Hochschule Wismar
Fachhochschule für Technik, Wirtschaft und Gestaltung
Fachbereich Wirtschaft

Hochschule Wismar
University of Technology, Business and Design
Department of Business

Harald Mumm

Entwurf und Implementierung einer objektorientierten Programmiersprache für die Paula-Virtuelle-Maschine

Heft 08 / 2003

W D P

Wismarer Diskussionspapiere / Wismar Discussion Papers

Der Fachbereich Wirtschaft der Hochschule Wismar, Fachhochschule für Technik, Wirtschaft und Gestaltung bietet die Studiengänge Betriebswirtschaft, Management sozialer Dienstleistungen, Wirtschaftsinformatik und Wirtschaftsrecht an. Gegenstand der Ausbildung sind die verschiedenen Aspekte des Wirtschaftens in der Unternehmung, der modernen Verwaltungstätigkeit im sozialen Bereich, der Verbindung von angewandter Informatik und Wirtschaftswissenschaften sowie des Rechts im Bereich der Wirtschaft.

Nähere Informationen zu Studienangebot, Forschung und Ansprechpartnern finden Sie auf unserer Homepage im World Wide Web (WWW): http://www.wi.hs-wismar.de/.

Die Wismarer Diskussionspapiere / Wismar Discussion Papers sind urheberrechtlich geschützt. Eine Vervielfältigung ganz oder in Teilen, ihre Speicherung sowie jede Form der Weiterverbreitung bedürfen der vorherigen Genehmigung durch den Herausgeber.

Herausgeber: Prof. Dr. Jost W. Kramer
Fachbereich Wirtschaft
Hochschule Wismar
Fachhochschule für Technik, Wirtschaft und Gestaltung
Philipp-Müller-Straße
Postfach 12 10
D – 23966 Wismar
Telefon: ++49 / (0)3841 / 753 441
Fax: ++49 / (0)3841 / 753 131
e-mail: j.kramer@wi.hs-wismar.de

ISSN 1612-0884
ISBN 3-910102-32-8

JEL-Klassifikation Z00

Alle Rechte vorbehalten.

© Hochschule Wismar, Fachbereich Wirtschaft, 2003.
Printed in Germany

Inhaltsverzeichnis

1 Zielstellung

Internet und Java sind moderne Schlagwörter der aktuellen DV-Technik. Die Beherrschung von Java als objektorientierte Programmiersprache ist für Studierende der Wirtschaftsinformatik unbedingt erforderlich, wenn sie in der Praxis eine schnelle Akzeptanz erreichen wollen.

Es wird in dieser Arbeit folgende Begriffsbildung verwendet. **Objekte** sind identifizierbare Einheiten der realen Welt, über die informiert werden soll. Ein *klassenbildendes Prädikat* führt zur **Klassen- oder Mengenbildung** derjenigen Objekte, für die dieses Prädikat wahr ist. Für die Objekte einer Klasse werden zweckmäßige **Merkmale** ausgesucht, die eine *Zuordnung* (Funktion) der Menge der Objekte in eine *Wertemenge* vornehmen. Die aneinandergereiten Werte der Merkmale werden auch *Wert des Objektes* genannt. Im folgenden wird zwischen Objekt und *Objektwert* nicht mehr unterschieden. Es ist klar, dass nicht Objekte im Computer abgespeichert werden können sondern nur Objektwerte. Zu jedem Merkmal einer Klasse korrespondiert i.allg. ein *Feld* im Objekt(wert), dass gerade den Wert dieses Merkmals für das Objekt beinhaltet.

Der Autor hat bei der Einführungsveranstaltung *Informatik* beobachtet, dass Studierende Schwierigkeiten haben, die bei der objektorientierten Programmierung auftretenden Teilprobleme, wie

1. die Verknüpfung von Datenbeschreibungen und Methoden von Objekten zu Klassen,
2. die Erstellung zweckmäßiger Methoden–Schnittstellen,
3. die Unterscheidung zwischen statischem und dynamischem Typ einer (Zeiger-) Variablen und
4. die Erstellung von polymorphem Code

zu lösen.

Das liegt zum einen daran, dass viele Studierende Schwierigkeiten bei der Modellierung, also der Abbildung von der Realität in Modelle der Mathematik und Informatik haben, aber auch zum anderen an der komplexen Syntax und Semantik der kommerziellen Programmiersprachen.

Das Ziel dieser Arbeit besteht in der Bereitstellung einer einfachen, objektorientierten Entwurfssprache, die es den Studierenden im Grundstudium schneller als mit kommerziellen Programmiersprachen ermöglicht, das Programmieren im neuen Stil, besonders mit Polymorphie, zu erlernen.

Zur Erreichung dieses Ziels waren folgende Aufgaben zu lösen:

1. Die Analyse vorhandener objektorientierter Programmiersprachen,
2. das Erfinden von zweckmäßigen Erweiterungen für die als Grundlage ausgewählte Entwurfssprache Anton[1],

[1] Christian Steinmann.

3. die Beschreibung der Syntax mittels Syntaxdiagrammen,

4. die Entwicklung des Compilers (Scanner und Parser) der neuen Sprache, um ihre Semantik präzise und offen zeigen zu können,

5. die Entwicklung eines Programmverbinders,

6. die Entwicklung einer Laufzeitumgebung.

In Anlehnung an die bereits genutzte Entwurfsprache Anton soll die Neuentwicklung OO-Anton (objektorientierter Anton) genannt werden.

Diese Arbeit geht davon aus, dass die virtuelle Maschine Paula[2] (PVM) incl. Animationsprogramm[3] vorhanden sind.
Der Parser wurde nach der Methode des rekursiven Abstiegs entwickelt. Die Semantik der Sprache wurde durch Abbildung auf die Anweisungen der PVM gefunden.
Die Laufzeitumgebung besteht aus einem Prozessverwalter, einer Speicher- und einer Trap-Task und der Anwendung.

1.1 Offene Probleme

In Anbetracht der für das Vorhaben zur Verfügung stehenden relativ kurzen Zeit konnten einige Probleme noch nicht gelöst werden, wie

1. der Test, ob eine homonym-Methode (eine Methode, deren Name nicht eindeutig sein soll) wirklich eine zweite Implementierung hat,

2. die Typzusicherung (casting),

3. die Prüfung der Zuweisungskompatibilität zwischen Objekten,

4. die Mehrfachverwendung von Klassenkomponentennamen (Namen für Methoden und Merkmale) in verschiedenen Klassenhierarchien.

1.2 Besondere Vorzüge

Nach der Aufzählung der offenen Probleme soll aber nicht unerwähnt bleiben, dass OO-Anton auch über Leistungen verfügt, die bei kommerziellen Sprachen, wie Turbo-Pascal nicht möglich sind. So kann z.B. der Rückgabetyp von Methoden ein Aggregattyp (Reihe) oder eine Klasse sein. Außerdem besitzt OO-Anton Konzepte für die Systemprogrammierung, wie Tasks, den Datentyp `ganzzahlImAkku` oder die Anweisung `aktiviereProzess`. Dadurch können mit OO-Anton auch Systemprogramme, wie der Betriebssystemkern oder Dienstprogramme des Betriebssystems, hier Tasks genannt, 100% problemorientiert formuliert werden.

2 Ein Beispiel

Zur Veranschaulichung wird in dieser Arbeit durchgängig das folgende Beispiel verwendet. Es kommt thematisch aus der Grafikprogrammierung und geht von zwei Figuren aus: Dem einfachen und dem schraffierten Rechteck. Ein Rechteck wird durch einen

[2] Mit freundlicher Genehmigung von H. Röck, Rostock.
[3] J. Zimmermann, Rostock.

Bezugspunkt und seine Größe beschrieben. Beim schraffierten Rechteck kommt noch eine Codierung der Schraffur hinzu.

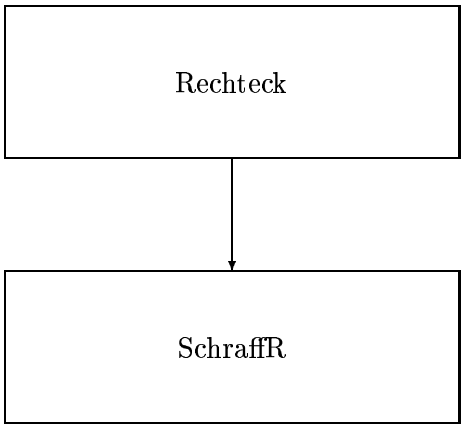


Abbildung 1: Die Objekthierarchie des Beispielprogrammes

(Der Pfeil in der Abbildung 1 symbolisiert eine Spezialisierung der Klasse am Pfeilanfang durch die Klasse am Pfeilende.)

Beide Klassen besitzen neben einem Konstruktor zur Initialisierung noch jeweils eine Methode `zeichne`, deren Name ein homonym ist, d.h. zu diesem Namen gibt es zwei Implementierungen, eine in der Klasse `Rechteck` und eine in der Klasse `SchraffR`. Diese Methode besitzt den Parameter `Art`, der für zwei Arten von Zeichnen-Operationen steht: Schwarz auf weiß bzw. weiß auf schwarz. Die erste Art symbolisiert ein visuelles Zeichnen und die zweite Art steht für ein "Wegradieren". (Da die Paula-Virtuelle-Maschine nicht zeichnen kann, wird hier nur eine Zahl ausgegeben.) Die Klasse `Rechteck` besitzt auch noch die Methode `verschiebe`, die wiederum die Methode `zeichne` aufruft. Da die Klasse `SchraffR` keine eigene Methode `verschiebe` besitzt, erben Objekte dieser Klasse diese Methode aus der Klasse `Rechteck`, wodurch der Code von `verschiebe` polymorph wird, da die zeichne-Methode aus der Klasse des Objektes, SchraffR, verwendet wird.

Im Hauptprogramm des Beispiels wird eine Liste aus zwei Elementen aufgebaut:

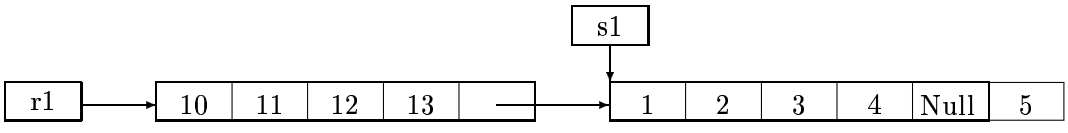


Abbildung 2: Die Datenstruktur des Beispielprogrammes

Durch die Anweisungen `reservPl(r1)` und `reservPl(s1)` wird zur Laufzeit Platz für die Objekte angefordert und reserviert. Die Platzreservierung wird durch den Speicherverwalter übernommen, der später erklärt wird.

