S E L B S T R E P R O D U K T I O N

B E I

P R O G R A M M E N

Jürgen Kraus

Diplomarbeit

Abteilung Informatik

Universität Dortmund

Februar 1980

Hiermit erkläre ich, daß ich diese Arbeit selbständig und ohne fremde Hilfe verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

J. Kraus

I

Inhalt

Alle in der vorliegenden Diplomarbeit vorkommenden Beispielprogramme wurden auf dem SIEMENS-Rechner, Typ 7738, der Abteilung Informatik an der Universität Dortmund gerechnet.

1.Einleitung

1.1.Motivation

Die Erde, wie sie sich heute präsentiert, ist mit einer Fülle von Lebensformen ausgestattet. In vergangenen Zeitepochen hat es schon Leben gegeben. Zum Teil handelte es sich dabei um die gleichen Formen wie heute, zum Teil um ausgestorbene Arten. Die Biologie hat alle bekannten Lebensformen in ein einheitliches System gestellt. So unterschiedlich die einzelnen Lebensformen auch sind, so entsprechen sie doch einem gemeinsamen Prinzip: Jedes Lebewesen ist aus Zellen aufgebaut. Zellen, die Grundeinheiten des Lebens, sind höchst komplexe biochemische Apparate. Auf Grund der Abstammungslehre muß es irgendwann in der Erdgeschichte eine erste Zelle gegeben haben. Diese Zelle hat sich als Ergebnis der chemischen Evolution auf der Erde herangebildet und wurde zum Ausgangspunkt der biologischen Evolution. Die Frage, wie es zu den ersten Zellen auf der Erde und somit zu den ersten Lebensformen kommen konnte, läßt sich im großen und ganzen mit Hilfe von Experimenten und der Wahrscheinlichkeitsrechnung klären [13] . Das Ergebnis ist, daß Entwicklung und Existenz von Leben praktisch als Konsequenzen der Komplexität der Verhältnisse auf der Früherde anzusehen sind. Sieht man diese Auffassung als richtig an, so ist es durchaus denkbar, daß sich auch in anderen genügend komplexen „Welten" Leben [1)] entwickeln kann oder diese „Welten" zumindest eine Existenzmöglichkeit für bestimmte Formen von Leben bieten.

Die Computertechnik hat in den letzten zwei Jahrzehnten gewaltige Fortschritte gemacht. Die Entwicklung immer

---

1) Auf die Schwierigkeiten, die sich bei der Definition von Leben ergeben, gehen wir ausführlich in Kapitel 7 ein.

neuerer und leistungsfähigerer elektronischer Bauelemente ermöglicht den Bau von digitalen Rechenanlagen, deren Kapazität noch vor wenigen Jahren Utopie gewesen wäre. Durch Zusammenschaltung mehrerer Rechenanlagen bis hin zu überregionalen Rechnernetzen [21] ist es möglich, die Kapazität weiter zu steigern. Dem Benutzer stehen somit Systeme zur Verfügung, die er kaum noch überblicken kann. So wird z.B. die Verwaltung von Rechnernetzen durch Hilfscomputer vorgenommen. Insgesamt gibt es also heute schon Rechenanlagen, die wie ein Universum - bestehend aus Schaltkreisen und Bits - wirken. Die Komplexität solcher Rechenanlagen erinnert durchaus an die Komplexität auf der Früherde. Ist die Vorstellung richtig, daß die Entstehung bzw. die Existenz von Leben eine Folge der Komplexität ist, so wäre die spekulative Idee von Leben auf Computerebene zumindest denkbar. Bei der Vorstellung, wie ein solches Leben aussehen könnte, kann man sich nur am gegenwärtigen biologischen Leben orientieren, da es das einzig bekannte Leben überhaupt ist. Eingangs haben wir die Zelle als Grundelement des biologischen Lebens angeführt. Ohne Kapitel 7 vorgreifen zu wollen, seien hier die Fähigkeit zur identischen Reproduktion auf eigene Veranlassung (Autoreproduktion) und die Möglichkeit zur fehlerhaften Reproduktion (Mutation) als zwei ckarakteristische Eigenschaften lebender Zellen genannt. Im Hinblick auf diese beiden Eigenschaften scheinen sich auf Computerebene selbstreproduzierende Programme als brauchbares Analogon zu lebenden Zellen zu erweisen. Wir werden in 1.2. selbstreproduzierende Programme als Programme definieren, die in der Lage sind, ihren eigenen Programmtext während ihrer Laufzeit auszugeben, ohne daß ihnen dazu der „Bauplan" ihres Textes von außerhalb mitgeteilt werden muß. Da elektronische Rechenanlagen nicht hundertprozentig fehlerfrei arbeiten, ist die Möglichkeit einer fehlerhaften Ausgabe des Programmtextes, also einer Mutation, automatisch immer vorhanden. Selbstreproduzierende Programme kämen also als Träger von Leben auf Computerebene durchaus in Frage.

Hauptaufgabe dieser Arbeit ist es nicht nur, die Existenz

selbstreproduzierender Programme zu beweisen (Kapitel 2), sondern konkrete Beispiele für selbstreproduzierende Programme in verschiedenen Programmiersprachen anzugeben (Kapitel 3 und 6) und deren Eigenschaften zu diskutieren (Kapitel 4 und 5). Reproduktion und Mutation sind Eigenschaften, die zur Evolution befähigen. Evolution tritt ein, wenn Selektion als zusätzliche Komponente mitwirkt. Evolution ist die Ursache für die unerhörte Artenfülle irdischen Lebens und müßte auch bei selbstreproduzierenden Programmen zu immer neuen Programmen mit verschiedensten Eigenschaften führen. Einige Modelle zur Evolution selbstreproduzierender Programme werden in den Kapiteln 8 und 9 erörtert. Zuvor wird jedoch in Kapitel 7 die Frage untersucht werden, inwieweit sich selbstreproduzierende Programme in der „Umwelt" Rechner wirklich mit lebenden Zellen vergleichen lassen.

1.2.Definition selbstreproduzierender Programme

Die vorliegende Arbeit handelt in erster Linie von Programmen und den Programmiersprachen, denen diese angehören. Um präzise zu sein, müßte daher an dieser Stelle definiert werden, was unter einer Programmiersprache zu verstehen ist, wie Programme in einer jeweiligen konkreten Programmiersprache aufgebaut sind (Syntax) und wie ein konkretes Programm auf einer konkreten Rechenmaschine zu interpretieren ist (Semantik) (vgl. etwa [9] [14]). Derartige Definitionen würden sicher den Rahmen dieser Arbeit sprengen. In Kapitel 2 werden sie jedoch wenigstens ansatzweise für die abstrakte Programmiersprache PL durchgeführt. Bei konkreten Programmiersprachen wird bzgl. der Syntax auf die jeweiligen Arbeiten verwiesen, in denen die Syntax beschrieben ist. Im Hinblick auf die Semantik wird sich der Begriff der von einem Programm realisierten Funktion als ausreichend erweisen (vgl.(5.1.1)). Im übrigen setzt die Arbeit voraus, daß der Leser mit dem in der Informatik üblichen Sprachgebrauch bzgl. Programmen und Programmiersprachen vertraut ist.

Konkrete Programmiersprachen lassen sich allgemein in höhere Programmiersprachen und in Assembler-Sprachen differenzieren. Im Hinblick auf Selbstreproduktion sind folgende Charakteristika wesentlich.

Assembler-Sprachen

> sind direkte Produkte der jeweiligen Maschinenstruktur und lassen sich deshalb als maschinenorientiert bezeichnen. Viele Kennzeichen einer konkreten Rechenanlage lassen sich an der zugehörigen Assembler-Sprache ablesen, unter anderem auch die Struktur des Aroeitsspeichers, auf den Assembler-Programme zugreifen können. Ausführbare Assembler-Programme befinden sich in der Form ihres Maschinenkodes im Arbeitsspeicher. Während der Laufzeit können Assembler-Programme also auf ihren eigenen Maschinenkode zugreifen und ihn auch verarbeiten.

Höhere Programmiersprachen

> sind Programmiersprachen, die die physikalische Struktur des Rechners unberücksichtigt lassen und somit auch nicht an eine feste Rechenanlage gebunden sind. Auf der Ebene von höheren Programmiersprachen gibt es daher auch in der Regel keine Zugriffsmöglichkeit auf den Arbeitsspeicher des jeweiligen Rechners. Programme in höheren Programmiersprachen haben also nicht die Möglichkeit, ihren eigenen Maschinenkode zu lesen und zu verarbeiten.

Programme werden im allgemeinen mit den von ihnen berechneten Funktionen identifiziert. Für die vorliegende Arbeit ist jedoch ein anderer Aspekt von Programmen ebenso wichtig:

> Programme sind endliche Zeichenketten, also Texte.

Im Verlauf seiner Verarbeitung durch den Rechner liegt ein und dasselbe Programm als unterschiedlicher Text über

verschiedenen Alphabeten vor. Ein Programm in Assembler-Sprache unterscheidet sich textuell von seiner Übersetzung in Maschinenkode. Während Assembler-Programme zunächst alphanumerische Texte darstellen, sind Programme in Maschinenkode nur aus den 16 Hexadezimalziffern Ø bis F aufgebaut. Ähnlich liegen die Verhältnisse bei höheren Programmiersprachen. Zwischen dem Quellprogramm in höherer Programmiersprache und dem Objektprogramm im Maschinenkode liegen unter Umständen jedoch noch ein oder mehrere Übergangsformen in irgendwelchen Zwischenkodes.

Gemäß den unterschiedlichen Charakterisierungen für höhere Programmiersprachen und Assembler-Sprachen weisen die Definitionen für selbstreproduzierende Programme in diesen beiden Sprachebenen Unterschiede auf.

Sei zunächst S eine höhere Programmiersprache im üblichen Sinn.

(1.2.1) Definition: Sei π ein (syntaktisch korrektes) Programm aus S.

(i) Weist π keine Eingabe auf, so heißt π (streng) selbstreproduzierend, falls π (genau) seinen Programmtext in S ausgibt.

(ii) Weist π Eingabe auf, so heißt π (streng) selbstreproduzierend, falls π bei jeder zulässigen Eingabe (genau) seinen Programmtext in S ausgibt.

Definition (1.2.1) schließt also selbstreproduzierende Programme mit Eingabe nicht grundsätzlich aus, verhindert jedoch, daß der Eingabe Informationen entnommen werden, die zur Selbstreproduktion benötigt werden; da die Selbstreproduktion bei jeder Eingabe erfolgt, erfolgt sie unabhängig von der Eingabe.

Sei nun M eine zu einer konkreten Rechenanlage gehörige

Assembler-Sprache. Die Definition für selbstreproduzierende Assembler-Programme spiegelt die Tatsache wider, daß Assembler-Programme ihren eigenen Maschinenkode lesen können.

(1.2.2) Definition: Sei π ein gültiges Programm aus der Assembler-Sprache M.

(i) Weist π keine Eingabe auf, so heißt π (streng) selbstreproduzierend, falls π (genau) seinen Maschinenkode ausgibt oder innerhalb des Arbeitsspeichers kopiert.

(ii) Weist π Eingabe auf, so heißt π (streng) selbstreproduzierend, falls π bei jeder Eingabe (genau) seinen Maschinenkode ausgibt oder innerhalb des Arbeitsspeichers kopiert.

Im Gegensatz zu höheren Programmiersprachen brauchen die Kopien selbstreproduzierender Assembler-Programme vor der Ausführung nicht in Maschinenkode übersetzt zu werden. Abb. 1.2.A zeigt die Unterschiede, die sich im Hinblick auf Selbstreproduktion zwischen höheren Programmiersprachen und Assembler-Sprachen ergeben, im Zusammenhang.

Aus der Tatsache, daß Assembler-Programme ihren eigenen Maschinenkode im Arbeitsspeicher lesen können,läßt sich bei einiger Kenntnis von Assembler-Sprachen leicht die Existenz selbstreproduzierender Assembler-Programme folgern. Auch die Angabe von Beispielen für selbstreproduzierende Assembler-Programme fällt nicht schwer (Abschnitt 3.4.). Anders liegen jedoch die Verhältnisse bei höheren Programmiersprachen. Hier ist die Existenz selbstreproduzierender Programme durchaus nicht intuitiv klar und muß daher in Kapitel 2 auf theoretischem Weg nachgewiesen werden. Auch die Angabe von realisierbaren Beispielen ist bedeutend schwieriger als bei Assembler-Sprachen. Ganz allgemein liegen im Hinblick auf Selbstreproduktion die Verhältnisse bei Assembler-Sprachen einfacher als bei höheren Programmiersprachen. Aus diesem Grund beschäftigen sich die Kapitel 3, 4 und 5 fast ausschließlich mit der Selbstreproduktion bei höheren Programmiersprachen.

Höhere Programmiersprachen	Assembler-Sprachen
Programme können Arbeitsspeicher nicht lesen	Programme können Arbeitsspeicher und damit ihren eigenen Maschinenkode lesen
⇓	⇓
Selbstreproduktion ≙ Erzeugung des Quellprogramms in der höheren Programmiersprache	Selbstreproduktion ≙ Erzeugung direkter Kopie des Maschinenkodes
⇓	⇓
Übersetzung der Kopie erforderlich	Keine Übersetzung der Kopie erforderlich

Abb. 1.2.A

2.Existenz selbstreproduzierender Programme

2.1.Einleitung

In diesem Kapitel soll auf theoretischem Weg die Existenz selbstreproduzierender Programme in höheren Programmiersprachen nachgewiesen werden. Wir werden dabei nicht mit Hilfe von realen Programmiersprachen (PASCAL,SIMULA,ALGOL etc.) und deren vielen Eigenarten argumentieren. Statt dessen definieren wir, soweit das in diesem Rahmen möglich ist, eine eigene einfache Programmiersprache, die sich durch besonders einfache Datentypen und allgemein in höheren Programmiersprachen benutzte Konstruktionsprinzipien auszeichnet. Diese Programmiersprache wird PL heißen. Trotz ihrer Einfachheit wird PL die gleiche „Berechenkapazität" haben wie alle gängigen Programmiersprachen. PL wird sich als geeigneter Einstieg in die Theorie der „berechenbaren Funktionen" erweisen. Diese Theorie werden wir nur soweit verfolgen, wie es zum Nachweis selbstreproduzierender Programme in PL notwendig ist.

Da alle gängigen Programmiersprachen die gleiche „Berechenkapazität" wie PL haben, läßt sich aus der Existenz selbstreproduzierender Programme in PL die Existenz selbstreproduzierender Programme in realen Programmiersprachen - sowohl in höheren Programmiersprachen als auch in Assembler-Sprachen - folgern.

2.2.Definition einer einfachen Programmiersprache PL(A)

Grundlage ist zunächst ein beliebiges, aber festes, endliches Alphabet $A=\{a_1,\ldots,a_n\}$, $n\in\mathbb{N}$. Die Menge A^* aller endlichen Wörter über A stellt die Menge der Daten dar, auf denen Programme in PL arbeiten. Das leere Wort $\varepsilon \in A^*$ ist dabei als Datum zugelassen.

(2.2.1) Definition (Ausdrücke):

(i) Die Konstanten in PL sind die Elemente von A^*.

(ii) Die Variablen $X_1, X_2, \ldots, Y, Z, W$ sind Elemente aus einer festen Menge VR. Jede Variable kann Werte aus A^* annehmen.

(iii) Operationen sind Xa und $\wp(X)$ für jedes $X \in VR$, $a \in A$.

Bedeutung:

Xa hat den Wert xa, falls $x \in A^*$ der Wert von X ist.
$\wp(X)$ hat den Wert $x \in A^*$, falls xa der Wert von X ist für ein Element $a \in A$. Andernfalls hat $\wp(X)$ den Wert ε.

(iv) Bedingungen haben die Form $\omega(X)=a$ oder $X=\varepsilon$ mit $X \in VR$ und $a \in A$.

Bedeutung:

$\omega(X)=a$ ist genau dann wahr, wenn a der letzte Buchstabe des Wertes der Variablen X ist. $X=\varepsilon$ ist genau dann wahr, wenn ε der Wert von X ist.

(2.2.2) Definition (Grundanweisungen):

Die Grundanweisungen in PL sind:
Die leere Anweisung $\gamma_1 : \overline{\varepsilon}$
und die Wertzuweisungen $\gamma_2 : X:=\varepsilon$
$\gamma_3 : X:=Xa$
$\gamma_4 : X:=Y$
$\gamma_5 : X:=\wp(X)$,
für alle Variablen X und Y aus VR und $a \in A$.

(2.2.3) Definition (Kontrollstrukturen):

Die Kontrollstrukturen in PL sind:

$\alpha_1 : \quad P;Q$

Bedeutung:
Hintereinanderausführung von Anweisungen.
Vergleiche übliche Programmiersprachen.

$\varkappa_2$: if p then goto L

Bedeutung:

$\varkappa_2$ stellt einen bedingten Sprung dar. p steht für eine Bedingung (vgl. (2.2.1)(iv)).
L ist eine Marke (vgl. (2.2.4)).
Ansonsten wie in üblichen Programmiersprachen.

$\varkappa_3$: if p then P else Q fi

Bedeutung:

Verzweigung; p ist eine Bedingung. Die Anweisungen P und Q stellen die Alternativen dar.
Vergleiche übliche Programmiersprachen.

$\varkappa_4$: while X=ε do P od

Bedeutung:

while-Schleife; Bedingungen der Form $\omega(X)=a$, $X \in VR$, $a \in A$, sind nicht erlaubt. P ist eine Anweisung. Ansonsten wie in üblichen Programmiersprachen.

$\varkappa_5$: loop X case $a_1 \longrightarrow P_1$, ⋮ ⋮, $a_n \longrightarrow P_n$, end

Bedeutung:

$\varkappa_5$ stellt eine loop-Schleife mit Fallunterscheidung dar. Bei der Auswertung wird zunächst eine interne Kopie der Variablen X angelegt. Danach wird der Wert von X von links nach rechts durchlaufen. Für jeden Buchstaben (d.h. Element aus A) a_j des Wertes von X wird die zugehörige Anweisung P_j ausgeführt. Fehlt für einen Buchstaben a_j die Vorschrift $a_j \longrightarrow P_j$ in der Liste der Alternativen, so wird so verfahren, als würde in der Liste $a_j \longrightarrow \overline{\varepsilon}$ stehen, $j \in [n]$.

(2.2.4) Definition (Marken):

Marken sind Elemente aus einer festen Menge

$M = \{L_1, L_2, \ldots\}$. Eine Marke kann in der Form
L : P vor jeder Anweisung P stehen.

(2.2.5) Definition (Anweisung):

Eine Anweisung in PL ist entweder eine Grundanweisung, oder sie besteht aus Grundanweisungen, die mittels der Kontrollstrukturen $\varkappa_1$ bis $\varkappa_5$ miteinander verknüpft sind.

(2.2.6) Definition (PL-Programme):

Ein Programm π in PL hat die Form

π = input $X_1, \ldots, X_r$;
AW_π;
output $Z_1, \ldots, Z_s$ $\qquad r \geq 0, s \geq 0$

wobei AW_π eine Anweisung ist.
Die paarweise verschiedenen Variablen $X_1, \ldots, X_r \in VR$ heißen Eingabevariable.
Die paarweise verschiedenen Variablen $Z_1, \ldots, Z_s \in VR$ heißen Ausgabevariable.

Tritt in AW_π die Kontrollstruktur $\varkappa_2$: if p then goto L auf, so darf L in AW_π nur einmal in der Form L : P auftreten, wobei P eine Anweisung ist.

(2.2.7) Definition (Ausführung von PL-Programmen):

Die Ausführung eines Programms π in PL beginnt damit, daß die Eingabevariablen $X_1, \ldots, X_r$ mit Eingabewerten belegt werden. Alle anderen in π vorkommenden Variablen werden mit ϵ initialisiert. Danach wird AW_π ausgeführt. Nach Ausführung von AW_π liegt das Ergebnis der Programmausführung in Form der Werte der Ausgabevariablen $Z_1, \ldots, Z_s$ vor.
Hält das Programm nie an, so ist das Ergebnis von π undefiniert.

(2.2.8) Bezeichnung: Prinzipiell haben wir hier nicht genau

eine Programmiersprache PL definiert, sondern eine Klasse von Programmiersprachen. Das liegt daran, daß wir noch Freiheit haben in der Wahl der Mengen VR und L und insbesondere in der Wahl des Alphabets **A.** Während die Elemente von VR und L nur programminterne Bezeichnungen darstellen, bestimmt das Alphabet A die Datenmenge, auf der PL-Programme arbeiten. Je nachdem, welches endliche Alphabet A wir zugrunde legen, werden wir in Zukunft die in den Definitionen (2.2.1) bis (2.2.7) definierte Programmiersprache mit PL(A) bezeichnen.

(2.2.9) Bemerkung: Streng genommen haben wir keine formale Definition von PL(A) vorgenommen. Fehlinterpretationen sind denkbar. Unsere Definition wäre exakt, hätten wir sowohl die Syntax als auch die Semantik von PL(A) mit formalen Methoden beschrieben. Besonders die Beschreibung der Semantik ist sehr mühsam und würde den gegebenen Rahmen sprengen. Es soll aber wenigstens die Syntax von PL(A) in Form einer kontextfreien Grammatik angegeben werden.

