bei gleichzeitigem Einsatz eines intelligenten Editors würde auch der reine Schreibaufwand nicht mehr so sehr ins Gewicht fallen.

Der Korrektheitsbeweis der Implementierung in Kapitel 8 nimmt mit Abstand den größten Platz in dieser Arbeit ein.
Der Hauptgrund dafür ist die große Anzahl von Operationen in QUELLPROG, deren korrekte Implementierung jeweils separat gezeigt werden mußte.
Die Beweise zu den einzelnen Operationen sind sich jedoch sehr ähnlich. Sie werden entweder direkt durch Evaluierung der Ausdrücke zu ihren Termen oder durch Induktion über die Länge der Terme oder über den Wert eines Argumentes der Sorte 'nat' geführt.
Diese Art von Beweisen **ist sicherlich schematisierbar** und könnte also auch mit Hilfe eines automatischen Beweisers durchgeführt werden, sodaß diese aufwendige Arbeit ebenfalls wesentlich verringert werden kann.
Bei vorhandenen Wissensbasen kann man evtl. auf bereits verifizierte Sätze und/oder Lemmata zurückgreifen, sodaß auch deren Entwicklung und Beweis entfallen könnte.
Mit allen oben genannten Hilfsmitteln würde sich also zumindest der formalistische Aufwand erheblich verringern. Wie unsere Arbeit zeigt, ist bei weniger komplexen Problemen, wie etwa dem vorliegenden, jedoch auch eine Lösung "per Hand", d.h. ohne die o.g. Hilfsmittel, möglich.

Wir haben in Kapitel 7 einen Implementierungsschritt durchgeführt. Als weitere Schritte sind denkbar :

- Implementierung mehrerer verschiedener abstrakter Datentypen durch einen neuen ADT
  z.B. Implementierung von NOKOMMENTAR, NOREDBLANK, NOCOMPAN und QPSAUBER durch einen konkreteren abstrakten Datentyp, der die Anzahl der "Programmdurchläufe" reduziert
- Abstieg von der Termebene (mathematische Ebene) zur Zustandsebene (Programmebene) .

Zur Erstellungszeit dieser Arbeit lagen für die o.g. Implementierungsschritte noch keine ausgereiften Implementierungskonzepte vor. Sie sind jedoch realisierbar und werden wohl in einiger Zeit zur Anwendung bereit stehen.

Der Übergang von der Termebene zur Zustandsebene wird sich wesentlich vereinfachen, wenn sich die Anwendung von funktionalen Programmiersprachen durchsetzt und vielleicht sogar durch geänderte Rechner-Architekturen Programme in funktionaler Sprache direkt interpretiert werden können, sodaß eine Zustandsebene im bisherigen Sinn nicht mehr von Bedeutung ist.

Literaturverzeichnis

[ADI 82] ADI-Verband (Herausgeber);
"Qualitätssicherung bei Software :
Mehr Engagement in den frühen Phasen"
erschienen in ONLINE, Journal für Informationsverarbeiter,
S.30-33, Köln, Mai 1982

[Bei 81a] Beierle, Christoph;
"Rewrite Rule Systems"
Universität Bonn, FB Informatik, interner Bericht, Feb. 1981

[Bei 81b] Beierle, Christoph;
"INTAKT"
Universität Bonn, FB Informatik, interner Bericht, Mai 1981

[Bei 82a] Beierle, Christoph;
"Einführung in die Bonner Spezifikationssprache"
Universität Bonn, FB Informatik,
Treffen Programmverifkations-Projekt, 21/22.1.1982

[Bei 82b] Beierle, Christoph;
"TRIPLEX-Implementations"
Universität Bonn, FB Informatik, interner Bericht, Feb. 1982

[Bei 82c] Beierle, Christoph;
Persönliches Gespräch mit Dipl.-Inform. Christoph Beierle,
Universität Bonn, FB Informatik, Institut III, März 1982

[BGO 82] Beierle, C.; Guntram, U.; Oberdörster, W.;
Raulefs, P.; Voß, Angelika.;
"The CTA-Approach to the Specification of Abstract Data Types"
Universität Bonn, FB Informatik, interner Bericht (vorläufige Version), März 1982