Das problemorientierte Beispiel-Programm

```
Programm GeomD;
Typ Rechteck = Klasse (
BPunkt  : Reihe[1..2] von ganzzahl;
Groesse : Reihe[1..2] von ganzzahl;
Next    : Rechteck;

Konstruktor setRWerte(
                  -> a: ganzzahl;
                  -> b: ganzzahl;
                  -> c: ganzzahl;
                  -> d: ganzzahl;
                  -> e: Rechteck;)
                     : ganzzahl;
Beginn
 BPunkt[1]  := a;
 BPunkt[2]  := b;
 Groesse[1] := c;
 Groesse[2] := d;
 Next       := e;
 liefereAlsFunktionswert(0);
Ende;

Methode holeNext : Rechteck;
Beginn
 liefereAlsFunktionswert(Next);
Ende;

Methode zeichne(-> Art : ganzzahl;)
               : ganzzahl;homonym;
Variable Hilfe : ganzzahl;
Beginn
 gibAus(BPunkt[Art]);           (1)
 liefereAlsFunktionswert(Art);
Ende;

Methode verschiebe(
                 -> v1: ganzzahl;
                 -> v2: ganzzahl;)
                       : ganzzahl;
Variable H: ganzzahl;
Beginn
 H := zeichne(2);
 BPunkt[1] := BPunkt[1] + v1;
 BPunkt[2] := BPunkt[2] + v2;
 H := zeichne(1);
 liefereAlsFunktionswert(H);
Ende;
                         );
```

```
Typ SchraffR = Klasse erweitert
                          Rechteck (
Schraffur : ganzzahl;
Konstruktor setSWerte(
                -> a: ganzzahl;
                -> b: ganzzahl;
                -> c: ganzzahl;
                -> d: ganzzahl;
                -> e: SchraffR;
                -> s: ganzzahl;) :
                  ganzzahl;
Variable Hilfe : ganzzahl;
Beginn
 Hilfe := setRWerte(a,b,c,d,e);
 Schraffur := s;
 liefereAlsFunktionswert(0);
Ende;

Methode zeichne(-> Art : ganzzahl;) :
 ganzzahl ; homonym;
Variable rc : ganzzahl;
Beginn
 rc := Rechteck.zeichne(Art); (2)
 gibAus(Schraffur);
 liefereAlsFunktionswert(Art);(3)
Ende;
 ) ;

Variable r1  : Rechteck;
Variable lauf: Rechteck;
Variable s1  : SchraffR;
Variable rc  : ganzzahl;
Beginn
 reservPl(s1);                (4a)
 rc:= s1.setSWerte(1,2,3,4,0,5);
 reservPl(r1);                (4b)
 rc:= r1.setRWerte(10,11,12,13,s1);
                              (4c)
 lauf := r1;
 solange  lauf <> 0 wiederhole (5)
 Beginn
  rc   := lauf.zeichne(1);     (6)
  rc   := lauf.verschiebe(6,7);(7)
  lauf := lauf.holeNext;
 Ende;
 gibFreiPl(r1);
 gibFreiPl(s1);
Ende!
```

Dieses Programm erzeugt die folgenden Ausgaben von Zahlen:

Beim 1. Schleifendurchlauf kommen die Zahlen 10 (durch Anweisung 6) bzw. 11 und 16 (durch Anweisung 7) zur Ausgabe. Beim zweiten Schleifendurchlauf werden die Zahlenpaare 1,5 (durch die Anweisung 6) bzw. 2,5 und 7,5 (durch Anweisung 7) ausgegeben.

Die Compiler und Linkeraufrufe lauten:

```
compobj geomd
lass 0 3 geomd rechteck schraffr
```

Dadurch wird das folgende Maschinenprogramm generiert, das sich aus den Übersetzungen des Hauptprogrammes und der Methoden zusammensetzt.

Startadresse	Hauptprogramm	Methode	Klasse
0	GeomD		
227		setRWerte	Rechteck
296		holeNext	Rechteck
304		zeichne	Rechteck
321		verschiebe	Rechteck
438		setSWerte	SchraffR
477		zeichne	SchraffR

Abbildung 3: Zusammensetzung des Maschienprogrammes

Die folgende Tabelle fasst die Stellen im Maschinenprogramm zusammen, bei denen zu einer Methode oder einem Prozess gesprungen wird (mit vier Ausnahmen).

Zeile	Prozess	Methode	Klasse	Gerufen von
12	reservPl			Hauptprogramm
70		setSWerte	SchraffR	dito
80	reservPl			dito
116		setRWerte	Rechteck	dito
146		zeichne	Rechteck oder SchraffR	dito
165		verschiebe	Rechteck	dito
180		holeNext	Rechteck	dito
193	gibFreiPl			Hauptprogramm
213	gibFreiPl			Hauptprogramm
343		zeichne	Rechteck oder SchraffR	verschiebe
431		zeichne	Rechteck oder SchraffR	verschiebe
460		setRWerte	Rechteck	setSWerte
489		zeichne	Rechteck	zeichne(SchraffR)

Abbildung 4: Sprünge im Maschienprogramm

2.1 Die Laufzeitumgebung

Das obige Maschinenprogramm ist alleine auf der PVM (Paula–Virtuelle–Maschine) nicht ablauffähig. Es werden ein einfacher Prozessverwalter[4] und ein Speicherverwalter benötigt. (Am Ende der Arbeit wird ein einfacher Prozessverwalter problemorientiert vorgestellt.) Es wird o.B.d.A. folgende Speicheraufteilung angenommen, bei der das Maschinenprogramm ab Adresse 523 vom Lader plaziert wurde.

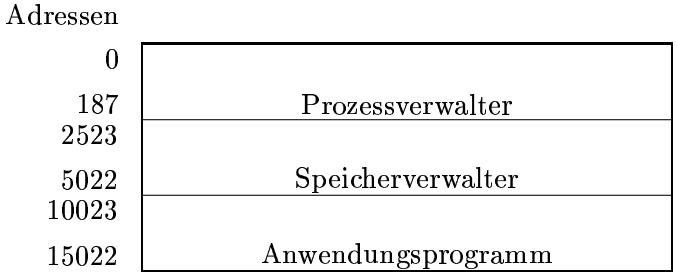


Abbildung 5: Die erste Laufzeitumgebung

Der Prozessverwalter soll in dieser Arbeit nicht intern besprochen werden, sondern nur seine Funktionalität. Bei Start des Gesamtsystems läuft zuerst der Prozessverwalter. Dieser übergibt die Steuerung an das Anwendungsprogramm. Bei den Anweisungen zur Platzreservierung auf der Halde des Anwendungsprogrammes namens `reservPl` muss der Speicherverwalter aufgerufen werden. Dieser stellt eine Adresse kleiner als 15014 und größer als 14814 bereit, weil hier willkürlich mit einer Haldengröße von 200 Plätzen gearbeitet wird.
Die Adresse wird so ausgewählt, dass der durch das Anwendungsprogramm angeforderte Platz ab der gelieferten Adresse frei ist. Die Wechsel zwischen Anwendungsprozess und Speicherverwalterprozess werden über den Prozessverwalter als "Drehscheibe" vollzogen.

[4] Prof. H. Röck, Universität Rostock.

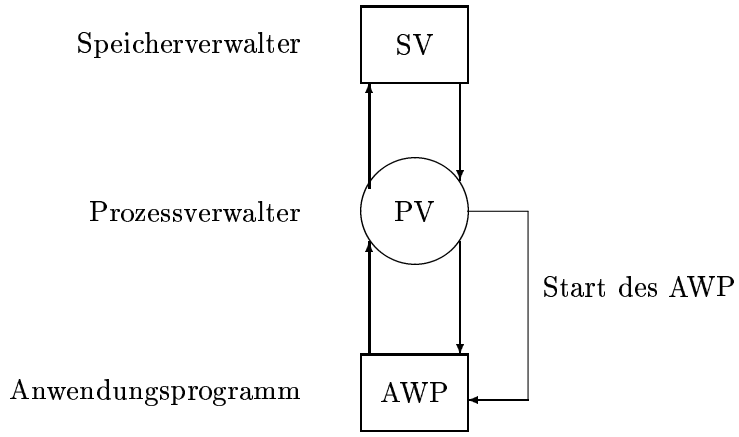


Abbildung 6: "Drehscheibe" Prozessverwalter

Es soll jetzt schrittweise verfeinernd der Speicheraufbau des Segmentes des Anwendungsprogrammes gezeigt werden. Nach dem Laden des Anwendungsprogrammes hat das Aktivierungssegment folgenden Aufbau:

Die Lage und der Aufbau des Aktivierungssegmentes der Anwendung sind willkürlich. Für den Keller der Anwendung stehen in dieser Konfiguration lediglich 16 Plätze zur Verfügung. Die Adresse 15017 wird nicht belegt, weil es sich um eine Anwendung handelt. Später wird gezeigt werden, dass beim Speicherverwalter, einem Programm des Betriebssystems, an dieser logischen Stelle die Adresse des Clienten, also 15022, stehen wird.

Die Startadressen der zeichne-Methoden kann man aus der Tabelle in Abbildung 3 entnehmen. Dabei ist zu beachten, dass es sich um relative Adressen handelt (bzgl. des Codeanfangs).

Der Keller und die Halde besitzen folgenden detaillierten Aufbau nach der ersten Platzreservierung und dem Aufruf des Konstruktors `setSWerte` aus dem Programm `Geomd` von Seite 7.

Die effektive Adresse der HMT der Klasse *SchraffR* ist in diesem Beispiel 10527. Auf Grund der Architektur der PVM wird sie im Objekt `s1` aber mit 504 angegeben, da zu diesem Wert bei den relevanten Paula-Befehlen noch der Epa-Wert 10023 addiert wird.

Durch die zweite Platzreservierung und den entsprechenden Konstruktoraufruf `setRWerte` aus dem prigramm `Geomd` von Seite 7 wird weiterer Haldenplatz belegt (Adressen 14998 bis 15006) und der verfügbare Haldenplatz verringert sich von 190 auf 181 (angefordert waren nur 6 Plätze, aber durch den Platz für drei Kontrollwerte, u.a. der Umfang, werden 9 Plätze effektiv benötigt).