2.3.Eine kontextfreie Grammatik für PL(A)

Die folgende kontextfreie erzeugende Grammatik
$G(A) = (V_T, V_N, s_0, P)$ erzeugt alle gültigen PL(A)-Programme zu gegebenem Alphabet A. Leider werden nicht genau die gültigen PL-Programme erzeugt, sondern auch Programme, die sich nicht durchführen lassen. Es sei hier auf Definition (2.2.7) verwiesen. Dort finden sich einige umgangssprachliche Regeln, wie z.B. „eine Marke L darf nur einmal in der Form L : P in AW_{π} auftreten". Derartige Regeln lassen sich nicht mittels einer kontextfreien Grammatik erfassen. Wir benötigen diese Regeln, um unter den von G(A) erzeugten Programmen die gültigen Programme von den nicht durchführbaren zu unterscheiden. Der gleiche Effekt tritt bei der Beschreibung realer Programmiersprachen durch kontextfreie Grammatiken auf. Auch hier kommt man i.a. nicht ohne umgangssprachliche Regeln aus.

Beispiel (SIMULA): „Sprünge in das Innere von while-Schleifen sind verboten". [17] [7]

(2.3.1)Angabe der Grammatik $G(A)=(V_T,V_N,s_o,P)$:

Die Menge der terminalen Zeichen V_T ist:

$V_T = A \cup M \cup VR \cup \{$ input , output , if , then , goto , else , fi , while , do , od , loop , case , end , : , = , $\longrightarrow$, ; , , , $\sqcup$, (,) , ς , ω , ϵ , $\bar{\epsilon}\}$

Grundsymbole

Die Menge der nichtterminalen Zeichen V_N ist:

$V_N = \{$ ⟨program⟩, ⟨statement⟩, ⟨simple statement⟩, ⟨identifier⟩, ⟨label⟩, ⟨identifier list⟩, ⟨condition⟩ $\}$

Das Startzeichen s_o ist ⟨program⟩.

Die Menge P umfaßt die Produktionen:

1. ⟨program⟩ $\longrightarrow$ input ⟨identifier list⟩ ; ⟨statement⟩ ; output ⟨identifier list⟩
2. ⟨identifier list⟩ $\longrightarrow$ ⟨identifier list⟩ , ⟨identifier⟩
3. ⟨identifier list⟩ $\longrightarrow$ ⟨identifier⟩
4. ⟨identifier⟩ $\longrightarrow$ X, für alle $X \in VR$
5. ⟨identifier⟩ $\longrightarrow \epsilon$
6. ⟨statement⟩ $\longrightarrow$ ⟨label⟩ : ⟨statement⟩
7. ⟨statement⟩ $\longrightarrow$ ⟨statement⟩ ; ⟨statement⟩
8. ⟨statement⟩ $\longrightarrow$ if ⟨condition⟩ then goto ⟨label⟩
9. ⟨statement⟩ $\longrightarrow$ if ⟨condition⟩ then ⟨statement⟩ else ⟨statement⟩ fi
10. ⟨statement⟩ $\longrightarrow$ while ⟨identifier⟩ $= \epsilon$ do ⟨statement⟩ od
11. ⟨statement⟩ $\longrightarrow$ loop ⟨identifier⟩ case
    $a_1 \longrightarrow$ ⟨statement⟩ ,
    ⋮
    $a_n \longrightarrow$ ⟨statement⟩ , end

12. ⟨statement⟩ ⟶ ⟨simple statement⟩
13. ⟨label⟩ ⟶ L, für alle $L \in M$
14. ⟨condition⟩ ⟶ $\omega(X)=a$ für alle $a \in A$, $X \in VR$
15. ⟨condition⟩ ⟶ $X=\varepsilon$ für alle $X \in VR$
16. ⟨simple statement⟩ ⟶ $\overline{\varepsilon}$
17. ⟨simple statement⟩ ⟶ $X:=\varepsilon$ für alle $X \in VR$
18. ⟨simple statement⟩ ⟶ $X:=Xa$, für alle $X \in VR, a \in A$
19. ⟨simple statement⟩ ⟶ $X:=X'$, für alle $X, X' \in VR$
20. ⟨simple statement⟩ ⟶ $X:=\S(X)$, für alle $X \in VR$

Wir können folgende Entsprechungen feststellen:

Regel 1-5 ≙ Definition (2.2.6)
Regel 6 ≙ Definition (2.2.4)
Regel 7-13 ≙ Definition (2.2.3),(2.2.5)
Regel 14-20 ≙ Definition (2.2.1),(2.2.2)

Die oben erwähnten umgangssprachlichen Regeln in den Definitionen bleiben von G(A) unberücksichtigt.

2.4.PL(A)-berechenbare Funktionen, Church'sche These

Sei nun A endliches Alphabet, $\pi \in PL(A)$. π besitzt $r \geqslant 0$ Eingabe- und $s \geqslant 0$ Ausgabevariable. Während der Programmausführung wird aus der Belegung der Eingabevariablen eine Belegung der Ausgabevariablen ermittelt, falls das Programm anhält. Hält das Programm an, was i.a. nicht vorausgesetzt werden kann, so steilt die letzte Belegung der Ausgabevariablen das Ergebnis der Programmausführung dar. Hält das Programm nicht, so ist das Ergebnis undefiniert. In beiden Fällen interessieren etwaige Zwischenbelegungen irgendwelcher Variablen während der Programmausführung nicht. Dieser Sichtweise entspricht Definition (2.4.1).

(2.4.1) Definition: Sei $\pi \in PL(A)$. Die von π berechnete Funktion ist $\varphi_\pi : (A^*)^r \longrightarrow (A^*)^s$, $r,s \geqslant 0$.
φ_π ordnet jeder Anfangsbelegung $(x_1,\ldots,x_r)$, $x_i \in A^*$, $i \in [r]$[1], der Eingabevariablen ein Ergebnis $(z_1,\ldots,z_s) = \varphi_\pi(x_1,\ldots,x_r)$, $z_j \in A^*$, $j \in [s]$, zu,

1) Notation: $[r] := \{1,\ldots,r\}$ für jedes $r \in \mathbb{N}$, nicht zu verwechseln mit Literaturverweisen.

falls das Programm π anhält. Hält π nicht an, so ist $\varphi_\pi(x_1,\ldots,x_r)$ undefiniert.

(2.4.2) Bemerkung: Aus Definition (2.4.1) folgt:

I. φ_π ist i.a. eine partielle Funktion.

II. Die Sonderfälle r=o und s=o sind ausdrücklich zugelassen. Die Bedeutung dieser Sonderfälle sei hier jedoch kurz erläutert. Es bezeichne () das Nulltupel:

(i) $\varphi_\pi: (A^*)^r \longrightarrow (A^*)^o$, $r \geq 1$, ordnet jedem r-Tupel $(x_1,\ldots,x_r) \in (A^*)^r$ das Nulltupel () zu, falls π mit $(x_1,\ldots,x_r)$ als Eingabebelegung anhält.

$$\varphi_\pi(x_1,\ldots,x_r) = \begin{cases} () \text{ , falls } \pi \text{ hält} \\ \text{undefiniert sonst} \end{cases}$$

(ii) $\varphi_\pi: (A^*)^o \longrightarrow (A^*)^s$, $s \geq 1$, ordnet dem Nulltupel () ein s-Tupel $(z_1,\ldots,z_s) \in (A^*)^s$ zu, falls π anhält.

$$\varphi_\pi() = \begin{cases} (z_1,\ldots,z_s) \in (A^*)^s, \text{ falls } \pi \text{ hält} \\ \text{undefiniert sonst} \end{cases}$$

(iii) $\varphi_\pi: (A^*)^o \longrightarrow (A^*)^o$ ordnet dem Nulltupel () das Nulltupel () zu, falls π hält.

$$\varphi_\pi() = \begin{cases} (), \text{ falls } \pi \text{ hält} \\ \text{undefiniert sonst} \end{cases}$$

(2.4.3) Definition: Sei A fest gewählt.

(i) Eine Wortfunktion $f : (A^*)^r \longrightarrow (A^*)^s$, $r,s \geq o$, heißt PL(A)-berechenbar oder kurz berechenbar, falls ein Programm $\pi_f \in PL(A)$ existiert mit $\varphi_{\pi_f} = f$.

(ii) Die Menge $\mathcal{P}(A) := \{ \varphi_\pi \mid \pi \in PL(A) \}$ heißt Menge der PL(A)-berechenbaren Funktionen.

Um den intuitiven Begriff der Berechenbarkeit zu präzisieren, hat es immer wieder Versuche gegeben, Klassen von

„berechenbaren" Funktionen zu definieren . All diese Versuche haben zu der gleichen Menge von „berechenbaren" Funktionen geführt. So ist zum Beispiel die Menge der „mit Turingmaschinen berechenbaren" Funktionen mit der Menge der „partiell rekursiven" Funktionen identisch. Mit diesen Mengen wiederum ist bei festem A die Menge $\mathfrak{P}(A)$ identisch. Diese Identitäten geben Anlaß zur Church'schen These.

(2.4.4) Church'sche These:

Jede intuitiv berechenbare Funktion ist PL(A)-berechenbar und umgekehrt.

Aus (2.4.4) folgt: Sind A_1 und A_2 zwei voneinander verschiedene endliche Alphabete, so lassen sich offensichtlich die Mengen $\mathfrak{P}(A_1)$ und $\mathfrak{P}(A_2)$ identifizieren. Wir geben daher die Differenzierung nach dem zugrunde liegenden Alphabet auf und schreiben von nun an einfach $\mathfrak{P}$ für die Menge der berechenbaren oder partiell rekursiven Funktionen (siehe auch (2.4.6)). Wichtig ist die folgende Ergänzung zur Church'schen These.

(2.4.5) Ergänzung zur Church'schen These:

Zu jeder berechenbaren Funktion f läßt sich für beliebiges endliches A effektiv ein Programm $\pi \in PL(A)$ angeben mit $f = \varphi_\pi$

(2.4.6) Definition:

(i) $\mathfrak{P}_s^r := \{f \in \mathfrak{P} \mid f : (A^*)^r \longrightarrow (A^*)^s,\ r,s \geq 0\}$

(ii) $\mathfrak{R} := \{f \in \mathfrak{P} \mid f \text{ ist total}\}$ ist die Menge der total rekursiven Funktionen.

(iii) $\mathfrak{R}_s^r := \mathfrak{R} \cap \mathfrak{P}_s^r$, $r,s \geq 0$

2.5. Kodierungen, Gödelisierungen von $\mathcal{P}$

(2.5.1) Definition: Sei A endliches Alphabet. Eine Menge $B \subseteq (A^*)^r$, $r \geq 0$, heißt entscheidbar oder auch rekursiv genau dann, wenn es eine total rekursive Funktion $\chi_B : (A^*)^r \longrightarrow A^*$ gibt mit

$$\chi_B(x_1,\ldots,x_r) = \varepsilon \iff (x_1,..,x_r) \in B$$

(2.5.2) Definition: Seien A_1 und A_2 endliche Alphabete. Eine Funktion $\xi : A_1^* \longrightarrow A_2^*$ heißt genau dann Kodierung von A_1^* durch A_2^*, falls gilt:

i) $\xi \in \mathcal{R}$,
ii) ξ ist injektiv,
iii) $\xi(A_1^*)$ ist entscheidbar,
iv) $\xi^{-1} \in \mathcal{P}$

Sei $A_0 = \{1\}$. Dann kann man A_0^* mit den natürlichen Zahlen einschließlich der Null, $\mathbb{N}_0$, wie folgt identifizieren.

$$\begin{aligned} \varepsilon &\mathrel{\hat{=}} 0 \\ \underbrace{111.....1111111}_{n \text{ Einsen}} &\mathrel{\hat{=}} n \in \mathbb{N} \end{aligned}$$

(2.5.3) Definition: Eine Kodierung $\xi : A^* \longrightarrow A_0^* = \{1\}^* \triangleq \mathbb{N}_0$ heißt Gödelisierung. $\xi(\omega)$ heißt Gödelnummer von ω für alle $\omega \in A_0^*$.

Es soll im folgenden eine Gödelisierung aller PL(A)-Programme mit festem A angegeben werden. Damit wird gleichzeitig eine Gödelisierung von $\mathcal{P}$ angegeben. Sei $A := \{a_1,..,a_n\}$ fest gewählt. In der Menge B listen wir alle Sonderzeichen und alle Buchstaben auf, aus denen die Wortsymbole „input , if , fi usw." von PL(A) aufgebaut werden.

$B := \{:,=,\varepsilon,\bar{\varepsilon},\longrightarrow,;,,,\sqcup,\omega,\S,(,),a,c,d,e,f,g,h,i,l,n,o,p,s,t,u,w\}$

Die Mengen der Marken und Variablen in PL(A)-Programmen sind $M := \{L_1,L_2,\ldots\}$ bzw. $VR := \{V_1,V_2,\ldots\}$. Dabei bezeichnen L_i bzw. V_j, $i,j \geq 1$, irgendwelche Namen für Variable bzw. Marken. Es war bisher nicht nötig, die Wahl dieser Namen einzuengen. Für

die folgenden Überlegungen muß jedoch sichergestellt werden, daß die Namen Wörterüber einem endlichen Alphabet sind. Es wird daher wie folgt normiert.

Der Name L_1 ist textuell gleich „L1"
Der Name L_2 ist textuell gleich „L2" u.s.w.

entsprechend

Der Name V_1 ist textuell gleich „V1" u.s.w.

Damit gilt:

$M \subset \{L,Ø,1,...,9\}^*$, $VR \subset \{V,Ø,1,...,9\}^*$

Sei nun $C := A \cup B \cup \{L,V,Ø,1,...,9\}$. Jedes Programm $\pi \in PL(A)$ läßt sich somit als Wort aus C^* und $PL(A)$ selbst als Teilmenge von C^* auffassen. Wir geben eine injektive Abbildung $H : C \longrightarrow \{1\}^* \triangleq \mathbb{N}_0$ elementweise an:

:	⟼	0	c	⟼	13	u	⟼	26
=	⟼	1	d	⟼	14	w	⟼	27
;	⟼	2	e	⟼	15	L	⟼	28
,	⟼	3	f	⟼	16	V	⟼	29
(	⟼	4	g	⟼	17	0	⟼	30
)	⟼	5	h	⟼	18	⋮		
␣	⟼	6	i	⟼	19	⋮		
⟶	⟼	7	l	⟼	20	9	⟼	39
ε	⟼	8	n	⟼	21	a_1	⟼	40
$\bar{\varepsilon}$	⟼	9	o	⟼	22	⋮		
ρ	⟼	10	p	⟼	23	⋮		
ω	⟼	11	s	⟼	24	a_n	⟼	39+n
a	⟼	12	t	⟼	25			

Die injektive Abbildung H läßt sich zu einer ebenfalls injektiven Abbildung $H^* : C^* \longrightarrow \mathbb{N}_0^*$ erweitern.

$$H^*(\varepsilon) \longmapsto \varepsilon$$
$$H^*(\bar{x}y) \longmapsto H^*(\bar{x})H(y), \quad \forall y \in C, \forall \bar{x} \in C^*$$

(2.5.4) Lemma: H^* ist eine Kodierung von C^* durch $\mathbb{N}_0^*$.

Beweis: (i) Nach Definition von H und H^* ist H^* natürlich intuitiv berechenbar und auf Grund der

Church'schen These berechenbar. H^* ist für alle Elemente aus C^* definiert. Also ist H^* total und insgesamt aus $\mathcal{R}$.

(ii) H^* ist trivialerweise injektiv.

(iii) Sei $D := H^*(C^*)$. D ist Teilmenge von $\mathbb{N}_o^*$. Sei $\bar{i} \in \mathbb{N}_o^*$. $\bar{i}$ hat endliche Länge $l(\bar{i})$, $\bar{i}=m_1 \ldots\ldots m_{l(\bar{i})}$ [1] mit $m_j \in \mathbb{N}_o$ für $j \in [l(\bar{i})]$. $\bar{i}$ ist genau dann aus D, wenn jedes m_j ein Urbild in C bzgl. H hat. Um festzustellen, ob $\bar{i} \in D$ ist, sind also höchstens $l(\bar{i}) \cdot \mathrm{card}(C)$ Tests nötig. Es existiert also eine total rekursive Funktion

$$\chi_D : \mathbb{N}_o^* \longrightarrow \mathbb{N}_o^* \qquad \text{mit}$$

$\chi_D(\bar{i}) = \varepsilon \iff$ (Jedes Element von $\bar{i}$ hat unter H Urbild in C) $\iff \bar{i} \in D$.
Also ist $D = H^*(C^*)$ entscheidbar.

(iv) In (iii) wurde schon gezeigt, daß man für jedes $\bar{i} \in H^*(C^*)$ effektiv das Urbild in C^* ermitteln kann. Also ist $(H^*)^{-1}$ überall dort, wo es definiert ist, auch berechenbar. Also $(H^*)^{-1} \in \mathcal{P}$.

Aus (i) - (iv) folgt: H^* ist Kodierung. % [2]

Wir betrachten nun die total rekursive Funktion $f : \mathbb{N}_o^* \longrightarrow \mathbb{N}_o$

mit $\bar{i} \longmapsto \begin{cases} 0, \text{falls } \bar{i} = \varepsilon \\ p_1^{m_1} \cdot \ldots\ldots \cdot p_{l(\bar{i})}^{m_{l(\bar{i})}+1} \dot{-} 1, \text{falls } \bar{i}=m_1 \ldots\ldots m_{l(\bar{i})} \end{cases}$

wobei p_j die j-te Primzahl ist.

Behauptung: f ist bijektiv.

Beweis: (i) f ist injektiv wegen der Eindeutigkeit der Primfaktorzerlegung.

(ii) f ist surjektiv, weil jede Zahl $m \in \mathbb{N}$ mit $m > 1$ eine Primzahlzerlegung hat, in der mindestens ein Primfaktor vorkommt. %

---

1) Das Symbol l bezeichnet bei beliebigem Alphabet B die Längenfunktion für Elemente aus B^*.

2) „%" wird als Endemarkierung für Beweise benutzt.

Mit H^* als Kodierung von C^* durch $\mathbb{N}_o^*$ und $f \in \mathcal{R}$ als bijektiver Funktion von $\mathbb{N}_o^*$ nach $\mathbb{N}_o$ folgt Lemma (2.5.5).

(2.5.5) Lemma: Die Funktion $g := f \circ H^* : C^* \longrightarrow \mathbb{N}_o$ ist Gödelisierung von C^* durch $\mathbb{N}_o$.

Jedem Wort w aus C^* und somit jedem Programm aus PL(A) ist also eindeutig eine Zahl $f \circ H^*(w)$ aus $\mathbb{N}_o$ zugeordnet. Da H^* injektiv und f bijektiv ist, ist die Abbildung g eine bijektive Gödelisierung von C^*.

(2.5.6) Lemma: Die Menge $T := \{g(x) \mid x \in PL(A)\} \subset \mathbb{N}_o$ ist entscheidbar.

Beweis: I. i = O ist nicht aus T, da das leere Wort kein Programm ist.

II. Sei $i \in \mathbb{N}$. Dann läßt sich i eindeutig in der Form

$$i = p_1^{m_1} \cdot \ldots \cdot p_k^{m_k+1} \div 1 \quad \text{darstellen} \quad (k \geq 1).$$

Existiert für eines der m_j ($j \in [k]$) kein Urbild bzgl. der Funktion H. so ist i nicht aus dem Bildbereich von g und damit auch nicht aus T. Sei nun jedes m_j aus dem Bildbereich von H. Dann existiert ein $w \in C^*$ mit $g(w)=i$. Aus der Grammatik G aus 2.3. und den umgangssprachlichen Regeln wie „Die Eingabevariablen sind paarweise verschieden." läßt sich ein Programm konstruieren (vgl. : „Formale Sprachen", „Compilerbau"), das bei jeder Eingabe $w \in C^*$ hält und ε ausgibt genau dann, wenn $w \in PL(A)$ ist. Insgesamt existiert also eine total rekursive Funktion, deren Ergebnis genau dann gleich ε ist, wenn das Urbild eines Elementes aus $\mathbb{N}_o$, falls es existiert, ein gültiges PL(A)-Programm ist. Also ist T entscheidbar.

%

(2.5.7) Definition: $B \subset (A^*)^q, q \geq o$, heißt aufzählbar genau dann, wenn B Wertebereich einer partiell rekursiven Funktion ist.

(2.5.8) Lemma: $B \subseteq (A^*)^q$, $q \geq 0$, entscheidbar $\Rightarrow$ B ist aufzählbar.

Beweis: Sei $B \subseteq (A^*)^q$ entscheidbar. Nach Definition (2.5.1) existiert eine total rekursive Funktion

$$\chi_B : (A^*)^q \longrightarrow A^* \text{ mit}$$

$$\chi_B(x_1,\ldots,x_q) = \varepsilon \iff (x_1,\ldots,x_q) \in B$$

Aus χ_B läßt sich eine partiell rekursive Abbildung

$f_B : (A^*)^q \longrightarrow A^*$ gewinnen mit

$$f_B(x_1,\ldots,x_q) = \begin{cases} \varepsilon & \text{, falls } \chi_B(x_1,\ldots,x_q) = \varepsilon \\ \text{undefiniert sonst.} & \end{cases}$$

Dann ist die Abbildung $g_B : (A^*)^q \longrightarrow (A^*)^q$

mit $g_B(\bar{x}) = \overline{\prod}_1^2(\bar{x}, f_B(\bar{x}))$, $\bar{x} \in (A^*)^q$, partiell rekursiv und hat als Wertebereich die Menge B.[1)]

%

(2.5.9) Korollar: T ist aufzählbar.