[BGS 81] Beierle, C.; Grieneisen, H.; Schmidt, P.;
"Spezifikation von INTAKT" (Version 1)
Universität Bonn, FB Informatik, Juli 1980

[EKP 78] Ehrig, H.; Kreowski, H.-J.; Padawitz, P.;
"Stepwise Specification and Implementation of Abstrakt Data Types"
TU Berlin, FB Informatik, März 1978

[EKP 79] Ehrig, H.; Kreowski, H.-J.; Padawitz, P.;
"Algebraische Implementierung abstrakter Datentypen"
TU Berlin, FB Informatik, Bericht, März 1979

[Grie 81] Grieneisen, Hartmut;
"Bemerkungen zu INTAKT"
Universität Bonn, FB Informatik, interner Bericht, Mai 1981

[Grie 82] Grieneisen, Hartmut;
"Eine algebraische Spezifikation des Software-Produkts INTAKT"
Universität Bonn, FB Informatik, Diplom-Arbeit, August 1982

[Hrus 82] Hruschka, P.;
"Neue Programmiersprachen im Wettstreit"
Vortrag von Dipl.-Ing. Dr. P. Hruschka, ADI-Veranstaltung,
Köln, März 1982

[Kreo 79] Kreowski, Hans-Jörg;
"Algebra für Informatiker"
TU Berlin, FB Informatik, schriftliches Material zur gleichnamigen Lehrveranstaltung im WS 1978/79

[Loe 81a] Loeckx, Jacques;
"Implementations of Abstract Data Types and Their Verification"
Universität des Saarlandes, FB 10-Informatik, Saarbrücken,
interner Bericht, April 1981

[Loe 81b] Loeckx, Jacques;
"Algorithmic Specifications :
A new specification method for abstract data types"
Universität des Saarlandes, FB 10-Informatik, Saarbrücken,
interner Bericht, Dezember 1981

[Muss 78] Musser, David R.;
"A Data Type Verification System based on Rewrite Rules"
University of Southern Carlifornia, USC Information Sciences Institut, June 1978

[Muss 80] Musser, David R.;
"On Proving Inductive Properties of Abstract Data Typs"
USC Information Sciences Institut,
aus 7th ACM Principles of Programming Languages Proceedings

[Pada 79] Padawitz, Peter;
"Proving the Correctness of Implementation by Exclusive Use of Term Algebras"
TU Berlin, FB Informatik, Bericht, June 1979

[Raul 79] Raulefs, Peter;
"Einführung in die Theorie der Datenstrukturen"
Universität Bonn, FB Informatik, Vorlesungsnotizen, SS 1979

[RoLe 77] Robinson, Lawrence; Levitt, Karl N.;
"Proof Technics for Hierarchically Structured Programs"
Standford Research Institut,
Herausgeber G. Manacher, S.L. Graham,
erschienen in Communications of the ACM, April 1977,
Volume 20, Number 4

[Sch 81a] Schmidt, Peter;
"Grundlagen zur Spezifikation des Aufbereitungsteils von INTAKT"
Universität Bonn, FB Informatik, interner Bericht, Aug. 1981

[Sch 81b] Schmidt, Peter;
Persönliches Gespräch mit Dr. Peter Schmidt,
Universität Bonn, FB Informatik, Institut III, Oktober 1981

[SIEM 80] SIEMENS Arbeitsgruppe SW-Spezifikationsmethoden/ Test- und Qualitätskontrollsysteme
"INTAKT, Interaktiver Arbeitsplatz zur Qualitätskontrolle von Software"
Funktionskatalog, München, Oktober 1980

[SPL 80]
"SPL3/BS2000 User's Guide"
herausgegeben von DvST SP31 Mch H/TQ, 3.Edition, May 1980

[SRL 79] Silverberg, B.A.; Robinson, L.; Levitt, K.N.;
"The HDM Handbook
Volume II : The Languages and Tools of HDM"
SRI Projekt 4828, June 1979

[Voß 81a] Voß, Angelika;
"TRIPLEX"
Universität Bonn, FB Informatik, interner Bericht, Nov. 1981

[Voß 81b] Voß, Angelika;
"Simple Spec Sysytems"
Universität Bonn, FB Informatik, interner Bericht, Okt. 1981