Die Adressen der reservierten Haldenplätze stehen auf den Plätzen der Variablen `r1` und `s1` im Keller des Aktivierungssegmentes.

An dieser Stelle sei bereits auf eine Abweichung dieses Vorgehens von kommerziellen Systemen hingewiesen. Zu jedem auf der Halde angelegten Objekt gehört ein Zeiger

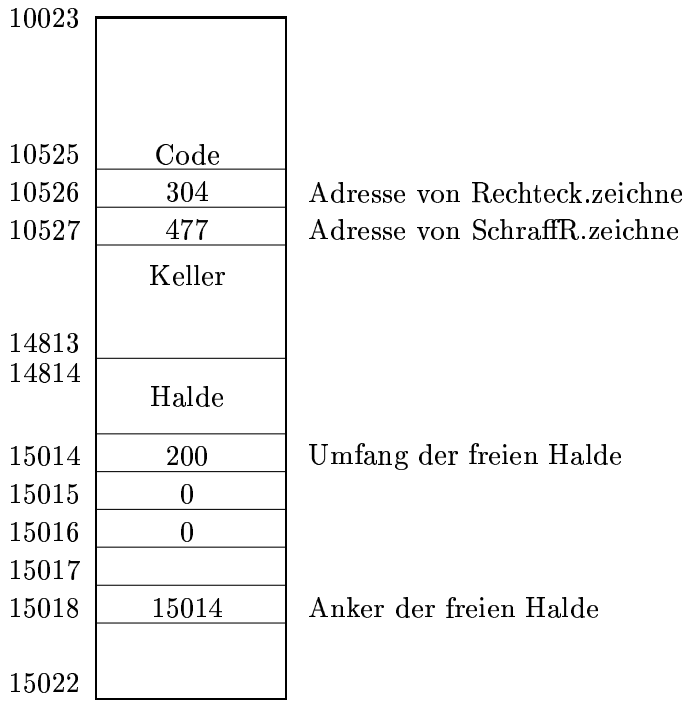


Abbildung 7: Das Aktivierungssegment des APW's beim Start

(Adresse) zu einer Zeile der HMM, und zwar genau zu derjenigen Zeile, die die Adressen der Methoden derjenigen Klasse enthält, von der das Objekt eine Ausprägung ist (siehe Adressen 15010 und 15001). In kommerziellen Systemen, wie Turbo–Pascal, wird diese Adresse nach den Feldern des Basisobjekts abgelegt, hier jedoch vor allen Feldern. Das ist nur möglich gewesen, weil der Speicherverwalter selbst geschrieben wurde und er diesen Platz als Reserve vorgehalten hat. Die Programmierung des Compilers vereinfacht sich hierdurch beträchtlich.

Das Objekt `r1` verweist auf die Adresse 504 seiner HMT. Effektiv verbirgt sich hinter dieser Adresse wieder eine andere, nämlich $504 + 10023 = 10527$, auf der die HMT der Klasse *Rechteck* auch tatsächlich beginnt.

2.2 Polymorphie im Beipiel

Wie kann es nun sein, dass der Aufruf der Methode `verschiebe` (siehe Anweisung Nr.7 im Programm `GeomD`) beim ersten Schleifendurchlauf zwei Zahlenausgaben (11 und 16) produziert und beim zweiten Durchlauf vier (2,5,7 und 5), wird doch äußerlich ein- und dieselbe Funktion aufgerufen? Das liegt daran, dass der Code der Methode `verschiebe` polymorph ist, weil in ihm die homonym-Methode `zeichne` aufgerufen wird. Welche Implementierung von `zeichne` aber verwendet wird, entscheidet letztendlich der Typ des durch die Variable `lauf` referenzierten Objektes.

Zeigt beim ersten Schleifendurchlauf (siehe Anweisung (5) auf Seite 7) die Variable `lauf` auf ein Objekt der Klasse `Rechteck`, wird die Implementierung von `zeichne` aus

10528		r1
10529		lauf
10530	4988(15011)	s1
10531	0	rc
15004	190	Umfang der freien Halde
15005	0	
15006	0	
15007	7	
15008	0	
15009	0	
15010	504(10527)	Adr($HMT_{SchraffR}$)
15011	1	Werte des Objektes
15012	2	
15013	3	
15014	4	
15015	0	
15016	5	
15017		
15018	15004	Anker der freien Halde

Abbildung 8: Keller und Halde nach der ersten Reservierung

der Klasse `Rechteck` verwendet, die eine Zahlenausgabe liefert (siehe Anweisung (1) von Seite 7).

Zeigt die Variable `lauf` beim zweiten Schleifendurchlauf auf ein Objekt der Klasse `SchraffR`, wird die Implementierung von `zeichne` aus der Klasse `SchraffR` verwendet, die zwei Zahlenausgaben liefert (siehe Anweisungen (2) und (3) von Seite 7).

Möglich wird diese Unterscheidung nur dadurch, dass die Objekte (im Speicher) vor ihren eigentlichen Werten noch einen Zeiger auf ihre HMT besitzen. Es ist die Aufgabe des Compilers, den Code von `verschiebe` so allgemein zu übersetzen, dass dies auch funktioniert.

2.2.1 Kelleraufteilung in Methoden

Damit Code polymorph wird, werden vor den Parametern und lokalen Variablen von Methoden noch folgende zusätzliche Daten abgelegt:

10528	4979(15002)	r1
10529		lauf
10530	4988(15011)	s1
10531	0	rc
15004	190	Umfang der freien Halde
15005	0	
15006	0	
15007	7	
15008	0	
15009	0	
15010	504(10527)	Adr($HMT_{SchraffR}$)
15011	1	Werte des Objektes
15012	2	
15013	3	
15014	4	
15015	0	
15016	5	
15017		
15018	14995	Anker der freien Halde

14995	181
14996	0
14997	0
14998	6
14999	0
15000	0
15001	503(10526)
15002	10
15003	11
15004	12
15005	13
15006	4988(15011)

Haldenausschnitt

Abbildung 9: Keller und Halde nach der zweiten Reservierung

Adresse	Wert
b_0	RSA
b_1	Funktionswert
b_2	Absolute Adresse des Aufrufers (this, self, dieses)
b_3	Absolute Adresse(HMT_{K_i}), Aufrufer ist vom Typ K_i
b_4	Hilfsplatz, Adresse der aufzurufenden homonym-Methode
b_5	Absolute Adresse der HMM
b_6	Parameter und lokale Variable

Der Compiler weiß auf Grund des Typs der Variablen `lauf`, dass es sich um eine Referenz auf ein Objekt handelt. Er weiß auch, daß dort die Adresse der HMT des Objektes steht und generiert Anweisungen zum Kopieren dieser Adresse nach b_3 der Methode `verschiebe`. Beim Übersetzen der Methode `verschiebe` hat er überall dort, wo eine homonym-Methode aufgerufen wurde, keinen absoluten Sprung generiert, sondern einen indirekten Sprung `springeAnZielIn` zur basisrelativen Adresse b4(wobei zuvor noch Anweisungen zur Berechnung der richtigen Adresse in b4 generiert wurden).

Die Sprungadresse bei diesem indirekten Sprung wird wie folgt berechnet. Im Keller des Aktivierungssegmentes (Adressen 956 und 957) stehen zur Laufzeit die Startadressen der beiden Implementierungen der homonym-Methode `zeichne` (die HMT besteht jeweils nur aus einer Spalte). Um die richtige Implementierung verwenden zu können, müssen vom Compiler also Anweisungen generiert werden, dass zur Adresse [5] der HMT noch die interne Nummer der homonym-Methode hinzu addiert wird. Diese Nummer ergibt sich einfach durch Hochzählen ab Null für jede homonym-Methode einer Klasse. Die Methode `zeichne` besitzt die interne Nummer Null.

Auf der Adresse b4 steht also während der Abarbeitung von `verschiebe` beim Aufruf von `Rechteck.zeichne` der Wert 418+0 = 418 und beim Aufruf von `SchraffR.zeichne` der Wert 418 + 1 = 419, wodurch die verschiedenen Implementierungen angesprungen werden.

2.3 Homonym–Methoden–Matrix(HMM)

Für jede Klasse mit homonym-Methoden wird eine Zeile[6] in einer Matrix von Methodenstartadressen angelegt, ab b_0 des Hauptprogramms fortlaufend. Zur besseren Veranschaulichung wählen wir hier ein fiktives Besipiel mit drei Klassen K_1, K_2 und K_3, wobei die zweite eine Spezialisierung der ersten und die dritte eine Spezialisierung der zweiten ist. Alle drei Klassen besitzen die homonym-Methoden f_1 und f_2. Dann ergibt sich folgende logische Struktur:

K \ M	M_1	M_2
K_1	Startadresse	Startadresse
K_2	Startadresse	Startadresse
K_3	Startadresse	Startadresse

Abbildung 10: Eine Homonym–Methoden–Matrix bei drei Klassen und zwei homonym-Methoden

Die HMM wird zeilenweise auf dem Keller des AWP's abgelegt:

[5] Eigentlich handelt es sich um eine Startadresse, denn eine Klasse kann durchaus mehrere homonym-Methoden besitzen.

[6] In der Literatur wird diese Zeile auch virtuelle Methodentabelle (kurz VMT) genannt, hier soll sie HMT heißen.