Beweis: Folgt aus Lemma (2.5.7), da T entscheidbar ist.

%

Wie im Beweis von (2.5.6) kann man zeigen, daß auch für jedes $m,k \geq 0$ die Menge

$T_{m,k} := \{g(\pi) \mid \pi \in PL(A), \pi$ hat genau m Eingabe- und k Ausgabevariable$\} \subseteq \mathbb{N}_0$

entscheidbar ist. Man braucht dem Entscheidungsalgorithmus im Beweis von (2.5.6) nur noch einen Test, ob ein Programm $v \in PL(A)$ genau m Eingabe- und k Ausgabevariable aufweist, hinzuzufügen. Aus der Entscheidbarkeit von $T_{m,k}$ folgt die Aufzählbarkeit von $T_{m,k}$ und damit die Existenz einer partiell rekursiven Funktion von $\mathbb{N}_0$ nach $\mathbb{N}_0$, die die Menge $T_{m,k}$ als Wertebereich hat. Da $T_{m,k} \neq \emptyset$ ist, folgt sogar die Existenz einer total rekursiven Funktion (vgl. [5] Seite 82)

---

1) $\overline{\prod}_i^n$ bezeichnet in dem für n-Tupel üblichen Sinne die Projektion auf die i-te Komponente.

$t_{m,k} : \mathbb{N}_o \longrightarrow \mathbb{N}_o$ mit

$t_{m,k}(\mathbb{N}_o) = T_{m,k}$

Es hat also Sinn, wieder vom i-ten Element aus $T_{m,k}$ zu sprechen. Für alle $m,k \geq o$ läßt sich also auch die Menge aller Programme mit m Eingabe- und k Ausgabevariablen in der Form $\{\pi_o, \pi_1, \pi_2, \ldots\}$ hinschreiben. Dabei ist π_j dasjenige Programm aus PL(A) mit $t_{m,k}(j) = g(\pi_j)$.

Insgesamt existiert also für alle $m,k \geq o$ eine total rekursive Funktion

$$\gamma_{m,k} : \mathbb{N}_o \longrightarrow \{\pi \mid \pi \in PL(A),\ \pi \text{ hat genau m Eingabe- und k Ausgabevariable}\} =: W,$$

mit der „Umkehrung" $\overline{\gamma}_{m,k} : W \longrightarrow \mathbb{N}_o$ mit

$$\overline{\gamma}_{m,k}(\pi) := \min \{j \mid \gamma_{m,k}(j) = \pi\} .$$

Mit $T_{m,k}$ ist natürlich auch die Menge $\mathcal{P}_k^m$ für alle $m,k \geq o$ entscheidbar und damit aufzählbar. Die Menge $\mathcal{P}_k^m$ läßt sich also in der Form $\{f_o, f_1, f_2, \ldots\}$ hinschreiben, wobei für jedes f_j gilt: $f_j = \varphi_{\pi_j}$. Die Gödelnummer von π_j überträgt sich dabei auf f_j. In der obigen Aufzählung von $\mathcal{P}_k^m$ kommen alle Funktionen aus $\mathcal{P}_k^m$ mehrfach vor. Das bedeutet aber, daß die Funktionen aus $\mathcal{P}_k^m$ mehrere Gödelnummern besitzen. Es gilt sogar:

(2.5.10) Lemma: Für alle $f \in \mathcal{P}_k^m$ gilt: f hat bzgl. der Gödelisierung g unendlich viele Gödelnummern $(m,k \geq o)$.

Beweis: Seien $m,k \geq o$, $f \in \mathcal{P}_k^m$. f wird realisiert durch das Programm

π_o = input $X_1, \ldots, X_m$;
AW$_{\pi_o}$;
output $Z_1, \ldots, Z_k$

Es gilt $\varphi_{\pi_o} = f$. f wird aber auch realisiert durch die Programme $\pi_1, \pi_2, \pi_3, \ldots$

mit π_i = input $X_1, \ldots, X_m$; AW$_{\pi_o}$; $\underbrace{\xi; \ldots \xi;}_{\text{i-mal}}$ output $Z_1, \ldots, Z_k$

Es gilt $f = \varphi_{\pi_1} = \varphi_{\pi_2} = \varphi_{\pi_3} = ...$

Wegen $g(\pi_1) \neq g(\pi_2) \neq .....$ hat f unendlich viele Gödelnummern.

%

(2.5.11) Definition: Sei $\mathcal{F}$ eine Menge von Wortfunktionen, und sei $\mathcal{F}_s^r := \{f : (A^*)^r \longrightarrow (A^*)^s \mid f \in \mathcal{F}\}, r,s \geq 0$.
Eine Funktion $\psi \in \mathcal{F}_s^{r+1}$ heißt universell für $\mathcal{F}_s^r$, wenn gilt:

$$\mathcal{F}_s^r = \{\lambda \bar{y}\,[\psi(x,\bar{y})] \mid x \in A^*,\ \bar{y} \in (A^*)^r\}.\ ^{1)}$$

(2.5.12) Satz: Für alle $m,k \geq 0$ gilt:

Es gibt zu $\mathcal{P}_k^m$ eine universelle Funktion $\psi_{m,k} \in \mathcal{P}_k^{m+1}$.

Beweis: (mittels Church'scher These)
Die Menge aller Programme mit m Eingabe- und k Ausgabevariablen liege in aufgezählter Form, etwa durch $\gamma_{m,k}$, vor:

$\pi_0, \pi_1, \pi_2, ...$

Dann ist die Funktion

$$\psi_{m,k} := \lambda x,\bar{y}\left[\varphi_{\pi_{\gamma_{m,k}(x)}}(\bar{y})\right] \quad , \bar{y} \in (A^*)^m$$

universell für $\mathcal{P}_k^m$, und $\psi_{m,k}$ ist intuitiv berechenbar. Wegen der Church'schen These gibt es ein $\pi_u^{m,k} \in PL(A)$ mit m+1 Eingabe- und k Ausgabevariablen mit $\psi_{m,k} = \varphi_{\pi_u^{m,k}}$.

%

(2.5.13) Bemerkung: Man kann (2.5.12)auch beweisen, indem man auf komplizierte Weise direkt $\pi_u^{m,k}$ konstruiert.

---

1)Zur Lambda-Notation siehe [20] Seite 13.

2.6.Lexikographische Ordnung von A^*

Wir wollen eine weitere Gödelisierung von A^* einführen. Dazu definieren wir zunächst, was unter der lexikographischen Ordnung auf A^* zu verstehen ist.

(2.6.1) Definition: Sei $A = \{a_1,\ldots,a_n\}$. Die Nachfolgerfunktion $\nu : A^* \longrightarrow A^*$ ist definiert durch:

$$\begin{aligned} \nu(\varepsilon) &= a_1 \\ \nu(xa_i) &= xa_{i+1} \\ \nu(xa_n) &= \nu(x)a_1 \end{aligned} \qquad \text{für } i \in [n-1],\ x \in A^*,\ a_i \in A$$

(2.6.2) Lemma: Die Nachfolgerfunktion $\nu : A^* \longrightarrow A^*$ ist bijektiv.

Beweis: Zunächst ist ν injektiv, da zwei Elemente aus A^* nur dann den gleichen Nachfolger haben können, wenn sie gleich sind. ν ist surjektiv, da man durch wiederholte Anwendung von ν jedes Wort aus A^* auf das leere Wort reduzieren kann. Also $\{\nu^i(\varepsilon) \mid i \in \mathbb{N}\} = A^*$.

%

Wegen $w_i = \nu^i(\varepsilon)$ für jedes Element w_i aus A^* erhält man eine Ordnung auf A^*. [5]

(2.6.3) Definition: Seien $w_i = \nu^i(\varepsilon)$ und $w_j = \nu^j(\varepsilon)$ aus A^*, dann ist $w_i \leq w_j$ genau dann, wenn $i \leq j$ gilt. Diese Ordnung heißt lexikographische Ordnung auf A^*.

Man kann also alle Wörter aus A^* eindeutig in lexikographischer Reihenfolge auflisten:

$$\varepsilon = w_0 < w_1 < w_2 \ldots\ldots$$

(2.6.4) Satz: Sei $A = \{a_1,\ldots,a_p\}$ und $A_o = \{1\}$. Sei $C_p : A^* \longrightarrow \mathbb{N}_o \triangleq \{1\}^*$ die Abbildung, die jedem $x \in A^*$ die Nummer von x in der lexikographischen

Reihenfolge zuordnet, also
$x \longleftrightarrow i$ mit $\nu^i(\epsilon) = x$.
Dann ist C_p eine bijektive Gödelisierung.

Beweis: (i) C_p ist total, da C_p auf ganz A^* definiert ist. Für jedes x läßt sich $C_p(x)$ durch endlich viele Anwendungen der Definition von ν effektiv ermitteln. Also ist $C_p \in \mathcal{R}$.

(ii) C_p ist injektiv, da es keine zwei Elemente aus A^* gibt mit gleicher Stellung in der lexikographischen Reihenfolge.

(iii) Wegen $C_p(A^*) = \mathbb{N}_o$ ist $C_p(A^*)$ natürlich entscheidbar.

(iv) Beginnend mit dem leeren Wort kann man für jedes $i \in \mathbb{N}_o$ mit Hilfe von ν effektiv alle Worte aus A^* bis zum i-ten Wort in lexikographischer Reihenfolge erzeugen. Dieses i-te Wort ist das Urbild von i . Also ist C_p^{-1} berechenbar und wegen $C_p(A^*) = \mathbb{N}_o$ sogar total. Also $C_p^{-1} \in \mathcal{R}$.

Aus (i) bis (iv) folgt: C_p ist Gödelisierung.
Wegen (ii) und $C_p(A^*) = \mathbb{N}_o$ folgt: C_p ist bijektiv.

%

(2.6.5) Lemma: Seien $A = \{a_1,\ldots,a_n\}$ und $B = \{b_1,\ldots,b_m\}$ zwei Alphabete, dann ist die Funktion

$$\mathfrak{F}_{n,m} := C_m^{-1} \circ C_n : A^* \longrightarrow B^*$$

eine bijektive Kodierung von A^* durch B^*.

Beweis: Offensichtlich.

%

2.7. Reduktion auf jeweils eine Eingabe- und eine Ausgabevariable

Wir wollen zeigen, daß es möglich ist, jedes Programm π aus PL(A) mit r Eingabe- und s Ausgabevariablen in ein

„äquivalentes" Programm π' mit nur jeweils einer Eingabe- und Ausgabevariablen zu transformieren. Wegen der Entsprechung von $\mathcal{P}$ und PL(A) genügt es demnach also, die Funktionen aus $\mathcal{P}_1^1$ zu betrachten, um ganz $\mathcal{P}$ zu behandeln. Die universelle Funktion für $\mathcal{P}_1^1$ ist in diesem Sinne dann universell für ganz $\mathcal{P}$.

Sei nun $\pi \in PL(A)$ mit r Eingabe- und s Ausgabevariablen, $r,s \geq 1$. Dann hat π folgenden Aufbau.

π = <u>input</u> $X_1,\ldots,X_r$;
AW_π;
<u>output</u> $Z_1,\ldots,Z_s$

Als Eingabebelegung für π kommt jedes Element $\bar{x}=(x_1,\ldots,x_r)$ aus $(A^*)^r$ vor. $\bar{x}$ kann man auch wie folgt darstellen:

$\bar{x} = x_1|x_2|\ldots|x_r \in (A \cup \{|\})^*$, dabei sei $| \notin A$.

In dieser Form wird $\bar{x}$ als Eingabe für ein zu π gleichwertiges Programm π' mit nur einer Eingabe- und einer Ausgabevariablen benutzt.

π' = <u>input</u> X;
$[X_1:=x_1;\ldots..X_r:=x_r]$;
AW_π;
$[Z:=z_1|\ldots..|z_s]$;
<u>output</u> Z

Die Programmteile in Klammern lassen sich wie folgt realisieren:

$[X_1:=x_1;\ldots..X_r:=x_r]$

durch:

$X_1:=\varepsilon;\ldots..X_r:=\varepsilon;$
<u>loop</u> X <u>case</u> $a_1 \longrightarrow X_r:=X_r a_1,$
$\vdots$ $\vdots$
$a_n \longrightarrow X_r:=X_r a_n,$
$| \longrightarrow X_1:=X_2;$
$X_2:=X_3;$
$\vdots$
$X_{r-1}:=X_r;$
$X_r:=\varepsilon,$
<u>end</u>

und $\left[Z:=z_1|\ldots\ldots|z_s\right]$

durch:

```
Z:=ε;
loop Z_1 case a_1 ⟶ Z:=Za_1,
                ⋮          ⋮
             a_n ⟶ Z:=Za_n,
     end;
Z:=Z|;
loop Z_2 case a_1 ⟶ Z:=Za_1,
                ⋮          ⋮
             a_n ⟶ Z:=Za_n,
     end;
Z:=Z|;
  ⋮
Z:=Z|;
loop Z_s case a_1 ⟶ Z:=Za_1,
                ⋮          ⋮
             a_n ⟶ Z:=Za_n,
     end
```

π'ist nicht Element aus PL(A), da das zugrunde liegende Alphabet nicht A sondern $A \cup \{|\} \neq A$ ist. Wir können aber aus π' ein ebenfalls zu π gleichwertiges Programm $\pi^* \in PL(A)$ mit nur einer Ein- und einer Ausgabevariablen gewinnen, indem wir $\{A \cup \{|\}\}^*$ durch A^* kodieren. Als Kodierung können wir die Funktion

$$\xi_{n+1,n} := C_n^{-1} \circ C_{n+1}$$

benutzen, wobei die Funktionen C_n^{-1} und C_{n+1} in 2.6. definiert sind.

Im wesentlichen lassen sich also alle Programme aus PL(A) auf eine Eingabe- und eine Ausgabevariable reduzieren, falls überhaupt Ein- bzw. Ausgabevariable vorhanden sind. Auf Grund der Church'schen These genügt es also, statt $\mathcal{P}$ die Menge $\mathcal{P}_1^1$ zu betrachten. Die universelle Funktion $\Psi_{1,1}$ für $\mathcal{P}_1^1$ ist in diesem Sinne universell für ganz $\mathcal{P}$. Um diesen Tatbestand deutlich zu machen, führen wir die

folgende Schreibweise ein.

(2.7.1) Definition: Sei $\psi_{1,1}$ universelle Funktion für $\mathcal{P}_1^1$.

$$\varphi_x^k := \lambda y_1,\ldots,y_k\left[\psi_{1,1}(x, \xi_{n+1,n}(y_1|\ldots|y_k)\right],$$

$x, y_1, \ldots, y_k \in A^*$.

Dabei ist $\varphi_x^k : (A^*)^k \longrightarrow A^*$.

2.8. s-m-n-Theorem, Rekursionstheorem

Im folgenden werden wir das Rekursionstheorem beweisen, das uns ermöglicht, die Existenz selbstreproduzierender PL(A)-Programme nachzuweisen. Sei A so gewählt, daß jedes PL(A)-Programm in A^* liegt (Vermeidung von Umkodierungen).

(2.8.1) Satz (s-m-n-Theorem):

Für alle $m,n \in \mathbb{N}$ gibt es eine Funktion $s_n^m \in \mathcal{R}_1^{m+1}$, die für alle $x \in A^*$, $\overline{y} \in (A^*)^m$, $\overline{z} \in (A^*)^n$ folgende Gleichung erfüllt:

$$\varphi_x^{m+n}(\overline{y},\overline{z}) = \varphi^n_{s_n^m(x,\overline{y})}(\overline{z})$$

Beweis: Seien $n,m \in \mathbb{N}$ fest gewählt.

1.Fall: x ist kein gültiger Programmtext. Dann ist φ_x^{m+n} undefiniert. In diesem Falle muß $\varphi^n_{s_n^m(x,\overline{y})}$ auch undefiniert sein. Es wird deshalb $s_n^m(x,\overline{y}) = \varepsilon$ gesetzt. Nun ist auch $s_n^m(x,\overline{y})$ kein gültiger Programmtext, und es gilt:

$$\varphi_x^{m+n}(\overline{y},\overline{z}) = \varphi^n_{s_n^m(x,\overline{y})}(\overline{z}) = \text{undefiniert} \quad .$$

2.Fall: x ist ein gültiger Programmtext. Also $x \in PL(A)$. Dann hat x den Aufbau:

x = input $Y_1,\ldots,Y_m,Z_1,\ldots,Z_n$;
AW_x;
output W

Mit $\bar{y} = (y_1, y_2, \ldots, y_m)$ wird gesetzt:

$s_n^m(x,\bar{y})$:=<u>input</u> $Z_1, \ldots, Z_n$;
$[Y_1 := y_1]; \ldots; [Y_m := y_m]$;
AW_x;
<u>output</u> W

Die Programmteile in eckigen Klammern lassen sich leicht - wie am Beispiel von $[Y_1 := y_1]$ gezeigt - realisieren:
y_1 ist Element aus A^* und hat somit endliche Länge $l(y_1) \in \mathbb{N}_0$

$$y_1 = a_{1_1} \ldots\ldots a_{1_{l(y_1)}} \quad \text{mit } a_{1_j} \in A^*.$$

Damit wird $[Y_1 := y_1]$ realisiert durch

$Y_1 := \varepsilon$;
$Y_1 := Y_1 a_{1_1}$
$\vdots$
$Y_1 := Y_1 a_{1_{l(y_1)}}$;

Auf Grund der Church'schen These ist $s_n^m \in \mathcal{R}_1^{m+1}$.

%

<u>(2.8.2) Korollar:</u> Es gibt ein $h \in \mathcal{R}_1^m$, so daß für alle $f \in \mathcal{P}_1^{m+n}$ gilt:

$$f(\bar{y},\bar{x}) = \varphi^n_{h(\bar{y})}(\bar{x}) \quad \text{für alle } \bar{x} \in (A^*)^m,\ \bar{y} \in (A^*)^n.$$

<u>Beweis:</u> Da f eine berechenbare Funktion ist, gibt es ein Programm $\pi_o \in A^*$, mit $f = \varphi^{m+n}_{\pi_o}$. Aus (2.8.1) folgt

$$f(\bar{y},\bar{x}) = \varphi^{m+n}_{\pi_o}(\bar{y},\bar{x}) = \varphi^{n}_{s^m_n(\pi_o,\bar{y})}(\bar{x}) .$$

Setzt man $h := \lambda \bar{y}\left[s^m_n(\pi_o,\bar{y})\right]$, so folgt das Korollar.

%

(2.8.3) Bezeichnung: Sei $g \in \mathcal{P}_1^{k+1}$. Dann existiert ein Programm $x \in A^*$ mit

$$g = \varphi_x^{k+1} .$$

Wir führen nun folgende Notation ein:

$$g_x := \varphi^{k}_{s^1_k(x,y)}$$

Es gilt mit dieser Notation:
$g_x(\bar{y}) = g(x,\bar{y})$ für alle $\bar{y} \in (A^*)^k$.

Sei $h \in \mathcal{P}_1^r$ und $h(\bar{x})$ = undefiniert für ein $\bar{x} \in (A^*)^r$, so ist nach obiger Notation die Funktion $g_{h(\bar{x})}$ überall undefiniert.

Wir sind nun in der Lage, das Kleene'sche Rekursionstheorem in der folgenden Fassung zu beweisen:

(2.8.4) Satz (Rekursionstheorem, Formulierung wie in [5]):

Zu jeder Funktion $f \in \mathcal{P}_1^1$ gibt es einen Text $x \in A^*$, so daß gilt:

$$\varphi_x = \varphi_{f(x)}$$

Beweis: Die Funktion

$$g = \lambda y,x\left[\varphi_{\varphi_y(y)}(x)\right]$$

liegt in $\mathcal{P}_1^2$ mit $x,y \in A^*$. In Korollar (2.8.2) wurde gezeigt, daß ein $h \in \mathcal{R}_1^1$ existiert mit

(1) $\varphi_{h(y)} = g_y = \varphi_{\varphi_y(y)}$ für alle $y \in A^*$

Sei nun $f \in \mathcal{P}_1^1$. Da $h \in \mathcal{R}_1^1$ ist, kann man die Hintereinanderausführung von f und h betrachten. $f \circ h$ liegt ebenfalls in $\mathcal{P}_1^1$. Auf Grund der Church'schen These kann man zu $f \circ h$ effektiv einen Programmtext π aus PL(A) angeben mit

(2) $\varphi_\pi = f \circ h$

Aus (1) und (2) folgt dann zusammen:

$$\varphi_{h(\pi)} \overset{(1)}{=} \varphi_{\varphi_\pi(\pi)} \overset{(2)}{=} \varphi_{f(h(\pi))}$$

Damit ist $x = h(\pi)$ das gesuchte x.

%

(2.8.5) Definition: Sei $f \in \mathcal{P}_1^1$. Ein Element $x \in A^*$ heißt Fixpunkt zu f, falls gilt $\varphi_x = \varphi_{f(x)}$.

(2.8.6) Korollar: Zu jeder Funktion $g \in \mathcal{P}_1^2$ existiert ein Text $x_o \in A^*$ mit $\varphi_{x_o} = g_{x_o}$.

Beweis: Nach Korollar (2.8.2) existiert eine Funktion $h \in \mathcal{R}_1^1$ mit $\varphi_{h(y)} = g_y$ für alle $y \in A^*$.

h hat nach dem Rekursionstheorem einen Fixpunkt x_o mit $\varphi_{x_o} = \varphi_{h(x_o)} = g_{x_o}$.

%

(2.8.7) Satz: Es existiert in $\mathcal{P}_1^1$ eine Funktion mit einem Programmtext $x_o \in A^*$, der für jede Eingabe $y \in A^*$ seinen eigenen Text x_o ausgibt.

Beweis: Die Funktion $g = \prod_1^2 : (A^*)^2 \longrightarrow A^*$ mit $g(x,y) := x$ für alle $x,y \in A^*$ ist trivialerweise aus $\mathcal{P}_1^2$.
Aus Korollar (2.8.6) folgt damit, daß die Gleichung $\varphi_x = g_x$ eine Lösung x_o besitzt. Es existiert also ein $x_o \in A^*$ mit

$$\varphi_{x_o} = g_{x_o} = \lambda y[x_o] \quad \Rightarrow \quad \varphi_{x_o}(y) = x_o \quad \forall\, y \in A^*$$

φ_{x_o} ist also eine Funktion, die bei jeder Eingabe $y \in A^*$ ihren eigenen Programmtext x_o ausgibt. Mit $\varphi_{x_o} \in \mathcal{P}_1^1$ (trivial) ist der Satz vollständig bewiesen.

%

Der folgende Satz ist eine Verallgemeinerung von Satz (2.8.7).

(2.8.8) Satz: Sei $f : A^* \longrightarrow A^*$ aus $\mathcal{P}_1^1$. Dann existiert in $\mathcal{P}_1^1$ eine Funktion mit einem Programmtext $x_o \in A^*$, der für jede Eingabe den Wert $f(x_o)$ ausgibt.

Beweis: Sei $f \in \mathcal{P}_1^1$. Die Funktion $g : (A^*)^2 \longrightarrow A^*$ mit

$$g(x,y) := \prod_1^2(f(x),y) = f(x), \quad x,y \in A^*,$$

liegt in $\mathcal{P}_1^2$. Dies folgt aus der Abgeschlossenheit von $\mathcal{P}$ gegenüber der Kombination und der Substitution von Funktionen (vgl. etwa [5]). Aus (2.8.6) folgt damit, daß die Gleichung $\varphi_x = g_x$ eine Lösung x_o besitzt. Es existiert also ein $x_o \in A^*$ mit

$$\varphi_{x_o} = g_{x_o} = \lambda y\left[f(x_o)\right] \Rightarrow \varphi_{x_o}(y) = f(x_o) \; \forall_{y \in A^*}$$

Da φ_{x_o} eine konstante Funktion ist, liegt φ_{x_o} trivialerweise in $\mathcal{P}_1^1$. φ_{x_o} ist die gesuchte Funktion.

%

Aus Satz (2.8.8) folgt, daß es PL(A)-Programme gibt, die sich nicht nur einfach selbstreproduzieren, sondern ihren eigenen Text mehrmals ausgeben.

(2.8.9) Korollar: Für jedes $i \in \mathbb{N}$ existiert eine Funktion mit einem Programmtext $x_{i_o} \in A^*$, der für jede Eingabe $y \in A^*$ seinen eigenen Text x_{i_o} i-mal hintereinander ausgibt.

Beweis: Sei $i \in \mathbb{N}$. Dann ergibt sich der Beweis aus dem Beweis von Satz (2.8.8) mit

$f = f_i : A^* \longrightarrow A^*$

$f_i(x) := x^i$ ($= \underbrace{x.....x}_{\text{i-mal}}$).

%

(2.8.10) Bemerkung: Satz (2.8.7) ergibt sich als Spezial-

fall von Satz (2.8.8) mit f = id.

Mit Hilfe der vorangegangenen Sätze haben wir die Existenz selbstreproduzierender PL(A)-Programme auf theoretischem Wege nachgewiesen. Die Beweise sind zwar konstruktiv, jedoch lassen sich die Konstruktionen nicht nachvollziehen, um ein konkretes selbstreproduzierendes PL(A)-Programm zu erzeugen. In 2.5. haben wir gesehen, daß die Menge aller PL(A)-Programme aufzählbar ist. Zählt man die Menge aller PL(A)-Programme auf, etwa in lexikographischer Reihenfolge, so garantiert Satz (2.8.7) die Existenz einer Zahl $i_o \in \mathbb{N}_o$ mit π_{i_o} ist selbstreproduzierend. Für die Zahl i_o ist jedoch eine Größenordnung zu erwarten, die Aufzählung als Mittel zur Gewinnung eines selbstreproduzierenden PL(A)-Programms ausschließt.

Die Bedeutung von Kapitel 2 liegt darin, daß es nicht prinzipiell sinnlos ist, selbstreproduzierende Programme in höheren Programmiersprachen zu suchen; sie existieren wirklich.

(2.8.11) Bemerkung: In 4.3. werden zyklisch selbstreproduzierende Programme behandelt (vgl. Definition (4.3.1)). Die Existenz zyklisch selbstreproduzierender Programme läßt sich wahrscheinlich ebenfalls aus dem Rekursionstheorem folgern.

3. Selbstreproduzierende Programme in realen Programmiersprachen - Einige Beispiele

3.1 Einleitung

In Kapitel 2 wurde gezeigt, daß in der fiktiven Programmiersprache PL(A) selbstreproduzierende Programme existieren. Da PL(A) die gleiche „Berechenkapazität" wie alle gängigen Programmiersprachen hat, müssen auch in konkreten Programmiersprachen selbstreproduzierende Programme existieren. Ausgehend von praktischen Überlegungen werden im folgenden einige Beispiele für möglichst kurze selbstreproduzierende Programme in den höheren Programmiersprachen SIMULA und PASCAL konstruiert. Wir werden dabei sowohl auf selbstreproduzierende Programme stoßen, die sich ohne weiteres auf realen Rechenanlagen implementieren lassen, als auch Programme finden, die zwar syntaktisch korrekt sind, sich aber aus verschiedensten Gründen nicht realisieren lassen. Wo es möglich ist, werden aus letzteren Programmen implementierbare Versionen gewonnen.

In Abschnitt 3.4. werden einige Beispiele für selbstreproduzierende Programme in einer maschinenorientierten Sprache (SIEMENS-Assemblersprache) angegeben (vgl. 3.4.).

3.2 Selbstreproduzierende Programme in SIMULA [1)]

In diesem Abschnitt sollen selbstreproduzierende Programme in der Programmiersprache SIMULA entwickelt werden. SIMULA steht hier als Beispiel für eine blockorientierte Programmiersprache. Die in 3.3. behandelte Sprache PASCAL ist dagegen nicht blockorientiert. Für uns wird sich jedoch ein anderes Unterscheidungsmerkmal als wichtiger erweisen. Es handelt sich dabei um die Verfügbarkeit von Textvariablen. Während PASCAL nur einfache Textkonstanten kennt, kann in SIMULA mit echten Textvariablen operiert werden. Durch Integration in das SIMULA-Klassenkonzept kann die Bearbeitung von Variablen des Typs text in SIMULA sehr komfortabel

1) SIMULA-Beschreibung in [19].

sein. Dies nutzen wir in Abschnitt 3.2.5. aus.

3.2.1.Naiver Ansatz

Um eine Vorstellung von der Problematik, ein selbstreproduzierendes SIMULA-Programm π anzugeben, zu erhalten, betrachten wir folgenden naiven Ansatz:

π enthält im wesentlichen nur eine Ausgabeanweisung. Diese Anweisung gibt den ganzen Programmtext von π aus.

Ein solcher Ansatz führt zu folgendem Programm π_o:

π_o = begin OUTTEXT(".....") end;

An dieser Stelle muß der Programmtext von π_o erscheinen, also:

begin OUTTEXT(".....") end;

siehe oben

Insgesamt entsteht also ein sich rekursiv aufblähendes Programm, das sich auch wie folgt schreiben läßt:

```
πo = begin OUTTEXT("BEGIN OUTTEXT("BEGIN
                   OUTTEXT("BEGIN OUTTEXT(".....
                   ..............................
                   ............END") END") END")
     end
```

π_o ist natürlich kein endlicher Text und damit kein Programm mehr. Die „Unmöglichkeit" von π_o läßt sich auch an der Unerfüllbarkeit der Textgleichung

x = begin OUTTEXT("x") end

zwischen dem Text x und den Konstanten begin OUTTEXT(" und ") end ablesen: Texte verschiedener Länge können nicht gleich sein!

3.2.2.Textzerlegung und Algorithmus

Aus 3.2.1. folgt, daß ein selbstreproduzierendes SIMULA-Programm π seinen eigenen Text nicht en bloc mit einer einzigen Ausgabeanweisung ausgeben kann. π muß seinen Text also in mehreren Schritten aus einigen Teilstrings zusammensetzen. Es muß also eine

(i) Zerlegung des Textes π

vorgenommen werden. Da wir über die Art der Zerlegung nichts wissen, versuchen wir es zunächst mit der totalen Zerlegung von π , das heißt, wir zerlegen π in einzelne Zeichen. Belassen wir es bei dieser Maßnahme, so kommen wir zu folgendem Programm.

```
π1 = begin OUTTEXT("B");
           OUTTEXT("E");
           OUTTEXT("G");
           OUTTEXT("I");
           OUTTEXT("N");
           OUTTEXT("␣");
           OUTTEXT("O");
                ⋮
```

Es ist klar, daß π_1 einen unendlichen Text darstellt und damit kein Programm sein kann.

Die Unendlichkeit von π_1 liegt darin begründet, daß zur Ausgabe eines Zeichens eine Anweisung - und damit auch ein Text - bestehend aus insgesamt 13 Zeichen notwendig ist. Damit ist ein rein sequentielles Programm wie π_1 zum Scheitern verurteilt, wenn der Programmtext in einzelne Buchstaben zerlegt wird. Die Wahl eines der Zerlegung des Programmtextes entsprechend strukturierten

(ii) Algorithmus

ist ein bedeutendes Kriterium, das bei der praktischen Konstruktion selbstreproduzierender Programme beachtet

werden muß.

(i) und (ii) stellen die beiden wichtigsten Aspekte selbstreproduzierender Programme dar. Bei den folgenden Konstruktionen selbstreproduzierender Programme wird es also darum gehen, eine geschickte Zerlegung des Programmtextes und einen geeigneten Ausgabealgorithmus zu finden.

3.2.3.Ein tabellengesteuertes Programm

Wir greifen mit Programm π_2 die Idee von der totalen Zerlegung des Textes π_2 in einzelne Zeichen aus 3.2.2. auf. Diese Zerlegung wird aber nicht wie in π_1 explizit sichtbar, sondern sie äußert sich im verwendeten Algorithmus. Dieser Algorithmus setzt den Text von π_1 aus einzelnen Zeichen zusammen. Die Menge der zulässigen Zeichen steht im Algorithmus in Form eines Feldes

character array C [Ø : maxchar]

zur Verfügung. Jedes Element von C enthält genau ein Zeichen, das zur Erstellung von SIMULA-Programmen verwendet werden darf. Alle derartigen Zeichen sind in C enthalten.

```
character array C [Ø : maxchar];
        :
C[Ø]:="A";        ┐
C[1]:="B";        │
        :         │   Buchstaben
C[25]:="Z";       ┘
C[26]:="Ø";       ┐
        :         │   Ziffern
C[35]:="9";       ┘
C[36]:=";";       ┐
C[37]:=":";       │   Sonderzeichen
        :         │
C[maxchar]:="*";  ┘
```

Der Algorithmus hat die Aufgabe, sukzessive Komponenten

aus C auszugeben, so daß insgesamt der Programmtext π_2 gedruckt wird:

```
while not p do
begin  ⟨berechne neues I⟩ ;
       OUTCHAR(C[I]);
       ⟨setze p⟩
  end
```

I ist vom Typ integer, p ist ein Prädikat, das den Algorithmus stoppt, sobald der Text π_2 gedruckt ist.

Insgesamt sieht das Programm π_2 wie folgt aus:

π_2 =

```
begin integer I;
      :                                          Ia
      C[Ø]:="A";
      C[1]:="B";
           :                                     II
           :
      C[maxchar]:="*";
      while not p do
      begin ⟨berechne neues I⟩;                  III
            OUTCHAR(C[I]);
            ⟨setze p⟩
      end
  end                                            Ib
```

Das Programm π_2 gliedert sich in 4 Teile. Man erkennt neben den initialisierenden und abschließenden Teilen Ia und Ib einen Teil II, der den Aufbau der Druckzeichentabelle vornimmt, und einen Teil III, der den Algorithmus realisiert.

Die Programmteile Ia,Ib und II bereiten sicherlich keine Schwierigkeiten. Auch der Algorithmus von Teil III macht einen durchsichtigen Eindruck. Es bleiben eigentlich nur noch die beiden Anweisungen ⟨berechne neues I⟩ und ⟨setze p⟩ durch SIMULA-statements zu ersetzen. Das Ersetzen von ⟨berechne neues I⟩ ist dabei sicherlich die schwierigere Aufgabe.

Wir wollen zunächst genauer untersuchen, was der Algorithmus aus Teil III und insbesondere die Anweisung ⟨berechne neues I⟩ leisten müssen.

(3.2.3.1) Definition: Sei D die Menge aller in SIMULA-Programmen zulässigen Zeichen:

$D := \{a,b,\ldots,z,\emptyset,1,\ldots,9,;,:,\ldots,*\}$

Dann ist $\forall \alpha \in D$ die Zahl $i_\alpha \in \mathbb{N}_0$ der Index, unter dem das Zeichen $\alpha \in D$ in der Tabelle C abgelegt ist: $\alpha = C[i_\alpha]$

(3.2.3.2) Lemma: Die Abbildung $\delta : D^* \longrightarrow \mathbb{N}_0^*$ mit

$\delta(\varepsilon) = \varepsilon$

$\delta(w\alpha) = \delta(w)i_\alpha \qquad \forall\, w \in D^*, \alpha \in D$

ist Kodierung von D^* durch $\mathbb{N}_0^*$.

Beweis: (i) $\delta(x)$ ist für jedes $x \in D^*$ definiert. Also ist δ total. δ ist trivialerweise berechenbar.

(ii) Da in der Tabelle C jedes Zeichen aus D genau einmal gespeichert ist, folgt: δ ist injektiv.

(iii) Sei $\bar{j} = j_1 \ldots\ldots j_n$ aus $\mathbb{N}_0^*$. $\bar{j}$ ist genau dann aus $\delta(D^*)$, wenn jedes j_k, $k \in [n]$, aus $\{\emptyset,\ldots\ldots,\text{maxchar}\}$ ist. Also ist $\delta(D^*)$ entscheidbar.

(iv) Sei $\bar{j} = j_1 \ldots\ldots j_n$ aus $\delta(D^*)$. Mit Hilfe der Tabelle C läßt sich für jedes j_k, $k \in [n]$, das Zeichen aus D ermitteln, das durch j_k kodiert wird. Mit höchstens $n \cdot \text{maxchar}$ Vergleichen läßt sich so das Urbild von $\bar{j}$ unter δ ermitteln. Also ist δ^{-1} berechenbar.

Aus (i) bis (iv) folgt: δ ist Kodierung (vgl.(2.5.2)).

%

(3.2.3.3) Bemerkung: Jedes SIMULA-Programm π hat als endliches Wort aus D^* eine Kodierung $\delta(\pi)$ in $\mathbb{N}_0^*$.

Bei jedem Durchlauf durch die while-Schleife im Algorithmus von π_2 wird ein neuer Wert für I berechnet. I nimmt also im Verlauf des Programms eine Folge von Werten an, die sich als Wort aus $\mathbb{N}_0^*$ auffassen läßt:

$$I = i_1, i_2, \ldots\ldots, i_{l(\pi_2)} \qquad i_j \in \mathbb{N}_0 \text{ für alle } j \in [l(\pi_2)]$$

Damit π_2 seinen eigenen Text ausgeben kann, muß gelten:

$$C[i_1]\ C[i_2] \ldots\ldots C[i_{l(\pi_2)}] \overset{!}{=} \pi_2$$

Es gilt daher:

- Die Funktion von ⟨berechne neues I⟩ ist die sukzessive Ermittlung der Kodierung von π_2 bezüglich δ.
- Die Funktion des gesamten Algorithmus von π_2 ist die Dekodierung der ermittelten Kodierung, also die Realisierung von δ^{-1}.

Die Berechnung der i_j, $j \in [l(\pi_2)]$, ist das noch verbleibende Problem. Die i_j müssen iterativ mittels einer Funktion $F : \mathbb{N}_0 \longrightarrow \mathbb{N}_0$ erzeugt werden:

```
Setze i_1;
i_{j+1} := F(i_j);    j ∈ [l(π_2)-1]
```

.

Die Funktion F läßt sich als Funktionsprozedur realisieren und innerhalb von Teil Ia von π_2 vereinbaren. Die Anweisung ⟨berechne neues I⟩ wird dann zu der Prozeduranweisung

$$I := F(I)$$

Da es unser Ziel ist, ein selbstreproduzierendes SIMULA-Programm anzugeben, das sich auch auf einer konkreten Rechenmaschine implementieren läßt, müssen an F folgende Forderungen gestellt werden:

a) F muß in vertretbarer Zeit berechenbar sein
und
b) die von F benutzten Zwischenergebnisse müssen im darstellbaren Zahlenbereich der Rechenmaschine liegen.

Es ist möglich, daß ein konkretes F nicht von I abhängt.

3.2.4.Wahl der Iterationsfunktion F

In diesem Abschnitt werden zwei Funktionen diskutiert, die als Iterationsfunktion denkbar wären.

3.2.4.1.Eine Iterationsfunktion mittels Modulo-Bildung

Wir erweitern die Kodierung δ zu einer Gödelisierung (vgl.(2.5.3)). Dazu definieren wir die Abbildung f_δ.

(3.2.4.1.1) Definition: Die Abbildung $f_\delta : \mathbb{N}_o^* \longrightarrow \mathbb{N}_o$ sei wie folgt definiert:

$$f_\delta(\bar{I}) = \begin{cases} \emptyset, \text{ falls } \bar{I} = \varepsilon \\ \sum_{j=1}^{k} i_j (\text{maxchar}+1)^{j-1}, \text{ falls } \bar{I} = i_1 \ldots i_k \end{cases}$$

Ist $\bar{I} \in \delta(D^*)$, so ist jedes i_j, $j \in [k]$, kleiner oder gleich (maxchar+1). Die Restriktion von f_δ auf $\delta(D^*)$ ist somit injektiv und kodiert die Elemente von $\delta(D^*)$ durch $\mathbb{N}_o$. Es folgt somit:

(3.2.4.1.2) Lemma: Die Abbildung $f : D^* \longrightarrow \mathbb{N}_o$ mit $f := f_\delta \circ \delta$ ist Gödelisierung von D^*.

Beweis: Offensichtlich

%

Von Interesse ist für uns die Tatsache, daß man aus der Zahl f(x) für jedes $x \in D^*$ effektiv die Komponenten von

$\delta(x)$ zurückberechnen kann. Dies gilt insbesondere für $f(\pi_2)$, und wir gewinnen mit

```
integer procedure F;
begin F:=X mod(maxchar+1);
      X:=X//(maxchar+1);
  end        ↑————— ganzzahlige Division
```

eine Iterationsvorschrift zur Erzeugung von $\delta(\pi_2)$, wenn wir für X den Startwert $f(\pi_2)$ wählen.

Da der Startwert von X, also $f(\pi_2)$, von π_2 nicht eingelesen werden darf, muß π_2 die Zuweisung

$$X := f(\pi_2)$$

enthalten. $f(\pi_2)$ ist eine ganze Zahl und damit textueller Bestandteil von π_2. Zum Zeitpunkt der Erstellung von π_2 ist die Zahl $f(\pi_2)$ unbekannt, sie kann erst nachträglich ermittelt werden. Beim Aufschreiben des Programms π_2 muß in der Anweisung X:=f(π_2); die Zahl $f(\pi_2)$ zunächst ausgelassen werden. Erst nach Erstellung des Programms - das $f(\pi_2)$ immer noch nicht enthält - läßt sich dann q := f(„ π_2 ohne den String $f(\pi_2)$") errechnen. Mit der Zahl q als Startwert für X wird π_2 nur seinen Text ohne die Zahl q reproduzieren können. Diesen Mißstand beseitigen wir, indem wir den Algorithmus von π_2 abändern:

Es läßt sich leicht feststellen, nach dem wievielten Schritt des Algorithmus die Zahl $f(\pi_2)$ ausgegeben werden muß. Es sei dies der r-te Iterationsschritt. Die Zahl r wird Bestandteil des Programms. Der Algorithmus von π_2 lautet dann:

```
X:=q;
Y:=1;
while Y<=1(„ π2 ohne den String q") do
begin I:=F;
      OUTCHAR(C[I]);
      if Y=r then OUTINT(q,...);
      Y:=Y+1
  end
```

und das Gesamtprogramm ist

```
π2 = begin
     integer I,X,Y;
     character array C[1:maxchar];
     integer procedure F;
       begin F:=X mod(maxchar+1);
             X:=X//(maxchar+1)
         end;
     C[Ø]:="A";
          :
          :
     C[maxchar]:="*";
     X:=q;
     Y:=1;
     while Y<=l(„π2 ohne den String q") do
       begin I:=F;
             OUTCHAR(C[I]);
             if Y=r then OUTINT(q,...);
             Y:=Y+1
         end
     end
```

Die Tabelle C braucht natürlich nur die Zeichen enthalten, die auch wirklich in π_2 vorkommen. Dementsprechend kann die Zahl maxchar möglichst klein gehalten werden. Die Zahlen r,q und l(„π_2 ohne den String q") lassen sich ermitteln, nachdem das übrige Programm erstellt wurde. Es ist leicht einzusehen, daß gilt:

π_2 reproduziert sich selbst.