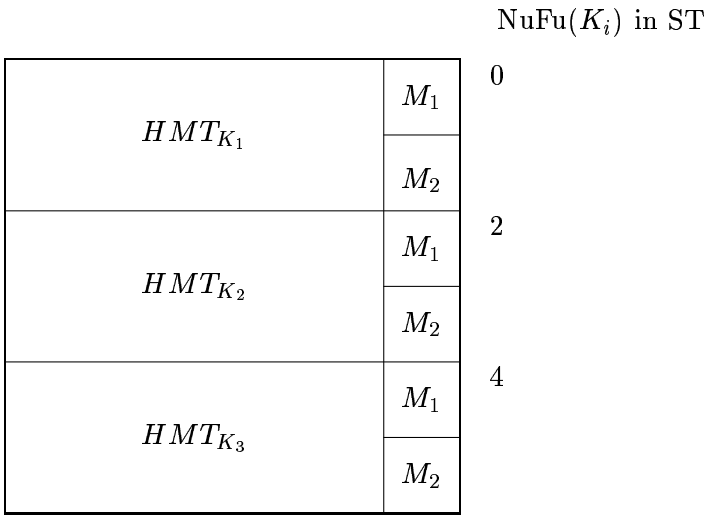


Abbildung 11: Datenstruktur der Homonym–Methoden–Matrix

3 Die Sprache OO–Anton

Anton ist der Name einer von Hans Röck und Christian Steinmann an der Universität Rostock entwickelten rechnerunabhängigen Entwurfssprache, die zur Ausbildung von Studierenden in problemorientierter Algorithmierung im Grundstudium eingesetzt wird. Sie beinhaltet einerseits alle wichtigen Konstrukte, wie Konstante, Variable, Typ, Zuweisung, Funktion, Aktion usw. von problemorientierten Programmiersprachen, ist aber andererseits nicht so formal und mit Details überlastet wie diese. Problemorientierte Programmiersprachen werden unter zwei Hauptaspekten entwickelt:

1. Die Bereitstellung von Konzepten für die Modellierung, also die Abbildung des Problems auf Konzepte der Programmiersprache, und
2. die Möglichkeit einer einfachen und effektiven Compilierung.

Bei Programmiersprachen steht der zweite Aspekt im Vordergrund, bei Entwurfssprachen, wie Anton, der erste Aspekt.

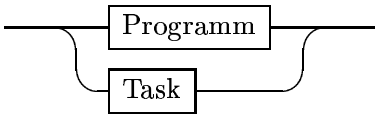
Was Anton jedoch fehlt, sind Konstrukte der objektorientierten Programmierung. Dieses Manko soll mit der vorliegenden Arbeit beseitigt werden, da das Trainieren objektorientierter Konzepte für die spätere Beschäftigung mit Programmiersprachen wie C++ oder Java für Anfänger wichtig ist. Der Name der erweiterten Entwurfssprache ist OO-Anton. Als Nebeneffekt lernen die Studierenden bei der Verwendung von OO-Anton einen zweckmäßigen Programmierstil durch den Zwang zu nutzerdefinierten Datentypen (Klassen) und sauberen Funktionsschnittstellen.

3.1 Die Syntax

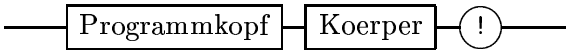
Es wird zwischen Definition und Deklaration von Variablen und Parametern einerseits und Klassen und Typen andererseits unterschieden. Variablen und Parameter werden **definiert**, weil damit die Reservierung von Speicherplatz verbunden ist. Klassen und Typen werden **deklariert**, weil damit keine Reservierung von Speicherplatz verbunden ist. Methoden werden durch die Angabe von Schnittstelle und Implementierung definiert. Gibt man nur ihre Schnittstelle an, so spricht man von Deklaration.

Beide Formen, Deklaration und Definition, führen zu Einträgen in der Symboltabelle.

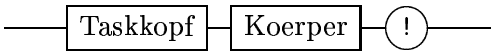
Uebersetzungseinheit



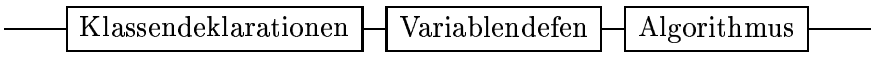
Programm



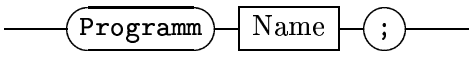
Task



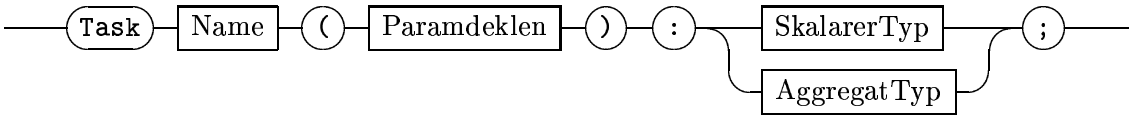
Koerper



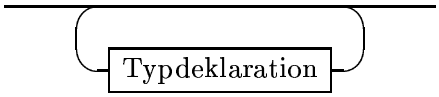
Programmkopf



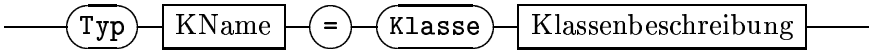
Taskkopf



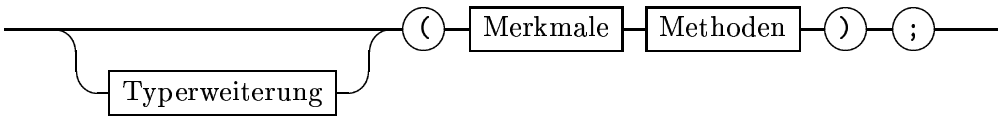
Klassendeklarationen



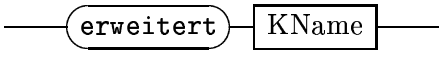
Typdeklaration



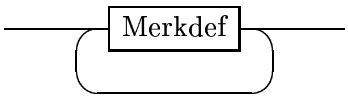
Klassenbeschreibung



Typerweiterung

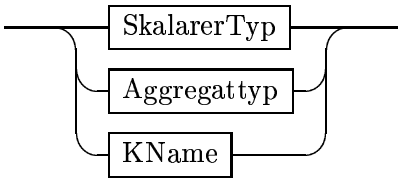


Merkmale

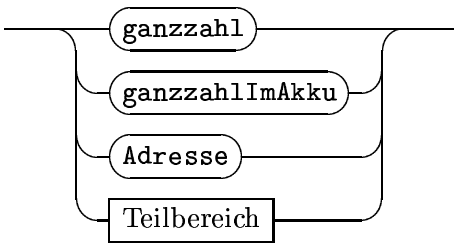


Merkdef

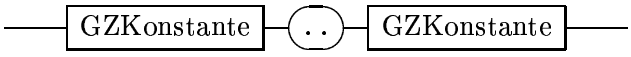
Merkmalstyp



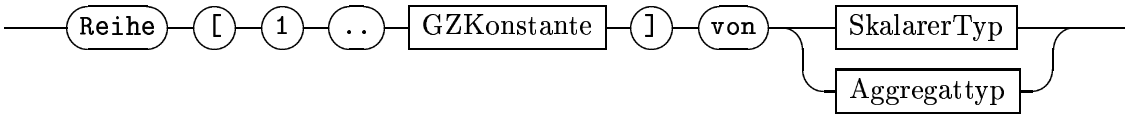
SkalarerTyp



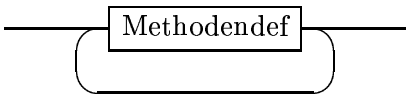
Teilbereich



Aggregattyp

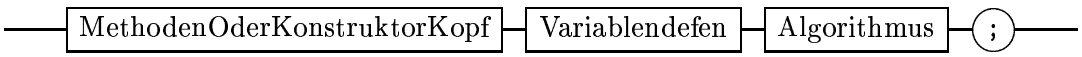


Methoden

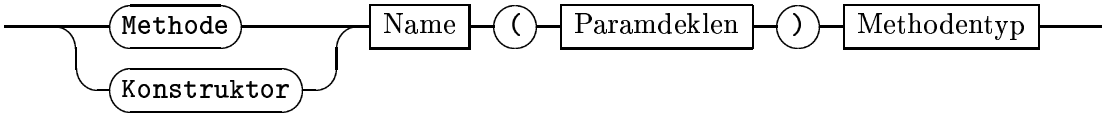


Methodendef

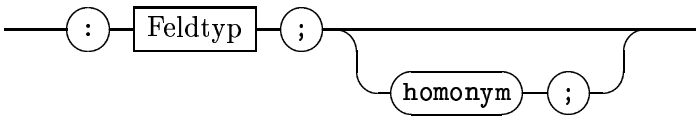
MethodenOderKonstruktorDef



MethodenOderKonstruktorkopf

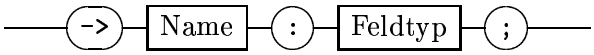


Methodentyp

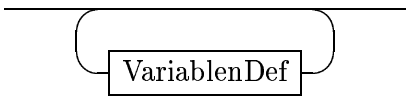


Paramdeklen

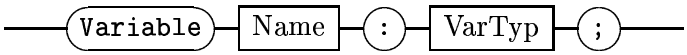
Paramdeklaration



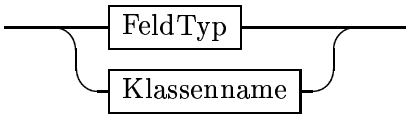
Variablendefen



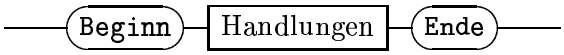
VariablenDef



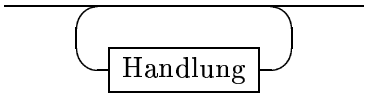
VarTyp



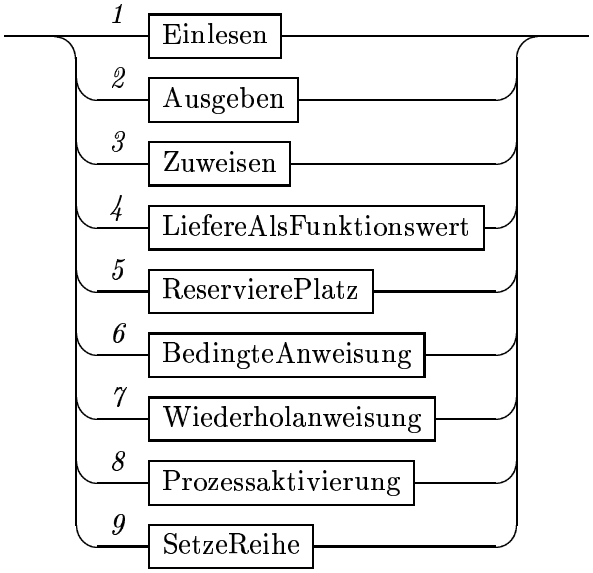
Algorithmus



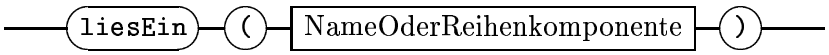
Handlungen



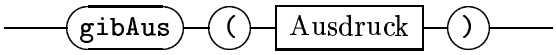
Handlung



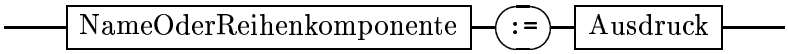
Einlesen



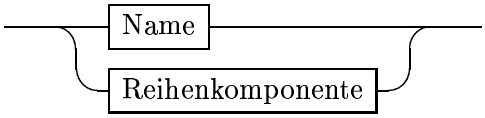
Ausgeben



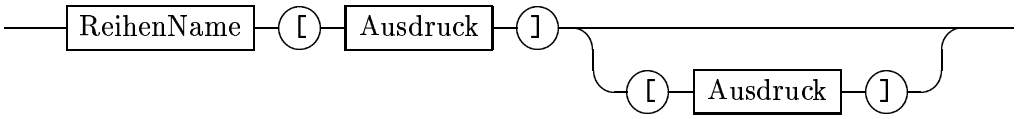
Zuweisen



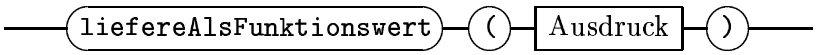
NameOderReihenkomponente



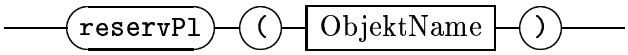
Reihenkomponente



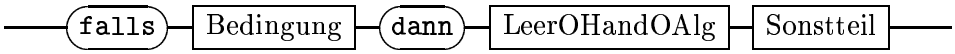
LiefereAlsFunktionswert



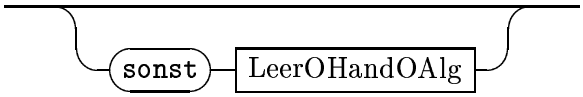
ReservierePlatz



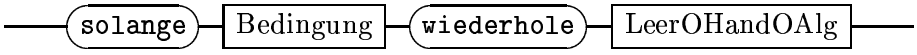
BedingteAnweisung



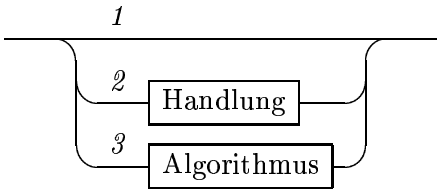
Sonstteil



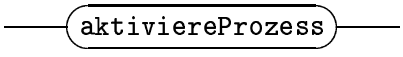
Wiederholungsanweisung



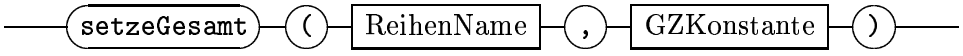
LeerOderHandOderAlg



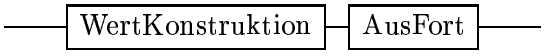
Prozessaktivierung



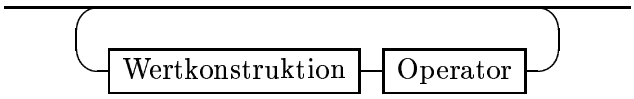
SetzeReihe



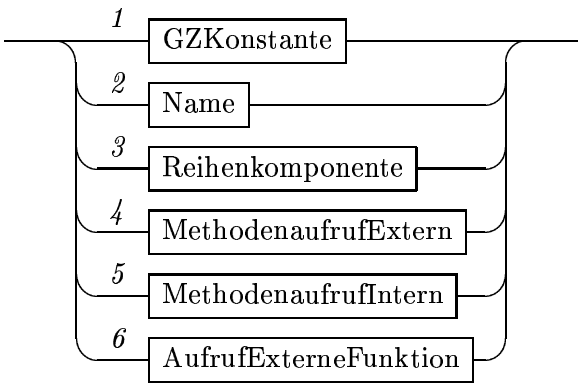
Ausdruck



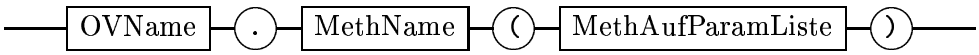
AusFort



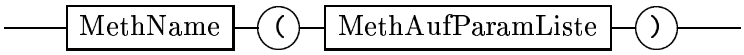
Wertkonstruktion



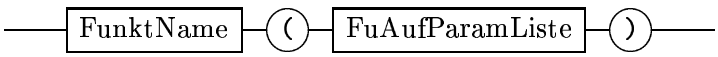
MethodenaufrufExtern



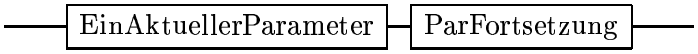
MethodenaufrufIntern



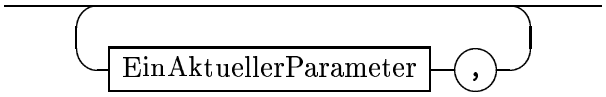
AufrufExterneFunktion



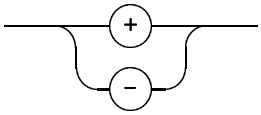
MethAufParamListe



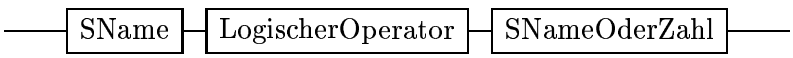
ParFortsetzung



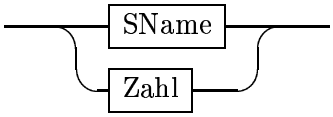
Operator



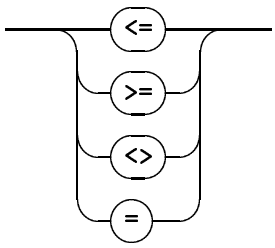
Bedingung



SNameOderZahl



LogischerOperator



3.1.1 Merkwürdigkeiten

In den Syntaxdiagrammen werden die Nichtterminals *Name*, *KName* , *OVName* und *SName* verwendet, obwohl rein äußerlich (für den Scanner) bei Ausprägungen dieser Nichtterminals kein Unterschied festgestellt werden kann. Es sind in allen Fällen Zeichenketten, die mit einem Buchstaben beginnen und danach Buchstaben oder Ziffern aufweisen. Falls dieser Name jedoch eine Variable für Objekte bezeichnen muß, was in der Symboltabelle vom Parser vermerkt wurde, steht im Syntaxdiagramm *OVName*. Die Anweisung `reservPl` z.B. ist nur mit dem Namen einer Variablen für Objekte als Parameter zulässig. Namen für Variablen, die nicht vom Typ `Klasse` oder `Reihe` sind, werden mit *Name* als Oberbegriff beschrieben. Ein Merkmal oder eine Methode kann auch vom Typ `Klasse` sein, wenn als Typname der Name der Klasse verwendet wird. Die obige Syntax wurde unter dem Aspekt ausgewählt, wichtige aber nicht alle Konzepte der Objektorientiertheit zu verwenden. So fehlen z.B. die Mehrfachvererbung und die Zugriffskonzepte wie "privat" oder "öffentlich". In OO–Anton sind alle Methoden öffentlich. Auf explizite Zeiger wurde ebenfalls verzichtet, wobei allerdings eine Variable vom Klassentyp ein Zeiger auf ein Objekt ist. Der direkte Zugriff auf Felder eines Objektes, also ohne die Verwendung von Methoden, ist nur den Methoden des Objektes erlaubt. In Methoden anderer Klassen oder im Hauptprogramm gäbe es beim Versuch, dies zu programmieren, einen Syntax–Fehler. Die Implementierungsdetails bzgl. Datenstrukturen einer Klasse können dadurch versteckt werden.

Es ist also in OO-Anton **unmöglich** (syntaktisch falsch) statt der Anweisung (4c) von Seite 7:

```
rc:= r1.setRWerte(10,11,12,13,s1);
```

zu schreiben:

```
r1.BP[1]      := 10;
r1.BP[2]      := 11;
r1.Groesse[1] := 12;
r1.Groesse[2] := 13;
r1.Next       := s1;
```

Felder einer Klasse (in Java auch Variable genannt), die selbst Objekte sind, haben z.Zt. noch einen eingeschränkten Handlungsspielraum. Sie können lediglich komplett zugewiesen werden aber selbst noch keine Methoden aufrufen. Dazu muss man sie einer Variablen zuweisen.

In OO-Anton gibt es nur zwei Arten von Unterprogrammen: Konstruktoren und Methoden. Konstruktoren sind spezielle Methoden, die später erläutert werden.

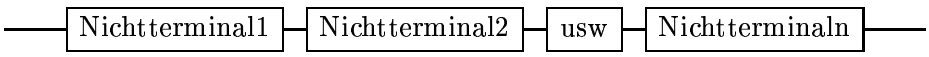
Definition 3.1 *:*
Methoden *sind in OO-Anton Funktionen mit einem impliziten Parameter vom Typ* `Klasse` *und weiteren expliziten Eingabeparametern, die einen skalaren oder reihenwertigen Funktionswert berechnen und Daten des Parameters dauerhaft verändern können.*

3.1.2 Der Parser

Aus den Syntaxdiagrammen kann man nach der Methode des rekursiven Abstiegs schematisch einen Parser ableiten. Es gelten die folgenden Regeln[7]:

R1: Jede Struktur der Form:

Nichtterminal



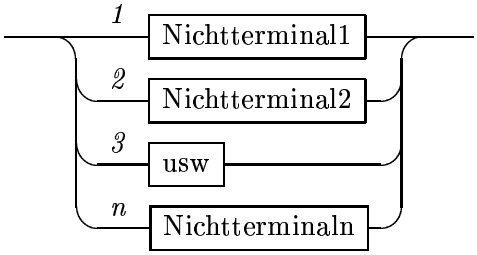
wird in die Anweisungsfolge übersetzt:

```
uebersetzeNT_1(..);uebersetzeNT_2(..);... uebersetzeNT_n(..);
```

Bemerkung: Für diese und die folgenden Regeln gilt: Steht anstelle des Nichtterminals ein Terminal, wird anstelle der Prozedur `uebersetzeNT..` die Prozedur `findeTreffer( )` aufgerufen. Sie überprüft lediglich, ob das erwartete Token mit dem tasächlichen übereinstimmt und bringt eine Fehlerausschrift, falls der Test negativ ausgegangen ist. Im positiven Fall holt sie das nächste Token mit allen Angaben aus der Symboltabelle.

R2: Jede Struktur der Form:

Nichtterminal



wird in die Anweisungsfolge übersetzt:

```
falls t.Token in F1 dann interpretiereNT_1(..) sonst
falls t.Token in F2 dann interpretiereNT_2(..) sonst
        ......
falls t.Token in Fn dann interpretiereNT_n(..) ;
```

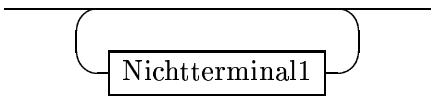
[7] N. Wirth.