Das Programm π_2 ist zwar ein syntaktisch richtiges Programm, aber dennoch nicht auf Rechenmaschinen realisierbar. Das liegt an der Größenordnung der Zahl q. q liegt bei weitem außerhalb des darstellbaren Zahlenbereichs üblicher Rechenmaschinen. Um dies einzusehen, schätzen wir die Zahl q nach unten ab.

Wie man leicht feststellt, kommen im Programm π_2 mindestens 32 verschiedene Zeichen vor. Damit gilt maxchar$\geq$32.

Für die Länge von π_2 gilt, wenn man nur die unbedingt nötigen blanks, die als Trennzeichen fungieren, mitrechnet:

$$l(\text{„}\pi_2 \text{ ohne die Zahl q"}) > 700$$

Aus Definition (3.2.4.1.1) und der Definition von q folgt damit: $q > 32^{700-1}$

Trotz dieser sehr groben Abschätzung von q zeigt sich, daß q in herkömmlichen Rechnern nicht darstellbar ist.

3.2.4.2.Eine Iterationsfunktion basierend auf der Gödelisierung g aus 2.5.

In Abschnitt 2.5. wurde die Gödelisierung $g : C^* \longrightarrow \mathbb{N}_o$ eingeführt. Auf völlig analoge Weise kann man eine Gödelisierung $g_D : D^* \longrightarrow \mathbb{N}_o$ konstruieren, indem man C durch D und die Abbildung $H : C \longrightarrow \mathbb{N}_o$ durch die Abbildung $H_D : D \longrightarrow \mathbb{N}_o$ mit $H_D(\alpha) := i_\alpha$ für alle $\alpha \in D$ ersetzt. Wie man aus jeder Gödelnummer g(w), $w \in C^*$, effektiv das Urbild w bestimmen kann, so kann man das gleiche auch für jede Gödelnummer $g_D(v)$, $v \in D^*$, durchführen.

Damit läßt sich wie in 3.2.4.1. eine Prozedur F' entwickeln, mit deren Hilfe es möglich ist, $\delta(\pi'_2)$ iterativ zu berechnen, wenn $q' := g_D(\text{„}\pi'_2 \text{ ohne die Zahl } g_D(\pi'_2)")$ als Startwert für die Iteration gewählt wird. Wir erhalten dann ein ähnliches selbstreproduzierendes Programm π'_2 wie in 3.2.4.1.. Auch dieses Programm ist syntaktisch korrekt, aber ebensowenig realisierbar wie das Programm aus 3.2.4.1. Diese Tatsache liegt an der Nichtdarstellbarkeit der Zahl q'. Wir schätzen q' grob nach unten ab.

Die Länge von π'_2 beträgt mindestens 650 Zeichen. Dann ist

$$q > p_1^{i_B} \cdot p_2^{i_E} \cdot p_3^{i_G} \cdot p_4^{i_I} \cdot p_5^{i_N} \cdot \ldots \cdot p_{588}^{i_\alpha+1} \div 1 \quad , \alpha \in D$$

Da schon die fünfte Primzahl, nämlich 11, größer als 10 ist und das Zeichen a mit $i_a=0$ nur wenig in π'_2 auftritt, gilt sicherlich:

$$q' > 10^{600}$$

Damit ist q' ebenso wie q nicht in üblichen Rechenanlagen darstellbar.

3.2.5. Ein textgesteuertes SIMULA-Programm π_3

In Abschnitt 3.2.4. wurden selbstreproduzierende Programme π_2 und π'_2 entworfen, die zwar syntaktisch richtig waren, sich aber nicht auf konkreten Rechenanlagen realisieren ließen. Wir wollen keine weiteren Anstrengungen unternehmen, realisierbare Iterationsfunktionen und Startwerte zu finden, sondern ändern vielmehr Textzerlegung und Algorithmus der Programme aus 3.2.4. ab. Wir gewinnen aus π_2 das Programm π_3, indem wir folgende Änderungen vornehmen:

(i) Zerlegung: Im Gegensatz zu π_2 wird π_3 nicht in einzelne Zeichen, sondern in größere Teilstrings zerlegt.
Aus character array C[1:maxchar] wird
text array C[1:maxtext] .
Damit findet erstmals das SIMULA-Textkonzept in unseren Überlegungen seine Anwendung.

(ii) Algorithmus: Die Aufgabe des Algorithmus von π_3 besteht darin, den Programmtext π_3 aus Teilstrings, die in dem Feld C gespeichert sind, zusammenzusetzen. Jede Feldkomponente wird durch ihren Index kodiert. Der Text π_3 läßt sich dann als Folge von Indizes kodieren:

$$\pi_3 \longmapsto i_1,\ldots,i_k \quad , \quad i_j \in \{1,\ldots,\text{maxtext}\}$$

Diese Folge von Indizes schreiben wir als Text in die text-Variable X. Mittels der text-Prozeduren SUB und GETINT[1] ist der Zugriff auf die einzelnen Zahlen i_j im Text X gewährleistet. Der Algorithmus von π_3 braucht also nur noch den Text X sequentiell zu durchlaufen und für jedes i_j aus X den Text $C[i_j]$ auszugeben.

Durch eine derartige Ausnutzung des Text-Konzepts der Programmiersprache SIMULA umgehen wir die Repräsentation von π_3 durch eine ganze Zahl, wie dies in π_2 und π'_2 der Fall war, und damit auch die nicht mehr darstellbaren Startwerte q bzw. q' .

---

1) Standard-Funktion siehe [19].

Der Text X enthält jedoch nicht seine eigene Kodierung. Aus diesem Grund muß das Ausdrucken des Textes X im Algorithmus von π_3 eine Sonderstellung einnehmen. In Analogie zu 3.2.4.1. - dort machte die Ausgabe der Zahl X ähnliche Schwierigkeiten - wird die Ausgabe von X durch die einfach zu ermittelnde Zahl r gesteuert:

```
X:-COPY("i1,...,ik");
for I:=1 step 1 until k do
begin
   ⟨ermittle nächstes ij aus X⟩;
   OUTTEXT(C[ij]);
   if I=r then OUTTEXT(X)
end;
```

π_3 =

```
 1) begin integer I,S,Z; text X;
 2)       text array C[1:34];
 3) C[1]:-COPY("BEGIN INTEGER I,S,Z; TEXT X;");
 4) C[2]:-COPY("TEXT ARRAY C[1:34];");
 5) C[3]:-COPY("X:-COPY(""");
 6) C[4]:-COPY("FOR I:=1 STEP 1 UNTIL 105 DO ");
 7) C[11]:-COPY("BEGIN S:=X.SUB(Z+1,2).GETINT;");
 8) C[12]:-COPY("OUTTEXT(C[S]);");
 9) C[13]:-COPY("Z:=(IF S<10 THEN 2 ELSE 3)+Z;");
10) C[14]:-COPY("IF I=99 THEN OUTTEXT(X) END END");
11) C[21]:-COPY("C[");
12) C[22]:-COPY("]:-COPY(""");
13) C[23]:-COPY(""");");
14) C[24]:-COPY("""");
15) C[31]:-COPY("1");
16) C[32]:-COPY("2");
17) C[33]:-COPY("3");
18) C[34]:-COPY("4");
19) X:-COPY("1,2,
              21,   31,22,    1,   23,
              21,   32,22,    2,   23,
              21,   33,22,    3,24,23,
```

Die blanks sind nur zur besseren Gliederung eingefügt. Der Algorithmus beachtet sie nicht.

```
21,   34,22,    4,   23,
21,31,31,22,   11,   23,
21,31,32,22,   12,   23,
21,31,33,22,   13,   23,
21,31,34,22,   14,   23,
21,32,31,22,   21,   23,
21,32,32,22,   22,24,23,
21,32,33,22,24,23,   23,
21,32,34,22,24,24,   23,
21,33,31,22,   31,   23,
21,33,32,22,   32,   23,
21,33,33,22,   33,   23,
21,33,34,22,   34,   23,
              3,23,4,11,12,13,14,");
20) for I:=1 step 1 until 105 do
21) begin S:=X.SUB(Z+1,2).GETINT;          {Bewirkt das
22)       OUTTEXT(C[S]);                        scannen
23)       Z:=(if S<10 then 2 else 3)+Z;    {des Textes X
24)       if I=99 then OUTTEXT(X)   end  end
```

Verifikation von π_3

Der algorithmische Teil des Programms arbeitet sequentiell eine Folge von Zahlen ab. Diese Zahlen sind in dem Text X gespeichert. Es sind genau 105 Zahlen. Jede Zahl j bewirkt das Ausdrucken eines Textes C[j]. Zunächst werden die Texte C[1] und C[2] ausgedruckt. Damit sind die ersten beiden Programmzeilen kopiert. Mit den folgenden 96 Zahlen werden die Programmzeilen 3 bis 18 ausgedruckt. Diese 96 Zahlen bestehen aus 16 Gruppen. Jede Gruppe ist durch das Zahlenpaar 21,...,23 begrenzt und druckt genau eine Programmzeile aus. Jede dieser Gruppen hat folgenden allgemeinen Aufbau:

21,	≙	C[
[Zahl,]	≙	Ziffer 1 bzw. 2 bzw. 3
Zahl,	≙	Ziffer 1 bzw. 2 bzw. 3 bzw. 4
22,	≙	]:-copy("
[24,]	≙	"
Zahl,	≙	text
[24,]	≙	"

23, ≙ ");

Damit entspricht der Aufbau der Zahlengruppen genau dem allgemeinen Aufbau einer der Programmzeilen 3 bis 18. Berücksichtigt man die im Programm vorkommende Kodierung, so ist klar, daß diese Zahlengruppen die Programmzeilen 3 bis 18 ausdrucken.

Nach diesen 96 Zahlen wird die Zahl 3 abgearbeitet. Dadurch wird das Drucken von x:-copy(" bewirkt. Gleichzeitig hat die Laufvariable I der for-Schleife nun den Wert 99 (99 Zahlen sind ja abgearbeitet). Deshalb wird nun der Text X gedruckt. Die Abarbeitung der Zahl 23 schließt Programmzeile 19 ab. Die restlichen Programmzeilen 20 bis 24 werden durch Abarbeitung der restlichen Zahlen 4,11,12, 13 und 14 kopiert. Die Laufvariable hat dann den Wert 105, und der Algorithmus bricht ab.

Verbesserung von π_3

(i) Die Textvariablen C[1]...C[33] und X enthalten die Teilstrings des Programms. Besonders wichtig sind dabei die mehrfach auftretenden Teilstrings. Es sind dies:

C[21] = C[
C[22] =]:-copy("
C[23] = ");
C[24] = "
C[31] = 1
C[32] = 2
C[33] = 3
C[34] = 4

Diese Strings stellen in ihrer Gesamtheit die „Bauelemente" dar, aus denen das Gerippe von π_3 aufgebaut ist. Auf sie kann nicht verzichtet werden. Bei den anderen Teilstrings ist eigentlich nicht einzusehen, warum sie gerade so aufgeteilt sind. So könnten zum Beispiel C[1] und C[2] zusammengelegt werden. Grundsätzlich ist zu sagen, daß maximale Teilstrings gebildet werden können, die kein " enthalten. Der String xxx"xxx müßte z.B. als

k) C[j]:-copy("xxx""xxx");

vereinbart werden. Die Zahl j in dem Text X bewerkstelligt

dann zwar das Ausdrucken von xxx"xxx , aber es läßt sich keine Zahlenfolge finden, die die Zeile k) druckt:

21,.....,22,j,23,

bewirkt

```
C[j]:-copy("xxx"xxx");
```

Hier fehlt ein ". Es kann nicht durch Ausgabe des Textes C[24] eingefügt werden, da es mitten im Text von C[j] fehlt.

Das Einfügen der Zahl 24 in X kann nur dann zum Erfolg führen, wenn das Hochkomma am Anfang oder am Ende von C[j] steht. Vergleiche dazu die Programmzeilen 12),13) und 14) und die dazugehörenden Zahlengruppen im Text X.

(ii) Das obige Programm arbeitet mit einem text array C und einem text X. Sowohl die Komponenten von C als auch X enthalten nur Textkonstanten. Eine Sonderstellung von X ist also nicht aufrechtzuerhalten. Die Konsequenz: Wir erweitern das Feld C um eine Komponente C[x]. C[x] bekommt den Wert von X zugewiesen. Damit wird auch die algorithmische Sonderstellung des Textes X aufgegeben. Die if-Anweisung in der for-Schleife entfällt. Der ehemalige Text X, der jetzt in C[x] steht, wird einfach durch Abarbeitung der Zahl x ausgedruckt. x ist dabei selbst Element von C[x].

(iii) Die Punkte (i) und (ii) deuten schon an, daß sich viel Programmtext und Komponenten von C einsparen lassen. Weniger Komponenten bedeutet aber, daß wir mit weniger Ziffern auskommen, um die Komponenten zu adressieren. Möglicherweise kann die Programmzeile 18 weggelassen werden, da die Ziffer 4 gar nicht zur Adressierung benötigt wird.

Führt man die Verbesserungen (i) bis (iii) an Programm π_3 konsequent durch, so erhält man das folgende Programm π'_3. Die Feldkomponenten C[1] und C[31] zeigen insbesondere

sehr schön die Bildung „maximaler" Texte (vgl.(i)).

π'_3 =

```
begin integer I,S,Z; text array C[1:31];
C[1]:-COPY("BEGIN INTEGER I,S,Z; TEXT ARRAY C[1:31];C[1]:-C
          OPY("""");
C[2]:-COPY("C[");
C[3]:-COPY("]:-COPY("""");
C[11]:-COPY("""");");
C[12]:-COPY("""""");
C[13]:-COPY("1");
C[21]:-COPY("2");
C[22]:-COPY("3");
C[23]:-COPY("1,1,12,11,
            2,  21,3,   2,  11,
            2,  22,3,   3,12,11,
            2,13,13,3,12,11,  11,
            2,13,21,3,12,12,  11,
            2,13,22,3,  13,   11,
            2,21,13,3,  21,   11,
            2,21,21,3,  22,   11,
            2,21,22,3,  23,   11,
            2,22,13,3,  31,   11,31,");
C[31]:-COPY("FOR I:=1 STEP 1 UNTIL 60 DO BEGIN S:=C[23].SUB
          (Z+1,2).GETINT;OUTTEXT(C[S]);Z:=(IF S<10 THEN
          2 ELSE 3)+Z END END");
for I:=1 step 1 until 60 do
begin S:=C[23].SUB(Z+1,2).GETINT;
      OUTTEXT(C[S]);
      Z:=(if S<10 then 2 else 3)+Z
  end
end
```

Diese Version von π_3 ist in ihrer logischen Funktionalität nicht mehr zu verbessern. Strebt man aber „textuell" kurze Programme an, so gibt es noch eine weitere Verbesserungsmöglichkeit:

(iv) Die Komponenten von C können so adressiert werden, daß die in C[23] häufig auftretenden Adressen möglichst kurz sind.

Adresse	Auftreten in C[23]	Zeichen insgesamt
1	2	1× 2= 2
2	10	1×10=10
3	10	1×10=10
11	10	2×10=20
12	4	2× 4= 8
13	6	2× 6=12
21	8	2× 8=16
22	5	2× 5=10
23	1	2× 1= 2
31	2	2× 2= 4

Um „optimal" zu adressieren, müssen wir erreichen, daß die 3 einstelligen Adressen am häufigsten auftreten. Wie die Tabelle zeigt, ist das bisher nicht der Fall. Adresse 1 tritt nur 2-mal auf, während die zweistellige Adresse 11 10-mal auftritt. Wir erreichen eine „optimale" Adressierung, wenn wir die Inhalte von C[1] und C[11] vertauschen und den Inhalt von C[23] entsprechend korrigieren. Ersparnis: 8 Zeichen.

3.2.6. Implementierung des Programms π_3

Das Programm π_3 gibt seinen Programmtext über die Standarddatei SYSOUT aus. Da diese Ausgabe in Form eines einzigen Strings ohne jede Blockung erfolgt, reicht die voreingestellte Pufferlänge von SYSOUT nicht aus. Die Pufferlänge muß im Programm π_3 erhöht werden, damit keine Laufzeitfehler auftreten. π_3 wird daher um die Anweisung

```
SYSOUT.IMAGE:-BLANKS(200);
```

ergänzt. Entsprechend wird die Textkonstante C[31] erweitert. Das resultierende Programm π'_3 zeigt Anhang A.1..

Der zur Verfügung stehende SIMULA-Compiler hat die Eingabelänge 72. Die Ausgabe von π_3 bzw. π'_3 ließe sich nur dann kompilieren, wenn sie in Blöcke zu jeweils höchstens 72 Zeichen unterteilt wäre. Dies ist aber weder bei π_3 noch bei π'_3 der Fall. Anhang A.2. zeigt eine Version

π''_3 von π_3, deren Ausgabe in Zeilen a 72 Zeichen unterteilt ist. Dies wird durch einen komplizierten Anweisungsteil erreicht. Die Ausgabe von π''_3 läßt sich erneut übersetzen. Sie stellt ein lauffähiges SIMULA-Programm dar, das gleich π''_3 ist.

3.2.7. Ein prozedurgesteuertes Programm π_4

In 3.2.5. wurde ein selbstreproduzierendes Programm π_3 konstruiert, das seinen Text in Teilstrings zerlegt enthielt. Diese Teilstrings brauchten von π_3 nur noch in der richtigen Reihenfolge ausgedruckt zu werden. In Form von Programm π_4 lernen wir nun ein selbstreproduzierendes SIMULA-Programm kennen, das die Abspeicherung seiner Teilstrings direkt mit der Ausgabe dieser Strings koppelt.

Statt C[Adresse]:-copy("text"); in π_3

schreiben wir procedure name;OUTTEXT("text"); in π_4

Die Zerlegung des Programmtextes von π_4 entspricht dabei der Zerlegung des Programmtextes von π_3. Der Algorithmus von π_4 besteht nur noch aus einer Folge von Prozeduraufrufen. Mit Programm π_4 lösen wir uns wieder vom eigentlichen SIMULA-Textkonzept. Wir benötigen lediglich die Möglichkeit, Textkonstanten als Argumente für Ausgabeanweisungen zu benutzen.

π_4 =

```
1) begin
2) procedure AA;OUTTEXT("BEGIN ");
3) procedure C;OUTTEXT("PROCEDURE ");
4) procedure A;OUTTEXT(";OUTTEXT("""");
5) procedure B;OUTTEXT("""");");
6) procedure AC;OUTTEXT("""""");
7) procedure BA;OUTTEXT("A");
8) procedure BB;OUTTEXT("B");
9) procedure BC;OUTTEXT("C");
10) procedure AB;OUTTEXT("AA;C;BA;BA;A;AA;B;C;BC;A;C;B;C;BA
```

```
        ;A;A;AC;B;C;BB;A;AC;B;B;C;BA;BC;A;AC;AC;B;C;BB;BA;A;
        BA;B;C;BB;BB;A;BB;B;C;BB;BC;A;BC;B;C;BA;BB;A;AB;B;AB
         END");
11) AA;
12) C;BA;BA;A;    AA;    B;
13) C;BC;    A;     C;    B;
14) C;BA;    A;     A;AC;B;
15) C;BB;    A;AC;  B;    B;
16) C;BA;BC;A;AC;AC;    B;
17) C;BB;BA;A;    BA;    B;
18) C;BB;BB;A;    BB;    B;
19) C;BB;BC;A;    BC;    B;
20) C;BA;BB;A;    AB;    B;
21) AB
22) end
```

Verifikation von π_4

AA; ist das erste statement des Anweisungsteils von π_4. Es bewirkt die Ausgabe der ersten Programmzeile. Die nächsten 9 Programmzeilen werden durch die Prozeduraufrufe der 9 Programmzeilen 12) bis 20) ausgegeben, was sich mit Hilfe der tabellarischen Schreibweise des Anweisungsteils von π_4 leicht nachvollziehen läßt. Das folgende und gleichzeitig letzte statement ist ein Aufruf der Prozedur AB. Dieser Aufruf bewirkt die Ausgabe der restlichen Programmzeilen 11) bis 22), da die Textkonstante der Prozedur AB den algorithmischen Teil von π_4 enthält. Durch Ausführung der Prozedur AB holt die Ausgabe des Programms π_4 die Ausführung von π_4 ein.