(F1=FIRST(NT_1), F2=FIRST(NT_2))

Allerdings müssen wir fordern, dass alle diese First-Mengen paarweise disjunkt sind.

R3: Jede Struktur der Form:

Nichtterminal



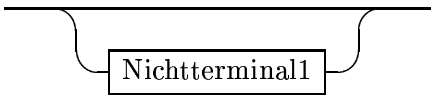
wird in die Anweisungsfolge übersetzt:

```
solange t.Token in F wiederhole
 interpretiereNT(..);
```

(F= FIRST(NT))

R4: Jede Struktur der Form:

Nichtterminal



wird in die Anweisungsfolge übersetzt:

```
falls t.Token in F dann
   interpretiereNT(..);
```

(F= FIRST(NT))

3.2 Die Semantik von OO-Anton

Die Semantik der Anweisungstypen wird anhand der gewählten Codegenerierung erläutert. Eine sehr große Rolle beim Verständnis des Parsers spielt die Symboltabelle. Aus Platzgründen können nur ausgewählte Spalten der Symboltabelle hier vorgezeigt werden. Immer dann, wenn der Compiler äußerlich nicht genügend Informationen aus dem Programmtext entnehmen kann, greift er auf die Symboltabelle zurück, um fehlende Informationen für die Codegenerierung zu ermitteln.

3.3 Datenstrukturen des Parsers

Der Parser wird in OO-Anton notiert.

```
Typ bool   = 0..1;
Typ TParser = Klasse (
       st: SymbolTabelle;
       au: Ausgabe;
      vau: Ausgabe;
       t : SymbolTabEintrag;
   Erfolg: bool;
Methode ueb...(  ) : bool;
   ......................
                   );
```

3.3.1 Die Symboltabelle

Die wichtigste Datenstruktur des Parsers ist die Symboltabelle. Das folgende Beispiel zeigt einen Auschnitt der Symboltabelle, wie sie beim überstezen des Beispielprogrammes `GeomD` von Seite 7 entstehen würde. Jedes Wort (Lexem) des Beispielprogrammes führt zu genau einer Zeile in der Symboltabelle, wobei ein mehrfaches Auftreten dieses Lexems im Quelltext keine erneute Aufnahme in die Symboltabelle nach sich zieht. Die einzelnen Spalten der Symboltabelle werden nachundnach erläutert. Die Leserin oder der Leser wird sich sicher fragen, ob eine Tabellenstruktur nicht zu statisch für einen Parser ist? Man weiß im voraus i.allg. nicht, wie lang z.B. ein Quelltext wird. Die physische Realisierung der Symbotabelle kann auf einer dynamischen Datenstruktur basieren. Zur Veranschaulichung der Wirkungsweise eines Parsers ist die Tabellenform aber sehr anschaulich und geeignet. Sie erinnert an die Systemtabellen relationaler Datenbankverwaltungssysteme.

An dieser Stelle sollen aber schon die Abkürzungen der Spaltennamen erklärt werden.

KlassN: Klassenname, Name der Klasse, in der das Lexem z.B. ein Feld bezeichnet.

ArtE: Art des Eintrags, z.B. ein Klassenname (klassname) oder ein Merkmalsname (merkname) oder der Name einer Methode (metname) oder der Name einer Variablen (varname).

MerNu: Merkmalsnummer, die entsteht, wenn alle Merkmale einer Klasse mit Null beginnend durchnumeriert werden.

MerOff: Verschiebung des Merkmalswertes zum Anfang des Objektwertes.

MetNu: Nummer einer homonym-Methode, die dadurch entsteht, dass alle homonym-Methoden einer Klasse mit Null beginnend durchnumeriert werden. Bei Klassennamen wird hier die Anzahl der homonym-Methoden in sämtlichen Vorgängerklassen festgehalten.

AnzKo: Anzahl Komponenten, i.allg. eins, aber bei Reihen oder reihenwertigen Methoden ein Wert größer als eins.

AnDi: Anzahl Dimensionen, die bei Merkmalen vom Typ `Reihe` z.Zt. eins oder zwei betragen kann. Es sind also in OO-Anton Vektoren oder Matrizen als Merkmale möglich.

DiWe: Dimensionswerte bei ein- oder zweidimensionalen Reihen. Die untere Grenze des Reihenindexwertes ist standardmäßig eins, die obere wird hier eingetragen. Dieses Merkmal ist selbst vom Typ `Reihe[1..2] von ganzzahl`.

TypN: Typname, wobei als vordefinierter Typ lediglich `ganzzahl` vorhanden ist. Der Nutzer kann durch die Definition von Klassen selber Typen kreieren.

TypA: Typart, wie Skalar, Reihe oder Klasse.

Lexem	KlassN	ArtE	MerNu	MerOff	MetNu	AnzKo	AnDi	DiWe	TypN	TypA
Rechteck	Rechteck	klassname	—	—	0	5	—	—	—	—
BPunkt	Rechteck	feldname	0	0	—	2	1	2,0	—	reihe
Groesse	Rechteck	feldname	1	2	—	2	1	2,0	—	reihe
Next	Rechteck	feldname	2	4	—	1	—	—	—	klasse
setRWerte	Rechteck	metname	—	—	-1	1	—	—	ganzzahl	skalar
.....										
holeNext	Rechteck	metname	—	—	-1	1	—	—	—	klasse
zeichne	Rechteck	metname	—	—	0	1	—	—	ganzzahl	skalar
.....										
verschiebe	Rechteck	metname	—	—	-1	1	—	—	ganzzahl	skalar
SchraffR	SchraffR	klassname	—	—	1	6	—	—	—	—
BPunkt	SchraffR	feldname	0	0	—	2	1	2,0	—	reihe
Groesse	SchraffR	feldname	1	2	—	2	1	2,0	—	reihe
Next	SchraffR	feldname	2	4	—	1	—	—	—	klasse
setRWerte	SchraffR	metname	—	—	-1	1	—	—	ganzzahl	skalar
.....										
holeNext	SchraffR	metname	—	—	-1	1	—	—	—	klasse
zeichne	SchraffR	metname	—	—	0	1	—	—	ganzzahl	skalar
.....										
verschiebe	SchraffR	metname	—	—	-1	1	—	—	ganzzahl	skalar
Schraffur	SchraffR	feldname	3	5	—	1	—	—	—	klasse
setSWerte	SchraffR	metname	—	—	-1	1	—	—	ganzzahl	skalar
r1		varname	—	—	—	1	—	—	Rechteck	klasse
lauf		varname	—	—	—	1	—	—	Rechteck	klasse
s1		varname	—	—	—	1	—	—	SchraffR	klasse
rc		varname	—	—	—	1	—	—	ganzzahl	skalar

3.3.2 Konvention zur Namensgebung

Der Parser besteht aus Methoden mit den Namen `ueb...`, die nachfolgend in Gruppen erläutert werden. Anstelle von `...` wird der Name des Nichtterminals verwendet, das für die Methode verantwortlich ist.

3.4 Das Hauptprogramm des Parsers und seine Methoden

```
Variable P1: TParser;
         rc: ganzzahl;
Beginn
  rc     := P1.beginneArbeit;
  rc     := P1.holeNaechstesToken;
  Erfolg := TRUE; {Optimistische Variante}
  rc     := P1.uebersetzeUebersetzungseinheit;
  rc     := P1.beendeArbeit;
Ende!
```

Die Methode `beginneArbeit` initialisiert die Symboltabelle mit den Schlüsselwörtern von OO-Anton und stellt leere Ausgabereihen für den Code der PVM und die Linkerinformationen bereit. `beendeArbeit` erzeugt aus diesen Reihen dann persistente Dateien.

3.5 Merkmale der Klassen

Merkmale einer Klasse werden teilweise wie Variablen übersetzt, denn jedes Merkmal wird unter seinem Namen in der Symboltabelle abgelegt. Aber anders als bei Variablen

erhalten sie keine basisrelativen Adressen zugewiesen. Dafür erhalten sie eine fortlaufende Nummer (Spalte *MerNu*) und einen Offsetwert (Spalte *MerOff*), der die Verschiebung ihrer Werte zum Anfang eines Objektes dieser Klasse angibt. Dieses Vorgehen ähnelt der Verfahrensweise relationaler Datenbankverwaltungssysteme mit den Systemdaten in Systemrealtionen.

3.5.1 Typerweiterung

Klauseln zur Typerweiterung, wie

```
Typ SchraffR = Klasse erweitert Rechteck ....
```

werden dadurch übersetzt, dass jede Zeile der Symboltabelle mit dem Wert `Rechteck` in der Spalte *KlassN* fast identisch ein zweites Mal in die Symboltabelle hineinkopiert wird. Der einzige Unterschied besteht darin, dass an Stelle des Wertes `Rechteck` der Wert `SchraffR` verwendet wird.

3.6 Konstruktoren und Methoden

3.6.1 Konstruktoren

Definition 3.2 *:*
Eine Methode heißt Konstruktor, *wenn sie genau wie eine Methode wirkt, aber zusätzlich noch die Adresse der zum Objekt gehörenden HMT vor das Objekt kopiert.*

Anstelle des Schlüsselwortes `Methode` schreibt man `Konstruktor`.