(3.2.7.1) Bemerkung: Das selbstreproduzierende SIMULA-Programm π_4 benötigt als Daten nur Textkonstanten. Zur Strukturierung verwendet π_4 neben der Hintereinanderausführung von Anweisungen nur das Prozedurkonzept. Insgesamt gesehen verwendet π_4 nur Elemente, die die meisten höheren Programmiersprachen zur Verfügung stellen. Daher gesehen müssen dem

Programm π_4 ähnelnde selbstreproduzierende Programme in fast allen höheren Programmiersprachen existieren.

3.2.8. Implementierung des Programms π_4

Für die Implementierung des Programms π_4 gelten die gleichen Bemerkungen, die zur Implementierung von π_3 in 3.2.6. gemacht wurden. Aus π_4 läßt sich mit geringem Aufwand ein lauffähiges selbstreproduzierendes Programm π'_4 gewinnen, indem die Anweisung

```
SYSOUT.IMAGE:-BLANKS(200);
```

in den Anweisungsteil von π_4 bzw. in die Textkonstante der Prozedur AB eingefügt wird.

Aus π_4 läßt sich wie in 3.2.6. ein selbstreproduzierendes Programm π''_4 ableiten, dessen Ausgabe so formatiert ist, daß ein lauffähiges Programm entsteht. Erreicht wird dies durch

- Einführung der Prozedur
  `procedure Q;OUTIMAGE;`
- Aufspalten der Textkonstanten der Prozedur AB auf die Prozeduren
  AB,CA,CB,CC,AAA und AAB

Die Ausgabe der zusätzlichen Prozeduren CA bis AAB bewirkt einen vergrößerten Anweisungsteil in π''_4.

Anhang A.3. und Anhang A.4. demonstrieren die aus π_4 resultierenden Programme π'_4 bzw. π''_4.

3.3. Selbstreproduzierende Programme in PASCAL [1)]

In diesem Abschnitt sollen selbstreproduzierende Programme in der Programmiersprache PASCAL vorgestellt werden. PASCAL ist neben der Tatsache, daß es nicht blockorientiert ist, eine Programmiersprache, die keine Textvariablen kennt; PASCAL sieht nur Textkonstanten vor. Von daher gesehen ist

1)PASCAL-Beschreibung in [10].

es nicht ohne weiteres möglich, aus dem SIMULA-Programm π_3 aus 3.2.5. ein entsprechendes selbstreproduzierendes PASCAL-Programm zu gewinnen. Auf Grund von Bemerkung (3.2.7.1) wird es jedoch keine Schwierigkeiten bereiten, das Programm π_4 aus 3.2.6. nach PASCAL zu übertragen.

3.3.1. Ein textgesteuertes PASCAL-Programm π_5

Wir versuchen trotz des Fehlens von Textvariablen in PASCAL, das Programm π_3 in ein selbstreproduzierendes PASCAL-Programm π_5 zu übertragen. Dazu gibt es verschiedene Möglichkeiten, von denen zwei genannt sein sollen:

(i) Wir simulieren die in π_3 vorkommenden Texte durch character arrays. Auf diese Weise wird das Feld C zweidimensional

```
var C : array [1..maxtext,1..maxlength] of char
```

wobei maxlength gleich der Länge des längsten Teilstrings ist, in die wir das PASCAL-Programm π_5 zerlegen. Jede Zeile von C beinhaltet genau einen String der Zerlegung von π_5.

Nachdem die textuelle Speicherung der Zerlegung geklärt ist, können wir uns dem algorithmischen Teil von π_5 zuwenden. Da keine Texte und somit auch keine Prozedur GETINT zur Verfügung stehen, behelfen wir uns wie folgt:

Wir kodieren jeden „Text" C[j,...] durch einen Buchstaben des Alphabets in der folgenden Weise:

```
C[1,...] ⟼ a
C[2,...] ⟼ b
C[3,...] ⟼ c
C[4,...] ⟼ d
    ⋮        ⋮
C[maxtext,..] ⟼ maxtext-terBuchstabe.
```

Der Programmtext π_5 läßt sich mittels der Zeilen von C zusammensetzen und daher auf eindeutige Weise durch eine endliche Folge von Buchstaben beschreiben. Diese

Buchstabenfolge ist der Inhalt von C[maxtext,...].
Der Algorithmus braucht dann nur noch diese Buchstabenfolge in eine Folge von Druckanweisungen umzusetzen:

```
for I:=1 to ⟨Länge von C[maxtext,..]⟩do
begin case C[maxtext,I] of
      'a' : ⟨Ausgabe von C[1,..]⟩
      'b' : ⟨Ausgabe von C[2,..]⟩
            .
            .
            .
  end
```

Leider treten auf den linken Seiten der case-Alternativen viele Hochkommata ' auf. Das Zeichen ' spielt in PASCAL die gleiche Rolle wie das Zeichen " in der Programmiersprache SIMULA. Der Algorithmus müßte also als Text in sehr viele Teilstrings zerlegt werden (vgl. 3.2.5.), was zu einem unüberschaubaren Programm führen würde. Einen Ausweg bietet die Transferfunktion ORD von char nach integer:

```
for I:=1 to ⟨Länge von C[maxtext,...]⟩ do
begin HELP:=ORD(C[maxtext,I])
      case HELP of
      ⟨ORD(a)⟩  :  ⟨Ausgabe von C[1,..]⟩
      ⟨ORD(b)⟩  :  ⟨Ausgabe von C[2,..]⟩
                .
                .
                .
  end
```

Die Realisierung von π_5 nach der bisher entworfenen Methode enthält noch einige Schwierigkeiten. Z.B.:

- Es müßte eigens eine Ausgabeprozedur für die statements vom Typ ⟨Ausgabe C[I,...]⟩ in π_5 enthalten sein.
- Die Zeilen von C sind in der Regel mit blank-Zeichen aufgefüllt. Die Ausgabe dieser blanks ist zu vermeiden.

Insgesamt würde ein durchaus korrektes, aber auch unübersichtliches Programm π_5 entstehen.

(ii) Um die in (i) entstandenen Schwierigkeiten zu vermeiden, speichern wir die Teilstrings von π_5 nicht in einem zweidimensionalen character array, sondern wir verwenden die implizite Speicherung mittels Ausgabeprozeduren wie im SIMULA-Programm π_4 aus 3.2.7.

Der Algorithmus von π_5 bleibt der gleiche wie der Algorithmus unter (i), wenn man statt der Zeilen von C nur die Ausgabeprozeduren durch Buchstaben kodiert.

Bei der Realisierung von π_5 durch Alternative (ii) bleibt das Programm überschaubar.

Mit

ORD(a) = 193
ORD(b) = 194
ORD(c) = 195
ORD(d) = 196
ORD(e) = 197
ORD(f) = 198
ORD(g) = 199
ORD(h) = 200
ORD(i) = 201
ORD(j) = 202
ORD(k) = 203

} bezogen auf die zur Verfügung stehende PASCAL-Implementierung

und der Kodierung

procedure A ⟼ a
procedure B ⟼ b
procedure C ⟼ c
procedure AA ⟼ d
procedure AB ⟼ e
procedure AC ⟼ f
procedure BA ⟼ g
procedure BB ⟼ h
procedure BC ⟼ i
procedure CA ⟼ j
procedure CB ⟼ k

ergibt sich:

π_5 =

```
program SELF(OUTPUT);
  var I,HELP : integer;
           X : array [1..72] of char;
  procedure A; begin WRITE('PROGRAM SELF(OUTPUT);VAR I,HELP
      : INTEGER; X : ARRAY [1..72] OF CHAR;') end;
  procedure B; begin WRITE('''';FOR I:=1 TO 72 DO BEGIN HELP
     :=ORD(X [I] ); CASE HELP OF 193:A;194:B;195:C;196:AA;197
     :AB;198:AC;199:BA;200:BB;201:BC;202:CA;203:CB; END; EN
     D; WRITELN END.') end;
  procedure C;begin WRITE('PROCEDURE ') end;
  procedure AA;begin WRITE(';BEGIN WRITE(''''') end;
  procedure AB;begin WRITE('''') END;') end;
  procedure AC;begin WRITE('''''') end;
  procedure BA;begin WRITE('A') end;
  procedure BB;begin WRITE('B') end;
  procedure BC;begin WRITE('C') end;
  procedure CA;begin WRITE('BEGIN X:=''') end;
  procedure CB;begin WRITE('ACGDAECHDFBECIDCECGGDDFECGHDFEE
     CGIDFFECHGDGECHHDHECHIDIECIGDJFECIHDKEJKB') end;
begin
X:='ACGDAECHDFBECIDCECGGDDFECGHDFEECGIDFFECHGDGECHHDHECHIDIE
   CIGDJFECIHDKEJKB';
for I:=1 to 72 do
begin HELP:=ORD(X[I]);
      case HELP of
      193 : A;
      194 : B;
      195 : C;
      196 : AA;
      197 : AB;
      198 : AC;
      199 : BA;
      200 : BB;
      201 : BC;
      202 : CA;
      203 : CB;
      end;
end;WRITELN
end.
```

Verifikation von π_5

Die Bedeutung der for-Schleife von Programm π_5 wurde oben erläutert. Für jeden Buchstaben der Textkonstanten X wird eine Alternative der case-Anweisung ausgeführt und somit ein String der Zerlegung des Programmtextes von π_5 ausgegeben. Einfaches Nachvollzieher der Abarbeitung von X bestätigt, daß π_5 sich selbst reproduziert (vgl. Verifikation von π_3).

3.3.2.Implementierung des Programms π_5

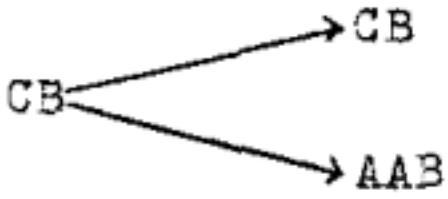
π_5 wird so implementiert, daß die Ausgabe von π_5 in Zeilen zu höchstens 132 Zeichen formatiert ist. Daher wird zunächst die Prozedur

```
procedure Q; begin WRITELN end;
```

in π_5 eingefügt.

Die Prozeduren A und B werden wegen ihrer relativ langen Textkonstanten in mehrere Prozeduren aufgespalten:

Die zusätzlichen Prozeduren bewirken eine Verlängerung der Kodierung von π_5. Dadurch wird eine Aufspaltung der Prozedur CB ebenfalls notwendig:



In π_5 enthält die Variable X die Kodierung des Programms. Die neu hinzugekommenen Prozeduren verursachen eine solche Zunahme der Kodierung, daß eine Variable (bedingt durch die vorhandene PASCAL-Implementierung) nicht mehr ausreicht, um die Kodierung aufzunehmen. Es wird neben X die Variable

```
Y : array [1..68] of char
```

zur Aufnahme der Kodierung von π_5 notwendig, was die Einführung der Prozedur CCA bewirkt. Der Algorithmus wird entsprechend geändert. Anhang A.5. zeigt das so veränderte Programm π_5 im einzelnen. Die

neuen Prozeduren sind dort wie folgt kodiert:

CC	⟶ l	AAB	⟶ n
AAA	⟶ m	CCA	⟶ q
AAC	⟶ p	Q	⟶ o

mit

ORD(l) = 211	ORD(o) = 214
ORD(m) = 212	ORD(p) = 215
ORD(n) = 213	ORD(q) = 216

3.3.3.Ein prozedurgesteuertes PASCAL-Programm π_6

Alle in Programm π_4 aus Abschnitt 3.2.7. verwendeten Sprachelemente finden sich auch in der Programmiersprache PASCAL. Damit läßt sich π_4 direkt in ein selbstreproduzierendes PASCAL-Programm π_6 übersetzen.

π_6 =

```
program PI6(OUTPUT);
procedure AA; begin WRITE('PROGRAM PI6(OUTPUT);PROCEDURE AA
   ; BEGIN WRITE(''') end;
procedure C; begin WRITE('PROCEDURE ') end;
procedure A; begin WRITE(';BEGIN WRITE(''') end;
procedure B; begin WRITE(''') END;') end;
procedure AC; begin WRITE('''') end;
procedure BA; begin WRITE('A') end;
procedure BB; begin WRITE('B') end;
procedure BC; begin WRITE('C') end;
procedure AB; begin WRITE('BEGIN AA;AA;AC;B;C;BC;A;C;B;C;BA
   ;A;A;AC;B;C;BB;A;AC;B;B;C;BA;BC;A;AC;AC;B;C;BB;BA;A;BA;B
   ;C;BB;BB;A;BB;B;C;BB;BC;A;BC;B;C;BA;BB;A;AB;B;AB;WRITELN
    END.') end;
begin AA;                AA;AC;B;
        C;BC;    A;     C;    B;
        C;BA;    A;     A;AC;B;
        C;BB;    A;AC;  B;    B;
        C;BA;BC;A;AC;AC;     B;
        C;BB;BA;A;    BA;    B;
        C;BB;BB;A;    BB;    B;
        C;BB;BC;A;    BC;    B;
        C;BA;BB;A;    AB;    B;AB;WRITELN end.
```

Verifikation von π_6

Die Verifikation von π_6 ergibt sich direkt aus der Verifikation von π_4 in 3.2.7.

3.3.4. Implementierung des Programms π_6

Programm π_6 schreibt seinen Text hintereinander ohne Blockung auf die Ausgabedatei OUTPUT. Das Ergebnis von π_6 ist ein einziger String. Dieser String ist sowohl für den Puffer des SIEMENS-Schnelldruckers, als auch für den Puffer des PASCAL-Compilers zu lang. Der String kann also weder ausgedruckt noch erneut kompiliert werden.

Um ein sichtbares Ergebnis zu erhalten, benötigen wir eine Ausgabe, die in Blöcke von höchstens 132 (=Pufferlänge des Schnelldruckers) Zeichen unterteilt ist. Es muß also in das Programm π_6 wiederholt die Prozedur WRITELN eingefügt werden. Wir kürzen die Prozedur wie folgt ab:

```
procedure Q; begin WRITELN end;
```

Da mit dieser Prozedurvereinbarung der Vereinbarungsteil von π_6 größer wird, muß die Prozedur AA entsprechend abgeändert werden. Die lange Textkonstante von Prozedur AB wird auf zwei Prozeduren verteilt. Zu diesem Zweck muß eine weitere Prozedur CA in das Programm aufgenommen werden. Anhang A.6. protokolliert das so veränderte Programm π_6.

3.4. Selbstreproduzierende Programme in SIEMENS-Assembler

In diesem Abschnitt werden Beispiele für selbstreproduzierende Programme in einer Assembler-Sprache angegeben. Die Beispiele verwenden den SIEMENS-Assembler. Die Tatsache, daß Assembler-Programme in der Lage sind, den Speicherbereich, in dem sie sich befinden, zu adressieren und zu lesen, vereinfacht das Schreiben selbstreproduzierender Assembler-Programme bedeutend. Selbstreproduzierende Pro-

gramme in Assembler brauchen nicht ihren Programmtext ebenfalls in Assembler auszugeben, sondern können direkt ihren Maschinenkode im Arbeitsspeicher kopieren (vgl.1.2.).

Alle in den folgenden Beispielen vorkommenden Adressierungen beziehen sich auf den Programmzähler PCR und sind somit relativ. Dadurch wird gewährleistet, daß die Funktionen der Kopien die gleichen sind wie die der jeweiligen Ursprungsprogramme. Die Kopien sind daher ebenfalls selbstreproduzierend.

Die Beispielprogramme sind soweit erläutert, wie es der Rahmen dieser Arbeit zuläßt. Nähere Angaben bzgl. des SIEMENS-Assemblers entnehme man den Schriften [22] und [23].

(3.4.1) Beispiel: Das selbstreproduzierende Assembler-Programm PROG1 legt eine Kopie seines Maschinenkodes ab dem 64-ten auf den ersten Befehl von PROG1 folgenden Byte im Arbeitsspeicher an.

Zeilen-nummer	Name	mnemot. Op.-Kode	Operanden	Befehls-format	Befehlslänge (in Byte)
Ø1	PROG1	START			
Ø2		BALR	1,ØØ	RR	2
Ø3		LA	2,2(Ø,Ø)	RX	4
Ø4		SR	1,2	RR	2
Ø5		LM	4,8,Ø(1)	RS	4
Ø6		STM	4,8,64(1)	RS	4
Ø7		SVC	X'5B'	RR	2
Ø8		END			
			Programmlänge in Byte		18

Die beiden Assembler-Anweisungen START und END erzeugen keinen Maschinenkode und brauchen daher beim Kopierprozeß nicht berücksichtigt zu werden.

Der erste ausführbare Befehl des Programms ist

```
BALR    1,ØØ
```

Dieser Befehl lädt in das Mehrzweckregister R1 den aktuellen Wert des Befehlszählers PCR. Da der Befehlszähler vor Ausführung eines Befehls um die Länge des betreffenden Befehls hochgezählt wird, enthält Register R1 nach Ausführung des BALR-Befehls die Startadresse von PROG1 im Arbeitsspeicher plus der Länge des BALR-Befehls. Der BALR-Befehl hat das Format RR und somit die Länge 2. Der Befehl

```
LA      2,2(Ø,Ø)
```

stellt in Register R2 die Zahl 2 bereit. Der SR (Subtrahieren Register)-Befehl

```
SR      1,2
```

subtrahiert den Inhalt des Registers R2 vom Inhalt des Registers R1. Nach Ausführung dieses Befehls enthält demnach Register R1 genau die Startadresse des Programms. R1 wird nachfolgend als Basisregister verwendet. Der Befehl in Zeile Ø5 ist ein LM(Laden mehrfach)-Befehl

```
LM      4,8,Ø(1)
```

LM-Befehle können aufeinanderfolgende Mehrzweckregister - also höchstens 16 - mit aufeinanderfolgenden Worten aus dem Arbeitsspeicher laden. Die ersten beiden Operanden bezeichnen das erste und das letzte der benutzten Mehrzweckregister. Der letzte Operand stellt die Adresse des ersten der zu transferierenden Arbeitsspeicherworte dar. In unserem Fall ist diese Adresse die Startadresse von PROG1. Deshalb wird der dritte Operand aus der Distanzadresse Ø und dem Register R1 als Basisregister zusammengesetzt. Das Programm PROG1 umfaßt

18 Bytes. Da ein Arbeitsspeicherwort 4 Bytes umfaßt, genügen also die 5 Mehrzweckregister R4 bis R8, um das gesamte Programm zu laden. Der nächste Befehl

```
STM    4,8,64(1)
```

ist ein STM(Speichern mehrfach)-Befehl. Dieser Befehl ist das Gegenstück zum LM-Befehl und legt die Inhalte der Register R4 bis R8 beginnend bei der Adresse, die der dritte Operand angibt, hintereinander im Arbeitsspeicher ab. Der dritte Operand des STM-Befehls benutzt wieder das Register R1 als Basisregister, die Distanzadresse ist 64. Nach Ausführung des STM-Befehls liegt also die Kopie von PROG1 bereits im Arbeitsspeicher vor. Sie ist 64 Bytes vom Ursprungsprogramm entfernt. Der letzte Befehl

```
SVC    X'5B'
```

dient nur dazu, PROG1 ordnungsgemäß zu beenden. Ein ausführliches Protokoll von PROG1 befindet sich in Anhang B.1..

Das folgende Beispielprogramm PROG2 ist um 2 Bytes kürzer als PROG1. Erreicht wird dies durch Ersetzung des LM- und des STM-Befehls durch einen MVC(Übertragen Zeichenfolge)-Befehl. PROG2 ist in der Lage, außer sich selbst noch einen gewissen, auf das Programm folgenden Speicherbereich mitzukopieren.

(3.4.2.) Beispiel:

Zeilen-nummer	Name	mnemot. Op.-Kode	Operanden	Befehls-format	Befehlslänge (in Byte)
Ø1	PROG2	START			
Ø2		BALR	1,ØØ	RR	2
Ø3		LA	2,2(Ø,Ø)	RX	4

Ø4	SR	1,2	RR	2
Ø5	MVC	64(6Ø,1),Ø(1)	SS	6
Ø6	SVC	X'5B'	RR	2
Ø7	END			
		Programmlänge in Byte		16

Die Programmzeilen Ø1 bis Ø4 sind mit denen von PROG1 identisch. Sie bewirken, daß die Startadresse in das Register R1 geladen wird. Der MVC-Befehl in Zeile Ø5 bewirkt, daß beginnend bei der Adresse

- Inhalt des Basisregisters R1 plus Distanzadresse Ø } vgl. 2-ter Operand
- 6Ø aufeinanderfolgende Bytes des Arbeitsspeichers in den mit der Adresse
  - Inhalt des Basisregisters R1 plus Distanzadresse 64 } vgl. 1-ter Operand

beginnenden Arbeitsspeicherbereich geschrieben werden. Da PROG2 selbst nur 16 Bytes lang ist, werden durch den MVC-Befehl 44 zusätzliche Bytes mitkopiert. Mit einem MVC-Befehl lassen sich sogar maximal 2^8 Bytes transferieren, wenn die Operandenlänge entsprechend angegeben wird (im Beispiel: 6Ø).

Die Programmzeilen Ø6 und Ø7 entsprechen den Zeilen Ø7 und Ø8 in PROG1. Rechnerprotokoll siehe Anhang B.2..

Die Zeilen Ø2 bis Ø5 von PROG2 stellen einen Programmteil dar, durch den sich andere Assemblerprogrammabschnitte (oder ganze Programme) zu selbstreproduzierenden Programmen (bzw. Programmabschnitten) ergänzen lassen (siehe

Abb. 3.4.A). Die Länge des resultierenden Programmabschnitts (in Byte) wird als Operandenlänge für den MVC-Befehl verwendet. Auf diese Weise können Programmabschnitte (bzw. Programme) bis zu einer Länge von 2^8-16 Bytes im Arbeitsspeicher kopiert werden. Entsprechend der Länge des Programmabschnitts muß die Distanzadresse, die die Lage der Kopie bestimmt, gewählt werden.

```
Ø1    PROGRAM   CSECT
Ø2              BALR    1,ØØ
Ø3              LA      2,2(Ø,Ø)
Ø4              SR      1,2
Ø5              MVC     ⟨Distanzadresse der Kopie⟩(⟨Anzahl der zu kopierenden Bytes⟩,1),Ø(1)
Ø6              .
Ø7              .
                .
                .       } Der zu kopierende Programmabschnitt (maximal 2^8-16 Bytes)
                .
                .
                .
                END
```

Abb. 3.4.A

Das folgende Beispielprogramm PROG3 ist ein selbstreproduzierendes Assembler-Programm, das nach seiner Abarbeitung

die Kontrolle an die Kopie übergibt. Erreicht wird dies durch einen unbedingten Sprung zur Startadresse der Kopie. Da die Kopie das gleiche Verhalten zeigt wie PROG3, iteriert sich dieser Prozeß. Der zur Verfügung stehende Arbeitsspeicher wird also mit Kopien von PROG3 vollgeschrieben. Die einzelnen Kopien folgen mit konstantem Abstand aufeinander.

(3.4.3) Beispiel:

Zeilen nummer	Name	mnemot. Op.-Kode	Operanden	Befehls-format	Befehlslänge (in Byte)
Ø1	PROG3	START			
Ø2		BALR	1,ØØ	RR	2
Ø3		LA	2,2(Ø,Ø)	RX	4
Ø4		SR	1,2	RR	2
Ø5		MVC	64(6Ø,1),Ø(1)	SS	6
Ø6		LA	2,64(Ø,Ø)	RX	4
Ø7		AR	1,2	RR	2
Ø8		BR	1	RR	2
Ø9		END			
			Programmlänge in Byte		22

Die Zeilen Ø1 bis Ø5 sind mit denen von Programm PROG2 identisch und bewirken bereits das Kopieren von PROG3. Die Kopie wird beginnend beim 64-ten auf den ersten Befehl von PROG3 folgenden Byte im Arbeitsspeicher abgelegt. Der LA(Laden Adresse)-

Befehl in Zeile Ø6

```
LA    2,64(Ø,Ø)
```

stellt in Register R1 die Zahl 64 bereit. Der folgende AR(Addieren Register)-Befehl

```
AR    1,2
```

erhöht den Inhalt des Basisregisters R1 um 64. R1 enthält nach Abarbeitung des AR-Befehls die Startadresse der Kopie. Daher erfolgt durch den BR(Springen unbedingt)-Befehl

```
BR    1
```

ein Sprung zum ersten Befehl der Kopie und damit die Abarbeitung der Kopie. Die Kopie legt darauf eine erneute Kopie an u.s.w.

Anhang B.3. demonstriert PROG3. Da PROG3 eine nicht abbrechende Programmfolge erzeugt, kommt es wegen Speichererschöpfung zu einem Fehlerabbruch.

In den vorangegangenen Beispielprogrammen erfolgte das Kopieren der Programme en bloc mit Hilfe des MVC-Befehls bzw. des LM- und des STM-Befehls. Das folgende Beispiel zeigt ein Programm, das seinen Kode explizit in Abschnitten zu je 4 Bytes kopiert. Dieses Programm ist algorithmisch etwas aufwendiger und dementsprechend länger als die bisherigen Beispielprogramme dieses Abschnitts.

(3.4.4) Beispiel:

Zeilen-nummer	Name	mnemot. Op-Kode	Operanden	Befehls-format	Befehlslänge (in Byte)
Ø1	PROG4	START			

Ø2	BALR	1,ØØ	RR	2
Ø3	LA	2,2(Ø,Ø)	RX	4
Ø4	SR	1,2	RR	2
Ø5	LA	3,4(Ø,Ø)	RX	4
Ø6	LA	4,48(Ø,Ø)	RX	4
Ø7	LA	1Ø,22(Ø,Ø)	RX	4
Ø8	AR	1Ø,1	RR	2
Ø9	MVC	64(4,1),Ø(1)	SS	6
1Ø	AR	1,3	RR	2
11	SR	4,3	RR	2
12	BRP	1Ø	RR	2
13	SVC	X'5B'	RR	2
14	END			
		Programmlänge in Byte		36

Die Befehle der Programmzeilen Ø2 bis Ø4 bewirken, daß das Register R1 die Anfangsadresse von PROG4 im Arbeitsspeicher enthält. Register R1 wird fortan als Basisregister benutzt. Die Befehle der Zeilen Ø5 bis Ø7 stellen in den Mehrzweckregistern R3, R4 und R1Ø die Werte 4, 48 und 22 bereit. 48 ist die Gesamtzahl der Bytes, die das Programm PROG4 im Arbeitsspeicher kopiert. Der Befehl

```
AR      1Ø,1
```

erhöht den Inhalt des Registers R1Ø um die Startadresse des Programms PROG4. Nach Ausführung dieses AR-Befehls enthält Register R1Ø die Sprungadresse, zu der der BRP(Springen, falls positiv)-Befehl in Zeile 12 verzweigt. Es handelt sich dabei um die Adresse des MVC-Befehls

```
MVC     64(4,1),Ø(1)
```

in Programmzeile Ø9. Der MVC-Befehl kopiert die ersten 4 Bytes des Maschinenkodes von PROG4 im Arbeitsspeicher. Die Kopie wird 64 Bytes vom ersten Befehl von PROG4 entfernt angelegt. Der MVC-Befehl benutzt zur Adressierung das Basisregister R1. Der Inhalt von R1 wird im darauffolgenden Befehl

```
AR    1,3
```

um den Inhalt des Registers R3, also nur den Wert 4, erhöht. Der nächste Befehl

```
SR    4,3
```

subtrahiert vom Inhalt des Registers R4, der gleich der momentanen Anzahl der noch zu kopierenden Bytes ist, den Wert 4. Hat diese Subtraktion einen positiven Wert ergeben, so sind noch nicht alle der insgesamt 48 Bytes kopiert, und es wird mittels

```
BRP   1Ø    (s.o.)
```

zum MVC-Befehl in Zeile Ø9 zurückgesprungen. Der MVC-Befehl kopiert dann die nächsten 4 Bytes von PROG4, da das Basisregister R1 bereits um 4 Bytes erhöht worden ist. Das Programm bricht ab, nachdem alle 48 Bytes kopiert worden sind. Da PROG4 nur 36 Bytes lang ist, werden also 12 zusätzliche, sich an PROG4 anschließende Bytes mitkopeirt. Der Befehl

```
SVC   X'5B'
```

in Zeile 13 beendet PROG4.
Anhang B.4. demonstriert PROG4.

Entsprechend PROG2 in Beispiel (3.4.2) lassen sich durch die Zeilen Ø1 bis 12 größere Abschnitte anderer Assemblerprogramme (bzw. ganze Programme) zu selbstreproduzierenden Programmabschnitten (bzw. Programmen) ergänzen (siehe Abb.

3.4.B). Die Selbstreproduktion erfolgt jedoch nicht en bloc, sondern in Abschnitten zu je 4 Bytes. Die Länge des ergänzten Abschnitts (bzw. Programms) in Byte braucht dann nur durch den Befehl in Zeile Ø6 in Register R4 bereitgestellt zu werden. Entsprechend der Länge des zu kopierenden Programmabschnitts (bzw. Programms) muß die Distanzadresse, die die Lage der Kopie im Arbeitsspeicher bestimmt, in Zeile Ø9 (MVC-Befehl) gesetzt werden.

```
Ø1  PROGRAM  CSECT
Ø2           BALR   1,ØØ
Ø3           LA     2,2(Ø,Ø)
 :           :
Ø6           LA     4,⟨Länge des Gesamtabschnitts⟩(Ø,Ø)
Ø7           LA     1Ø,22(Ø,Ø)
 :           AR     1Ø,1
 :           MVC    ⟨Distanzadresse der Kopie⟩(4,1),Ø(1)
 :           :
 :           :
12           BRP    1Ø
13           :   )
 :           :   )
 :           :   ) Der zu kopierende Programmabschnitt
 :           :   )
 :           :   )
             END
```

Abb 3.4.B

Im Gegensatz zu der aus PROG2 abgeleiteten Methode zur Selbstreproduktion größerer Programmabschnitte liegt keine Begrenzung auf 2^8-16 Bytes vor.

4. Varianten zur Selbstreproduktion von Programmen

In diesem Kapitel sei S eine beliebige höhere Programmiersprache im üblichen Sinn (vgl. 1.2.).

4.1. Motivation

In Abschnitt 1.2. haben wir eine Definition selbstreproduzierender Programme angegeben. Diese Definition lautete sinngemäß:

Sei π aus S. π heißt selbstreproduzierend, wenn π ohne Benutzung von Eingabe seinen Programmtext in S ausgibt.

Betrachtet man diese Definition etwas differenzierter, so ergeben sich zwei Anforderungen an die Ausgabe eines selbstreproduzierenden Programms π aus S:

a) Die Ausgabe von π muß ein syntaktisch korrektes Programm π' aus der Programmiersprache S enthalten.

b) π' muß gleich π sein.

Läßt man die Forderung b) fallen, so ist das Programm i.a. nicht mehr selbstreproduzierend; es ist allenfalls als „reproduzierend " zu bezeichnen.

Sei nun π „reproduzierend", dann sind zum Beispiel folgende Möglichkeiten denkbar:

(i) Das Programm π gibt das Programm π' aus. π' seinerseits gibt das Programm π'' aus, und es gilt $\pi'' = \pi$. π und π' sind dann sicher für sich nicht selbstreproduzierend. Trotzdem liegt aber ein gewisser Selbstreproduktionsmechanismus mit „Zwischenstufe" vor.

(ii) Das Programm $\pi = \pi^0$ gibt das Programm π^1 aus, π^1 seinerseits das Programm π^2 u.s.w..
Allgemein: π^i gibt π^{i+1} aus, $i \geq 0$.
Für alle $i, j \geq 0$ gilt $\pi^i \neq \pi^j$, falls $i \neq j$.

Andererseits könnte man die obige Definition der Selbstreproduktion verschärfen, indem man gewisse Zusatzforderungen stellt.

Insgesamt sind also einige interessante Varianten zur Selbstreproduktion denkbar. Einige dieser Varianten sollen in diesem Kapitel präsentiert und in bezug auf die realen Programmiersprachen SIMULA und PASCAL an Hand von Beispielen erläutert werden.

4.2.Unendlich reproduzierende Programme

(4.2.1) Definition: Sei π ein (syntaktisch korrektes) Programm aus S.

(a) (i) Weist π keine Eingabe auf, so heißt π (streng) reproduzierend, wenn π (genau) ein syntaktisch korrektes Programm π' aus S ausgibt.

(ii) Weist π Eingabe auf, so heißt π (streng) reproduzierend, wenn π bei jeder zulässigen Eingabe (genau) ein syntaktisch korrektes Programm π' aus S ausgibt.

(b) R(S) bezeichnet die Menge aller reproduzierenden Programme aus S.

(4.2.2) Bemerkung: Jedes (streng) selbstreproduzierende Programm ist selbstverständlich (streng) reproduzierend.

Aus (4.2.2) folgt, daß es in den Programmiersprachen SIMULA und PASCAL reproduzierende Programme gibt, da in diesen Sprachen selbstreproduzierende Programme existieren.

(4.2.3) Lemma: In den Programmiersprachen SIMULA und PASCAL existieren unendlich viele reproduzierende Programme, die nicht selbstreproduzierend sind.

Beweis: (i) Für jedes $k \in \mathbb{N}$ ist das SIMULA-Programm

```
π_SIM(k) = begin
           OUTTEXT("BEGIN INTEGER I;
                    I:=k;
                    OUTINT(k, <Stelligkeit von k>)
                    END")
           end
```

reproduzierend, da die Textkonstante, die von $\pi_{SIM(k)}$ ausgegeben wird, ein gültiges SIMULA-Programm darstellt.

(ii) Ein analoges Programm läßt sich in PASCAL angeben:

```
π_PAS(k) = program T(OUTPUT);
           begin
           WRITE('PROGRAM T(OUTPUT);
                  BEGIN
                  WRITE(k)
                  END.')
           end.
```

%

(4.2.4) Definition: Sei $(\pi_i)_{i \in \mathbb{N}_o} = \pi_o, \pi_1, \pi_2, \ldots$ eine (unendliche) Folge von Programmen aus der Programmiersprache S. $(\pi_i)_{i \in \mathbb{N}_o}$ heißt Reproduktionsfolge, falls gilt

π_j reproduziert π_{j+1} für alle $j \in \mathbb{N}_o$.

Aus (4.2.4) folgt, daß jedes Programm einer Reproduktionsfolge als Startprogramm einer neuen Reproduktionsfolge aufgefaßt werden kann. Man braucht nur die entsprechende Teilfolge zu bilden. Dies rechtfertigt die folgende Definition.

(4.2.5) Definition: Sei $(\pi_i)_{i\in\mathbb{N}_0}$ eine Reproduktionsfolge aus der Programmiersprache S. Sei π_j, $j\in\mathbb{N}_0$, ein Element dieser Folge.

(i) π_j heißt unendlich reproduzierend.

(ii) Die Teilfolge $(\pi_k)^j_{k\in\mathbb{N}_0}$ von $(\pi_i)_{i\in\mathbb{N}_0}$ mit $\pi_k = \pi_{j+k}$ für alle $k\in\mathbb{N}_0$ heißt die Reproduktionsfolge von π_j.

(iii) U(S) bezeichnet die Menge aller unendlich reproduzierenden Programme aus S.

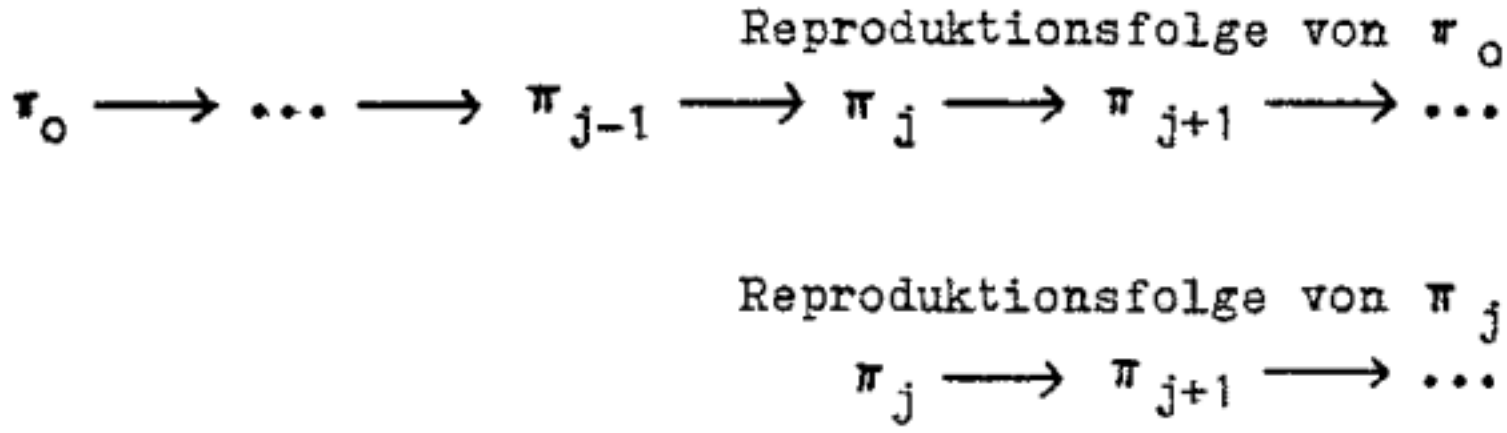


Abb.: 4.2.A

(4.2.6) Bemerkung: Jedes selbstreproduzierende Programm π ist unendlich reproduzierend. Die Reproduktionsfolge von π ist konstant.

(4.2.7) Satz: Es existiert ein unendlich reproduzierendes PASCAL-Programm, in dessen Reproduktionsfolge kein Programm mehrfach vorkommt.

Satz (4.2.7) wird durch das folgende Beispielprogramm $\tilde{\pi}_o$, das den Forderungen des Satzes genügt, bewiesen.

(4.2.8) Beispiel:

```
π̃o = program UR(OUTPUT);
     var I,K : integer;
     procedure Z(J : integer); begin WRITE(J+1) end;
```

```
procedure AA; begin WRITE('PROGRAM UR(OUTPUT); VA
   R I,K : INTEGER; PROCEDURE Z(J : INTEGER); BEG
   IN WRITE(J+1) END; PROCEDURE AA; BEGIN WRITE('
   '') end;
procedure C; begin WRITE('PROCEDURE ') end;
procedure A; begin WRITE('; BEGIN WRITE(''') end;
procedure B; begin WRITE(''') END;') end;
procedure AC; begin WRITE('''') end;
procedure BA; begin WRITE('A') end;
procedure BB; begin WRITE('B') end;
procedure BC; begin WRITE('C') end;
procedure CA; begin WRITE('BEGIN K:=') end;
procedure AB; begin WRITE(';FOR I:=1 TO K DO BEGI
   N WRITELN(I,I*I,I*I*I) END;AA;AA;AC;B;C;BC;A;C
   ;B;C;BA;A;A;AC;B;C;BB;A;AC;B;B;C;BA;BC;A;AC;AC
   ;B;C;BB;BA;A;BA;B;C;BB;BB;A;BB;B;C;BB;BC;A;BC;
   B;C;BC;BA;A;CA;B;C;BA;BB;A;AB;B;CA;Z(K);AB;WRI
   TELN END.') end;
begin
K:=Ø;
for I:=1 to K do
begin WRITELN(I,I*I,I*I*I) end;
AA;AA;AC;B;
C;BC;   A;    C;   B;
C;BA;   A;    A;AC;B;
C;BB;   A;AC; B;   B;
C;BA;BC;A;AC;AC;   B;
C;BB;BA;A;   BA;   B;
C;BB;BB;A;   BB;   B;
C;BB;BC;A;   BC;   B;
C;BC;BA;A;   CA;   B;
C;BA;BB;A;   AB;   B;CA;Z(K);AB;
WRITELN
end.
```

$\tilde{\pi}_o$ ist ein unendlich reproduzierendes Programm, in dessen Reproduktionsfolge kein Programm mehrfach auftritt.

Verifikation:
$\overset{\infty}{\pi}_0$ entspricht im wesentlichen dem selbstreproduzierenden Programm π_6 aus 3.3.2. Daß $\overset{\infty}{\pi}_0$ nicht ebenfalls selbstreproduzierend ist, wird durch Erhöhung der Obergrenze der Laufvariablen I um 1 in der Kopie $\overset{\infty}{\pi}_1$ von $\overset{\infty}{\pi}_0$ verhindert. Diese Erhöhung wird durch Aufruf der Prozedur Z erreicht, deren Text in den Programmkopf integriert ist. Durch die Erhöhung der Obergrenze der Laufvariablen ist die Kopie von $\overset{\infty}{\pi}_0$ sowohl textuell, als auch im Hinblick auf die Bedeutung des Programms, von $\overset{\infty}{\pi}_0$ verschieden. Da die Kopie sich lediglich in einer integer-Konstanten von $\overset{\infty}{\pi}_0$ unterscheidet, bleibt sie ein lauffähiges reproduzierendes Programm. In gleicher Weise unterscheidet sich die Kopie $\overset{\infty}{\pi}_2$ des Programms $\overset{\infty}{\pi}_1$ von $\overset{\infty}{\pi}_1$. Die Obergrenze der Laufvariablen I hat in $\overset{\infty}{\pi}_2$ einen um 2 größeren Wert als in $\overset{\infty}{\pi}_0$. Insgesamt gilt:

$\overset{\infty}{\pi}_0$ ist ein unendlich reproduzierendes Programm. In jedem Element $\overset{\infty}{\pi}_j$ der Reproduktionsfolge von $\overset{\infty}{\pi}_0$ hat die Obergrenze der Laufvariablen I den Wert j.

(4.2.9) Bemerkung:

I. Die Programme $\overset{\infty}{\pi}_j$, $j \in \mathbb{N}_0$, sind reproduzierend, aber nicht streng reproduzierend. Aus den Programmen $\overset{\infty}{\pi}_j$ lassen sich aber durch Streichung der Anweisung

```
WRITELN(I,I*I,I*I*I)
```

streng reproduzierende Programme gewinnen.
Satz (4.2.7) ließe sich also in dieser Hinsicht auch schärfer formulieren.