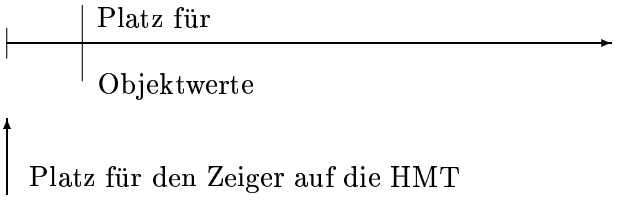


Abbildung 12: Wirkung des Konstruktoraufrufes

Dieses Vorgehen weicht vom Vorgehen kommerzieller Sprachen, wie Turbo-Pascal ab. In Turbo-Pascal wird die Adresse der HMT hinter die Werte der Basisklasse kopiert.

```
Methode uebMethodenOderKonstruktorDef(
        ->  Klassenname: SymbolTabEintrag;
                                    ): bool;
Variable  flag: 0..2;
Beginn
  flag  := uebMethodenOderKonstruktorKopf(Klassenname);
```

```
  uebVariablendefen;
  uebAlgorithmus;

  falls flag  = 1 dann
  Beginn
    hipl:= gibFreienPlatz;
    erzeugeNeuenPaulaBefehl(au,setzeAkku,absolut,-1);
    erzeugeNeuenPaulaBefehl(au,addiere,basisrelativ,2);
    erzeugeNeuenPaulaBefehl(au,merke,basisrelativ,hipl);
    erzeugeNeuenPaulaBefehl(au,lade,basisrelativ,3);
    erzeugeNeuenPaulaBefehl(au,merkeAnZielIn,basisrelativ,hipl);
  Ende;
  erzeugeNeuenPaulaBefehl(au,springeAnZielIn,basisrelativ,0);
  findeTreffer(C_SE);
Ende;
```

Falls es sich um einen Konstruktor handelt (`flag = 1`, wird zur Adressrechnung ein Speicherplatz im Keller des Konstruktors reserviert. Die Adresse des aufrufenden Objektes steht auf b2, die Adresse der zuständigen HMT steht auf b3. Diese Adresse wird vor das Objekt kopiert.

Die letzte Anweisung jeder Methode ist der Rücksprung zum rufenden Code.
Für Terminale wird die Methode `findeTreffer` aufgerufen. Als Parameter wird das Token (die Codierung) des erwarteten Lexems übergeben. Hier wird ein Semikolon erwartet, dessen Token den Namen `C_SE` hat.

3.6.2 Homonyme Methoden

Anhand des Schlüsselwortes `homonym` entscheidet der Compiler, ob es sich um einen Methodennamen handelt, zu dem es mehrere Implementierungen gibt. Bei derartigen Methoden wird bei der Übersetzung der Methodendefinition wie folgt vorgegangen:

1. Genau wie bei skalaren Variablen wird ein Speicherplatz im Keller des AWPs für die Startadresse dieser Methode reserviert: `st.HMTA := st.HMTA + 1;` (basisrelative Adresse). Dadurch wird nacheinander Platz für die HMT reserviert und bei mehreren HMT's entsteht der Platz für die HMM.

2. Jede homonym-Methode erhält eine fortlaufende Nummer innerhalt ihrer Klasse. Die Erste eine Null, die Zweite eine Eins usw. Diese Nummer wird in der Eintragung des Methodennamens in der Symboltabelle notiert: `t1.NuFu := st.FuNu;`.

3. Im Inhaltsverzeichnis der Klasse für den Linker wird diese Methode als homonym-Methode gekennzeichnet.

4. Es werden zwei Anweisungen zum Eintragen der Startadresse der homonym-Methode in den Keller des AWPs generiert. Die basisrelative Adresse des Zielplatzes ergibt sich aus der Addition der Startadresse der Klasse und der Nummer der Methode:
   `HilfAdr := HMTAdresse(st,t1) + t1.NuFu;`

5. Die physische absolute Startadresse der homonym-Funktion kann erst später durch den Linker eingetragen werden. Ihm muß mitgeteilt werden, wo er die

Ersetzung vorzunehmen hat:

```
erzeugeNeueLinkerZeile(vau.Linker,vau.pegel,t1.Lexem,K.Lexem);
```

3.7 Der Zugriff auf Daten

Sowohl in Ausdrücken und Zuweisungen als auch bei Parametern in Methodenaufrufen spielt das Lesen und Schreiben von Daten eine zentrale Rolle bei der Programmierung. Die maschinensprachliche Umsetzung des Datenzugriffs, also des Lesens und Schreibens von Daten, hängt vom Typ der Daten ab. Die folgende Tabelle fasst die wichtigsten Arten und ihre Pendants in der PVM zusammen. Dabei werden folgende Abkürzungen verwendet: rw. für reihenwertig, sk. für skalar.

Datenobjekttyp	Leseopeartion	Schreiboperation
sk.Konstante	setzeAkku	—
sk.Variable	lade	merke
Klassenvariable	lade	merke
sk.Klassenmerkmal	ladeAusZielIn	merkeAnZielIn
rw.Klassenmerkmal	Schleifenkonstrukt	Schleifenkonstrukt
sk.Methodenwert	lade	—
rw.Methodenwert	Schleifenkonstrukt	Schleifenkonstrukt
rw.Variable	Schleifenkonstrukt	Schleifenkonstrukt
Reihenkomponente	ladeAusZielIn	merkeAnZielIn

3.7.1 Adressrechnung

Bei der Adressrechnung werden Informationen aus der Symboltabelle verwendet. Zu jeder skalaren Variablen und Klassenvariablen wird in der Symboltabelle eine basisrelative Adresse verwaltet. Eine Klassenvariable beinhaltet lediglich eine Adresse auf ein Objekt in der Halde des AWPs. Der direkte Zugriff auf Klassenmerkmale darf nur innerhalb der Methoden der Klasse erfolgen. Jeder Methode ist auf b2 die Startadresse des Objektes bekannt. Bei skalaren Klassenmerkmalen ist dann lediglich zu dieser Adresse der Wert der Spalte `MerOff` (Zeile des entsprechenden Merkmals) aus der Symboltabelle hinzuzuaddieren. Handelt es sich beim Klassenmerkmal um eine Reihe, ist beim Zugriff auf eine Komponente dieser Reihe noch der entsprechende Offset der Komponente hinzuzuaddieren. Soll die gesamte Reihe gelesen oder geschrieben werden, muss wieder ein Schleifenkonstrukt generiert werden. Die Anweisung (1) von Seite 7:

```
gibAus(BPunkt[Art]);
```

erfordert danach eine doppelte Adressrechnung. Zuerst muss die Verschiebung des Merkmals `BPunkt` innerhalb seiner Klasse `Rechteck` ermittelt werden (hier gleich Null) und danach die Verschiebung der Komponente mit dem Index `Art`.

Adressrechnung bei Reihenkomponenten

Wenn es sich um eine eindimensionale Reihe (der entsprechende Wert in der Spalte `AnDi` der Symboltabelle ist gleich eins) handelt, ist die Adressrechnung für eine Reihenkomponente sehr einfach. Da in OO-Anton die erste Reihenkomponente den Index eins hat, muß vom Index lediglich eins subtrahiert werden, um die Verschiebung der Reihenkomponente zum Reihenanfang zu ermitteln. Etwas komplizierter wird es bei zweidimensionalen Reihen (der entsprechende Wert in der Spalte `AnDi` der Symboltabelle

ist gleich zwei). Zweidimensionale Reihen werden bei OO-Anton zeilenweise abgelegt. Dann errechnet sich der Offset einer Komponente `...[Zeilenindex][Spaltenindex]` wie folgt:

$$Offset = (Zeilenindex - 1) * DiWe[2] + Spaltenindex - 1 \ .$$

DiWe ist der Name einer Spalte in der Symboltabelle. Hier stehen die oberen Schranken für den Zeilen- und den Spaltenindex.

Mehr als zwei Dimensionen sind in OO-Anton nicht zugelassen.

Initialisierung von Reihen

Sehr häufig kommt es vor, dass Reihen Matrizen beschreiben, deren Elemente alle den gleichen Typ besitzen und initialisiert werden müssen. Ein gebräuchliches Programmiermuster wäre in Anton das folgende (Initialisierung mit Null):

```
Variable  Umsatz : Reihe[1..12] von Reihe[1..100] von ganzzahl
Variable  lauf1:  ganzzahl
Variable  lauf2:  ganzzahl
  lauf1:= 1
  lauf2:= 1
  solange lauf1 <= 12 wiederhole
  Beginn
    solange lauf2 <= 100 wiederhole
    Beginn
      Umsatz[lauf1][lauf2] := 0
      lauf2 := lauf2 + 1
    Ende
    lauf1:= lauf1 + 1
  Ende
```

Wegen der oben beschriebenen Adressrechnung mit Multiplikation für den Zugriff auf die Komponenten `Umsatz[lauf1][lauf2]` ist dies eine teure (im Sinne von viel Rechenzeit bei der Abarbeitung) mögliche Schreibweise. Besonders teuer wird diese Schreibweise auf der PVM, da diese keine Maschinenanweisung zur Multiplikation besitzt. Hier müsste man eine externe Funktion zur Multiplikation hinzubinden. Deshalb gibt es in OO-Anton die Anweisung `setzeGesamt`, mit deren Hilfe alle Komponenten einer Matrix mit einer Konstanten belegt werden können. Die maschinensprachliche Umsetzung von `setzeGesamt` ist einfach, da lediglich in einer Schleife alle Komponenten der Matrix (die Komponentenanzahl ist in der Symboltabelle in der Spalte *AnzKo* hinterlegt) mit der Konstanten beschrieben werden müssen.

Zugriff auf den Funktionswert einer Methode

Die Methode stellt den Funktionswert auf den Platz mit der basisrelativen Adresse b1 bereit. Durch die Basisverschiebung vor dem Sprung zur Methode hat derselbe Platz aus Sicht der rufenden Methode (oder Hauptprogramm) eine andere basisrelative Adresse, die in der rufenden Methode aber bekannt ist.

3.8 Methodenaufrufe und Methodenimplementierungen

3.8.1 Methodenaufrufe

Der Compiler generiert bei der Übersetzung eines Methodenaufrufes Anweisungen, die

1. Kopien der expliziten, aktuellen Parameter im Keller der gerufenen Methode ablegen,
2. die absolute Adresse des rufenden Objektes (versteckter Parameter) in den Keller der gerufenen Methode nach b2 eintragen,
3. der gerufenen Methode in ihren Keller (Adresse b3) die absolute Adresse der zuständigen HMT (eine Zeile in der HMM) eintragen (eigentlich nur für und
4. eine Basisverschiebung und einen Sprung zur ersten Anweisung der gerufenen Methode bewirken.

Welche HMT zuständig ist, erkennt der Compiler am Typ des aufrufenden Objektes. Damit die absolute Adresse der zuständigen HMT ermittelt werden kann, wird die absolute Adresse der HMM benötigt. Die HMM beginnt auf b0 des Hauptprogrammes. Zur Berechnung der absoluten Adresse der HMM wird der Basiswert des Hauptprogrammes benötigt. Im Hauptprogramm kann dieser mittels `addiereBasisabstand` gelesen werden. Innerhalb einer Methode, also bei lokalen Objekt-Variablen, geht das nicht so einfach, da die HMM eine Datenstruktur des Hauptprogrammes ist. Deshalb wird jeder Methode auf b5 die absolute Adresse der HMM als impliziten Parameter übergeben. Die Startadresse einer homonym-Methode wird durch die Addition:

$$StartadresseDerHMM(b5) + OffsetDerKlasse + OffsetDerHomonymMethode$$

und Lesen von dieser Adresse ermittelt. Die letzten beiden Summenden stehen in der Symboltabelle in der Spalte *MetNu* und der Zeile für den Eintrag des Klassennamens bzw. des Namens der homonym-Methode.

4 Linker und Lader

Der Compiler hat aus der Programmdatei `geomd.txt` sechs Dateien erzeugt.

geomd.obj: 226 Maschinenanweisungen des Hauptprogrammes,

rechteck.obj: 211 Maschinenanweisungen für die Methoden der Klasse `Rechteck`,

schraffr.obj: 65 Maschinenanweisungen für die Methoden der Klasse `SchraffR` (die Objektdateien beginnen alle auf adresse Null),

geomd.lin: Informationen über durch das Hauptrogramm gerufene Methoden:

Zeilennummer	Methode	Klasse	Anweisung
-1	geomd		
2	zeichne	Rechteck	setzeAkku
4	zeichne	SchraffR	setzeAkku
70	setSWert	SchraffR	springe
116	setRWert	Rechteck	springe
146	zeichne	Rechteck	springeAnZielIn
165	verschie	Rechteck	springe
180	holeNext	Rechteck	springe

rechteck.lin: Informationen über Methoden, die in Methoden der Klasse `Rechteck` gerufen werden:

Zeilennummer	Methode	Klasse	Anweisung
-1	Rechteck		
116	zeichne	entfaellt	springeAnZielIn
204	zeichne	entfaellt	springeAnZielIn

schraffr.lin: Informationen über Methoden, die in Methoden der Klasse `SchraffR` gerufen werden:

Zeilennummer	Methode	Klasse	Anweisung
-1	SchraffR		
22	setRWert	entfaellt	springe
51	zeichne	Rechteck	springe

rechteck.inv: Informationen über die Startadressen der Methoden der Klasse `Rechteck` in der Datei `rechteck.obj`:

Zeilennummer	Art(codiert)	Methode
0	0	setRWerte
69	0	holeNext
77	1	zeichne
94	0	verschiebe

schraffr.inv: Informationen über die Startadressen der Methoden der Klasse `SchraffR` in der Datei `schraffr.obj`:

Zeilennummer	Art(codiert)	Methode
0	0	setSWerte
39	1	zeichne

(Die Codierung für die Spalte *Art* lautet Eins für eine homonym-Methode und Null sonst.)

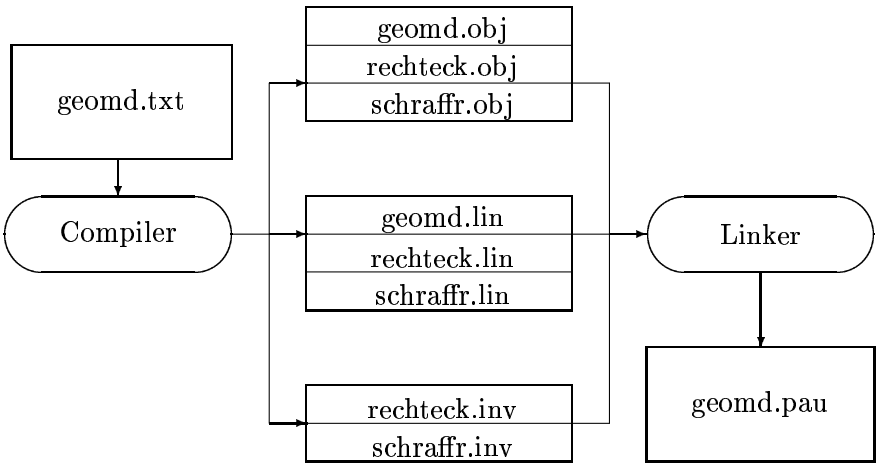


Abbildung 13: Ein- und Ausgabedateien von Compiler und Linker

Der Linker erzeugt aus diesen sechs Dateien die ablauffähige Maschinen-Programm-Datei: `geomd.pau`, die von der PVM interpretiert werden kann. Dazu verkettet er die obj-Dateien zu insgesamt 418 Maschinenanweisungen und aktualisiert die Operanden der in den lin-Dateien registrierten Maschinenanweisungen.

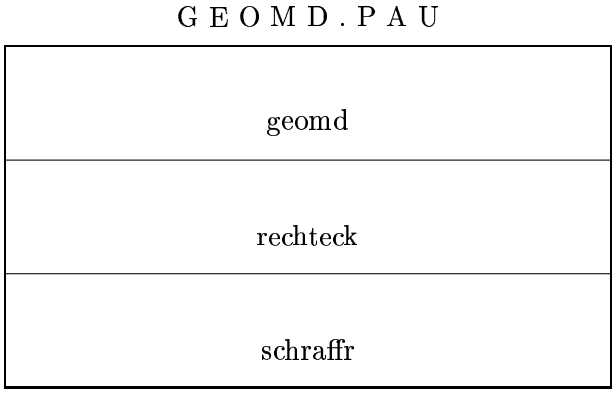


Abbildung 14: Zusammensetzung des Maschinenprogramms geomd.pau

Das Laden des vom Linker erzeugten fast fertigen Maschinenprogrammes könnte von der PVM übernommen werden, was jedoch nicht geschieht. Mittels eines einfachen Texteditors wird das vom Linker erzeugte Programm (genau eine Datei) hinter den Speicherverwalter und Prozeßverwalter geladen und die Systemdaten wie Größe der Halde und Basiswert eingetragen.

5 Zusammenfassung

Das Ziel dieser Arbeit bestand im Entwurf und der Implementierung einer sehr einfachen objektorientierten Programmiersprache OO-ANTON als Erweiterung bzw. Modifikation der bereits seit mehreren Jahren eingesetzten Entwurfssprache ANTON. Dabei sollte besonderes Augenmerk auf die Transparenz der Polymorphie von Code gelegt werden, um allen Studienanfängern der Informatik dieses wichtige Prinzip des objektorientierten Programmier-Paradigmas einfach erklären zu können. Dieses Ziel wurde erreicht.

Dazu mussten zuerst kommerzielle Programmiersprachen nach brauchbaren Konzepten durchsucht werden, darauf aufbauend eigene Konzepte fixiert werden und anschließend der Scanner und Parser, sowie der Programmverbinder geschrieben werden.

Die Erkenntnisse auf dem Weg zu diesem Ziel versetzten aber den Autor in die Lage, ein weiteres Ziel anzusteuern: Die nochmalige Erweiterung von OO-ANTON mit dem Ziel der problemorientierten Darstellung von sogenannten Tasks, den Programmen des Betriebssystems. Auch dieses Ziel wurde erreicht und somit ist OO-ANTON nicht nur eine Sprache für die Anwendungsprogrammierung sondern auch für die Ausbildung in Systemprogrammierung geeignet.

Als Beispiel zur experimentellen Realisierung des erstellten Compilers wurde ein Anwendungsprogramm aus dem Bereich Grafik-Editor gewählt und als Systemprogramm wurde ein sehr einfacher Speicherverwalter entwickelt. Der Speicherverwalter wurde für das Anwendungsprogramm sofort als Server zur Speicherbeschaffung und Speicherfreigabe eingesetzt.

In Auswertung der Schwierigkeit aller durchgeführten Arbeiten muss festgestellt werden, dass das Compilieren von Systemprogrammen die schwierigste Arbeit war, da sie in verschiedenen Adressräumen arbeiten müssen.

Diese Arbeit wäre ohne die Paula-Virtuelle-Maschine von Herrn Prof. Dr. Hans Röck (siehe [1]) und ohne deren Animation von Herrn Dipl. Kaufm. Jörg Zimmermann von der Universität Rostock undenkbar gewesen; dafür beiden meinen herzlichen Dank.

Literatur

[1] Hans Röck, Die Neumann-Maschine, Arbeitsmaterial Nr. 5, Universität Rostock, Wintersemester 1998.

[2] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilerbau, Addison-Wesley 1992, Teil 1 und Teil2, ISBN 3-89319-150 und 3-89319-151.

[3] Niklaus Wirth, Compilerbau, Teubner Studienbücher 1986, ISBN 3-519-32338-9.

[4] Martin Reiser, Niklaus Wirth, Programmieren in Oberon, Addison-Wesley 1995, ISBN r-89319-657-9.

[5] David Flanagan, Java in a nutshell, O'Reilly 1998, ISBN 3-89721-100-9.

[6] Martin Schader, Programmieren in C++, Springer 1997, ISBN 3-540-61834-1.

Autorenangaben

Prof. Dr. Harald Mumm
Fachbereich Wirtschaft
Hochschule Wismar
Philipp-Müller-Straße
Postfach 12 10
D – 23966 Wismar
Telefon: ++49 / (0)3841 / 753 450
Fax: ++49 / (0)3841 / 753 131
E-mail: h.mumm@wi.hs-wismar.de

WDP - Wismarer Diskussionspapiere / Wismar Discussion Papers

Heft 01/2003	Jost W. Kramer: Fortschrittsfähigkeit gefragt: Haben die Kreditgenossenschaften als Genossenschaften eine Zukunft?
Heft 02/2003	Julia Neumann-Szyszka: Einsatzmöglichkeiten der Balanced Scorecard in mittelständischen (Fertigungs-)Unternehmen
Heft 03/2003	Melanie Pippig: Möglichkeiten und Grenzen der Messung von Kundenzufriedenheit in einem Krankenhaus
Heft 04/2003	Jost W. Kramer: Entwicklung und Perspektiven der produktivgenossenschaftlichen Unternehmensform
Heft 05/2003	Jost W. Kramer: Produktivgenossenschaften als Instrument der Arbeitsmarktpolitik. Anmerkungen zum Berliner Förderungskonzept
Heft 06/2003	Herbert Neunteufel/Gottfried Rössel/Uwe Sassenberg: Das Marketingniveau in der Kunststoffbranche Westmecklenburgs
Heft 07/2003	Uwe Lämmel: Data-Mining mittels künstlicher neuronaler Netze
Heft 08/2003	Harald Mumm: Entwurf und Implementierung einer objektorientierten Programmiersprache für die Paula-Virtuelle-Maschine