II. Die Programme der mittels $\overset{\infty}{\pi}_0$ gewonnenen Reproduktionsfolge enthalten alle den kompletten Selbstreproduktionsmechanismus von Programm π_6 aus 3.3.2. Gefordert war jedoch nur eine schwächere Eigenschaft, nämlich Reproduktion. Um nur Reproduktion zu erhalten, hatten wir den Selbstreproduktionsmechanismus durch Hinzunahme des zusätzlichen Programmteils

```
for I:=1 to ... do
begin ... end
```

abgeschwächt. Diese Vorgehensweise erscheint auf den ersten Blick widersinnig zu sein. Die Programme der Folge $(\overset{\infty}{\pi}_i)_{i\in\mathbb{N}_0}$ benötigen aber anscheinend den Selbstreproduktionsmechanismus, um unendlich viele voneinander verschiedene syntaktisch korrekte Programme zu erzeugen. Wünschenswert wäre eine Reproduktionsfolge mit Programmen, die mit schwächeren Mechanismen als dem Selbstreproduktionsmechanismus auskommen. Die Schwierigkeit, solche Folgen zu finden, kann als Hinweis darauf interpretiert werden, daß unendlich viele sukzessive auseinander hervorgehende Programme irgendwie „dicht" beieinander liegen müssen; so dicht, daß „Quasi-Selbstreproduktion" nötig ist, um sie überhaupt zu erzeugen.

III. Programm $\overset{\infty}{\pi}_0$ enthält nur Sprachkonzepte, die auch in der Programmiersprache SIMULA enthalten sind. Daher läßt sich Satz (4.2.7) auch entsprechend für die Programmiersprache SIMULA formulieren.

IV. Satz (4.2.7) hat in erster Linie theoretische Bedeutung. In der Praxis gibt es zwar unendlich reproduzierende Programme, aber nicht die entsprechenden Folgen, denn bei endlicher Speicherkapazität lassen sich auch nur endlich viele verschiedene Programme darstellen.

4.2.1.Implementierung des Programms $\overset{\infty}{\pi}_0$

Programm $\overset{\infty}{\pi}_0$ schreibt seine gesamte Ausgabe unformatiert in eine Zeile. Diese Zeile ist sowohl für den Puffer des Schnelldruckers als auch für den Eingabepuffer des PASCAL-Compilers zu lang. Für eine ausreichende Demonstration des Programms $\overset{\infty}{\pi}_0$ ist es aber wünschenswert, die Ausgabe von $\overset{\infty}{\pi}_0$, nämlich $\overset{\infty}{\pi}_1$, zu übersetzen und auszuführen. Wir verfahren deshalb wie in 3.3.3. und führen zur Formatierung der Eingabe die Prozedur Q ein:

```
procedure Q; begin WRITELN end;
```

Die relativ lange Textkonstante aus Prozedur AB wird auf 4 Prozeduren aufgespalten. Zu diesem Zweck werden die Pro-

zeduren AAA, CB und CC zusätzlich in das Programm aufgenommen. Anhang A.7. demonstriert das so veränderte Programm $\bar{\pi}_0$.

4.3.Zyklisch selbstreproduzierende Programme

(4.3.1) Definition: Sei π_0 ein unendlich reproduzierendes Programm aus der Programmiersprache S. Sei $(\pi_i)_{i \in \mathbb{N}_0}$ die Reproduktionsfolge von π_0.

(i) Existiert $j \geq 1$ mit $\pi_j = \pi_0$, so heißt π_0 zyklisch selbstreproduzierend.

(ii) Ist π_0 zyklisch selbstreproduzierend, so heißt das kleinste $j \geq 1$ mit $\pi_j = \pi_0$ die Zykluslänge von π_0.

(iii) Die Menge aller zyklisch selbstreproduzierenden Programme aus S wird mit Z(S) bezeichnet.

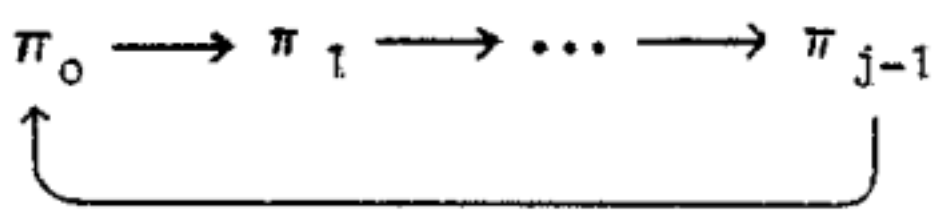


Abb.: 4.3.A

(4.3.2) Bemerkung: Jedes Programm π_j aus der Reproduktionsfolge eines zyklisch selbstreproduzierenden Programms π_0 ist zyklisch selbstreproduzierend und hat die gleiche Zykluslänge wie π_0.

(4.3.3) Satz: In der Programmiersprache PASCAL existiert für jedes $k \geq 1$ ein zyklisch selbstreproduzierendes Programm $\overset{k}{\pi}$ mit der Zykluslänge k.

Beweis: Das PASCAL-Programm $\overset{\infty}{\pi}_0$ aus 4.3. ist unendlich reproduzierend. Man kann aber sehr einfach aus $\overset{\infty}{\pi}_0$ ein zyklisch selbstreproduzierendes Programm $\overset{k}{\pi}_0$ für jedes $k \geq 1$ herleiten, indem man

```
procedure Z(J : integer); begin WRITE(J+1) end;
```

durch

```
procedure Z(J : integer); begin WRITE((J+1) mod k)end;
```

ersetzt.

Die Programme aus der Reproduktionsfolge von $\overset{\infty}{\pi}_0$ unterscheiden sich gerade in dem von Z ausgegebenen Wert. Die geänderte Prozedur Z stellt sicher, daß für jedes $k \geq 1$ gilt $\overset{k}{\pi}_0 = \overset{k}{\pi}_k$. Mit $\overset{k}{\pi}_0$ als $\overset{k}{\pi}$ gilt der Satz. %

Wir wollen noch ein Beispiel für zyklisch selbstreproduzierende Programme angeben.

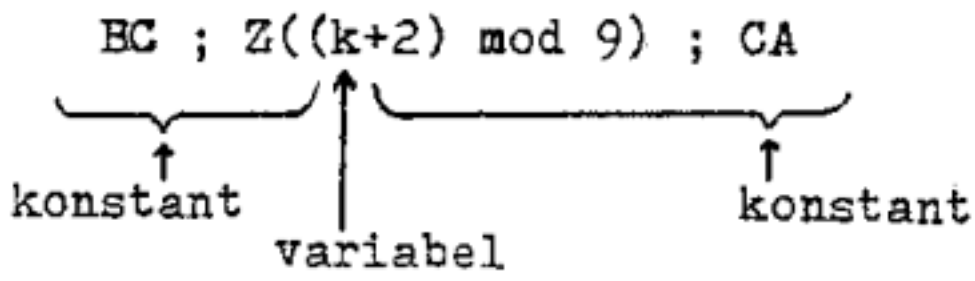
(4.3.4) Beispiel: Das folgende Programm π_0^{zyk} ist, abgesehen von einigen Umbenennungen, eine Abwandlung von π_6 aus Abschnitt 3.3.2. π_0^{zyk} soll sich erst nach einem Zyklus von N = 9 Schritten selbstreproduzieren. Dies wird dadurch erreicht, daß π_0^{zyk} einige seiner Prozeduren in einer anderen Reihenfolge ausgibt. Das resultierende Programm π_1^{zyk} verfährt mit den Prozeduren in analoger Weise. Erst nach 9 Schritten ist die Ausgangskonstellation der Prozeduren erreicht und π_0^{zyk} liegt wieder vor.

π_0^{zyk} hat gegenüber π_6 einen erweiterten Vereinbarungsteil durch:

(i)
```
integer I,K;
    procedure Z(J : integer); begin WRITE(J) end;
```

(ii) Aufspaltung der den Algorithmus von π_o^{zyk} druckenden Prozedur in zwei Prozeduren BC und CA.

Der Anweisungsteil von π_o^{zyk} muß bewirken, daß sich die direkte Kopie π_1^{zyk} von π_o^{zyk} unterscheidet. Erst nach 9 Schritten darf π_o^{zyk} wieder hergestellt sein. Der Anweisungsteil von π_o^{zyk} muß in allen Kopien „fast" der gleiche , aber doch variabel sein. Daher kann der Anweisungsteil von π_o^{zyk} nicht en bloc kopiert werden. Daraus resultiert auch die unter (ii) angegebene Aufspaltung. Mit den folgenden Prozeduraufrufen wird der Algorithmus von π_o^{zyk} und seiner Kopien angegeben.



π_o^{zyk} =

```
program ZYKLUS(OUTPUT);
var I,K : integer;
procedure Z(J : integer); begin WRITE(J) end;
procedure A; begin WRITE('PROGRAM ZYKLUS(OUTPUT);VAR I,K :
   INTEGER;PROCEDURE Z(J : INTEGER);BEGIN WRITE(J) END; PRO
   CEDURE A;BEGIN WRITE('') end;
procedure B; begin WRITE('PROCEDURE ') end;
procedure C; begin WRITE(';BEGIN WRITE(''') end;
procedure AA; begin WRITE(''') END;') end;
procedure AB; begin WRITE('''') end;
procedure AC; begin WRITE('A') end;
procedure BA; begin WRITE('B') end;
procedure BB; begin WRITE('C') end;
procedure BC; begin WRITE('BEGIN A;A;AB;AA;K:=') end;
procedure CA; begin WRITE(';FOR I:=1 TO 9 DO BEGIN B;CASE K
   OF Ø:BEGIN BA;C;B END;1:BEGIN BB;C;C;BA END;2:BEGIN AC;AC
   ;C;AB;AA END;3:BEGIN AC;BA;C;AB;AB END;4:BEGIN AC;BB;C;A
   C END;5:BEGIN BA;AC;C;BA END;6:BEGIN BA;BA;C;BB END;7:BE
```

```
   GIN BA;BB;C;BC END;8:BEGIN BB;AC;C;CA END END;AA;K:=(K+1) MO
   D 9 END;BC;Z((K+1) MOD 9);CA;WRITELN END.') end;
begin
A;A;AB;AA;
K:=1;
for I:=1 to 9 do
begin B;
  case K of
Ø:begin BA;C;B end;
1:begin BB;C;C;AB end;
2:begin AC;AC;C;AB;AA end;
3:begin AC;BA;C;AB;AB end;
4:begin AC;BB;C;AC end;
5:begin BA;AC;C;BA end;
6:begin BA;BA;C;BB end;
7:begin BA;BB;C;BC end;
8:begin BB;AC;C;CA  end
  end;
  AA;
  K:=(K+1) mod 9
end;
BC;Z((K+1) mod 9);CA;
WRITELN
end.
```

Verifikation

Die Programme π_0^{zyk} , π_1^{zyk} , ... , π_8^{zyk} stimmen bis auf die Reihenfolge der Prozeduren B,...,CA und den Startwert von K textuell überein. Die Ausgabefolge dieser Prozeduren wird über die Variable k gesteuert. Wir betrachten dazu die folgende Tabelle:

Startwert k in π_i^{zyk} ↓ — Die Wertefolge, die k in π_i^{zyk} durchläuft — Startwert k, der von π_i^{zyk} an π_{i+1}^{zyk} weitergegeben wird ↓

		I=1	I=2	I=3	I=4	I=5	I=6	I=7	I=8	I=9	
π_0^{zyk}	k=1	2	3	4	5	6	7	8	0	1	2
π_1^{zyk}	k=2	3	4	5	6	7	8	0	1	2	3
π_2^{zyk}	k=3	4	5	6	7	8	0	1	2	3	4
π_3^{zyk}	k=4	5	6	7	8	0	1	2	3	4	5
π_4^{zyk}	k=5	6	7	8	0	1	2	3	4	5	6
π_5^{zyk}	k=6	7	8	0	1	2	3	4	5	6	7
π_6^{zyk}	k=7	8	0	1	2	3	4	5	6	7	8
π_7^{zyk}	k=8	0	1	2	3	4	5	6	7	8	0
π_8^{zyk}	k=0	1	2	3	4	5	6	7	8	0	1

In jeder Zeile der Tabelle kommen alle Werte von Ø bis 8 in den Spalten „I=1" bis „I=8" genau einmal vor. Damit ist aufGrund der case-Anweisung sichergestellt, daß jedes π_i^{zyk} alle Prozeduren B bis CA ausgibt. Aus der letzten Spalte folgt, daß der Startwert von k in jeder Kopie π_i^{zyk}, i=Ø,...,8 verschieden ist. Ebenfalls aus der letzten Spalte folgt, daß der Startwert von k in der Kopie von π_8^{zyk} gleich dem Startwert von k in π_o^{zyk} ist. Da sich die Anweisungsteile der π_i^{zyk} nur in dem Startwert von k unterscheiden, gilt:

$$\pi_9^{zyk} = \pi_o^{zyk}.$$

(4.3.5) Bemerkung:

I. Im wesentlichen gilt auch im Falle des zyklisch selbstreproduzierenden Programms π_o^{zyk} Bemerkung (4.2.9)II. für das unendlich reproduzierende Programm $\overline{\pi}_o^{\infty}$. Überhaupt haben unsere Beispielprogramme für unendlich reproduzierende und zyklisch selbstreproduzierende Programme keine Vereinfachung des Selbstreproduktionsmechanismus von Programm π_6 gebracht.

II. Satz (4.3.3) läßt sich natürlich analog für die Programmiersprache SIMULA formulieren.

(4.3.6) Beispiel: Als Beispiel für ein zyklisch selbstreproduzierendes SIMULA-Programm sei hier die SIMULA-Version des Programms π_o^{zyk} angegeben.

```
begin
integer I,K;
procedure Z(J); integer J; OUTINT(J,1);
procedure A; OUTTEXT("BEGIN INTEGER I,K; PROCEDURE
   Z(J); INTEGER J; OUTINT(J,1); PROCEDURE A; OUTTE
   XT(""");
procedure B; OUTTEXT("PROCEDURE ");
procedure C; OUTTEXT(";OUTTEXT(""");
procedure AA; OUTTEXT(""");");
procedure AB; OUTTEXT("""");
procedure AC; OUTTEXT("A");
```

```
procedure BA; OUTTEXT("B");
procedure BB; OUTTEXT("C");
procedure BC; OUTTEXT("A;A;AB;AA;K:=");
procedure CA; OUTTEXT(";FOR I:=1 STEP 1 UNTIL 9 DO
   BEGIN B;IF K=Ø THEN BEGIN BA;C;B END ELSE IF K=1
   THEN BEGIN BB;C;C;AB END ELSE IF K=2 THEN BEGIN
   AC;AC;C;AB;AA END ELSE IF K=3 THEN BEGIN AC;BA;C
   ;AB;AB END ELSE IF K=4 THEN BEGIN AC;BB;C;AC END
   ELSE IF K=5 THEN BEGIN BA;AC;C;BA END ELSE IF K=
   6 THEN BEGIN BA;BB;C;BC END ELSE IF K=7 THEN BEG
   IN BA;BB;C;BC END ELSE BEGIN BB;AC;C;CA END;AA;K
   :=(K+1) MOD 9;END;BC;Z((K+2) MOD 9);CA END;");
A;A;AB;AA;
K:=1;
for I:=1 step 1 until 9 do
begin B;
      if K=Ø then begin BA;C;B end
else if K=1 then begin BB;C;C;AB end
else if K=2 then begin AC;AC;C;AB;AA end
else if K=3 then begin AC;BA;C;AB;AB end
else if K=4 then begin AC;BB;C;AC end
else if K=5 then begin BA;AC;C;BA end
else if K=6 then begin BA;BA;C;BB end
else if K=7 then begin BA;BB;C;BC end
            else begin BB;AC;C;CA end;
AA;
K:=(K+1) mod 9;
end;
BC;Z((K+1) mod 9);CA
end;
```

Verifikation

Die Verifikation dieses SIMULA-Programms ist identisch mit der des PASCAL-Programms π_o^{zyk}.

4.3.1. Implementierung des Programms $\overset{k}{\pi}_o$

Für die Implementierung von $\overset{k}{\pi}_o$ gelten die gleichen Bemerkungen wie in Abschnitt 4.2.1. zur Implementierung von Programm $\overset{\infty}{\pi}_o$. Näheres ist aus Anhang A.8. ersichtlich.

4.3.2. Implementierung des Programms π_o^{zyk}

Auch für die Implementierung von π_o^{zyk} gelten die Bemerkungen von Abschnitt 4.2.1. Die Textkonstante, die den Algorithmus von π_o^{zyk} darstellt, muß aus Gründen der Formatierung auf noch mehr Prozeduren aufgespalten werden, als dies etwa in $\overset{\infty}{\pi}_o$ der Fall ist. Alles Weitere siehe Anhang A.9.

4.4. Unter Wechsel der Programmiersprache sich zyklisch selbstreproduzierende Programme

In Abschnitt 4.3. haben wir zyklisch selbstreproduzierende Programme als spezielle unendlich reproduzierende Programme kennengelernt. Unendlich reproduzierende Programme sind ihrerseits reproduzierende Programme. Nach Definition (4.2.1) gibt ein reproduzierendes Programm π ein Programm π' aus. Dabei sind π und π'Programme aus derselben Programmiersprache S. Wir hätten reproduzierende Programme auch anders definieren können, indem wir π' aus einer Programmiersprache $S' \neq S$ zugelassen hätten. Eine solche Definition wäre allgemeiner als Definition (4.2.1). Entsprechend allgemeiner wären dann auch die Definitionen für unendlich reproduzierende und zyklisch selbstreproduzierende Programme ausgefallen. Daß eine solche Verallgemeinerung durchaus sinnvoll wäre, soll das folgende Beispiel demonstrieren. Beispiel (4.4.1) stellt ein Programm vor, das nicht nur ein Programm in einer anderen Programmiersprache ausgibt, sondern sich auch noch zyklisch selbstreproduziert.

(4.4.1) Beispiel: Wir gehen aus von dem SIMULA-Programm π_4 aus 3.2.6. und dem PASCAL-Programm π_6 aus 3.3.2.. Beide Programme sind nahezu identisch, da sie praktisch ihre gegenseitigen Übersetzungen darstellen. Aus beiden Programmen kombinieren wir jeweils ein PASCAL-Programm π_{PAS} und ein SIMULA-Programm π_{SIM} mit:

π_{PAS} gibt π_{SIM} aus
π_{SIM} gibt π_{PAS} aus

π_{SIM} und π_{PAS} sind also zyklisch selbstreproduzierende Programme mit Wechsel der Programmiersprachen.

π_{PAS} =

```
program X(OUTPUT);
var T,F:boolean;
procedure A(Z:boolean);begin if Z
   then WRITE('BEGIN BOOLEAN T,F;')
   else WRITE('PROGRAM X(OUTPUT);VAR T,F:BOOLEAN;') end;
procedure B(Z:boolean);begin if Z
   then WRITE('PROCEDURE ')
   else WRITE('PROCEDURE ') end;
procedure C(Z:boolean);begin if Z
   then WRITE('(Z);BOOLEAN Z;IF Z THEN OUTTEXT(")
   else WRITE('(Z:BOOLEAN);BEGIN IF Z THEN WRITE(''') end;
procedure AA(Z:boolean);begin if Z
   then WRITE('") ELSE OUTTEXT(")
   else WRITE(''') ELSE WRITE(''') end;
procedure AB(Z:boolean);begin if Z
   then WRITE('");')
   else WRITE(''') END;') end;
procedure CB(Z:boolean);begin if Z
   then WRITE('")
   else WRITE('''') end;
procedure BA(Z:boolean);begin if Z
   then WRITE('A')
   else WRITE('A') end;
```

```
procedure BB(Z:boolean);begin if Z
   then WRITE('B')
   else WRITE('B') end;
procedure BC(Z:boolean);begin if Z
   then WRITE('C')
   else WRITE('C') end;
procedure AC(Z:boolean);begin if Z
   then WRITE('T:=TRUE;F:=FALSE; ⊛ END')
   else WRITE('BEGIN T:=TRUE;F:=FALSE; ⊛ ;WRITELN END.') end;
begin T:=true;F:=false;
A(T);
B(T);BA(T);       C(T); A(F);AA(T);        A(T);       AB(T);
B(T);BB(T);       C(T); B(T);AA(T);        B(T);       AB(T);
B(T);BC(T);       C(T); C(F);AA(T);        C(T);CB(T);AB(T);
B(T);BA(T);BA(T);C(T);AA(F);AA(T);CB(T);AA(T);CB(T);AB(T);
B(T);BA(T);BB(T);C(T);AB(F);AA(T);CB(T);AB(T);       AB(T);
B(T);BC(T);BB(T);C(T);CB(F);AA(T);       CB(T);CB(T);AB(T);
B(T);BB(T);BA(T);C(T);BA(T);AA(T);       BA(T);      AB(T);
B(T);BB(T);BB(T);C(T);BB(T);AA(T);       BB(T);      AB(T);
B(T);BB(T);BC(T);C(T);BC(T);AA(T);       BC(T);      AB(T);
B(T);BA(T);BC(T);C(T);AC(F);AA(T);       AC(T);      AB(T);
AC(T)
;WRITELN
end.
```

⊛

π_{PAS} gibt das SIMULA-Programm π_{SIM} aus:

π_{SIM} =

```
begin
boolean T,F;
procedure A(Z);boolean Z;if Z then
   OUTTEXT("PROGRAM X(OUTPUT);VAR T,F:BOOLEAN;")
   else OUTTEXT("BEGIN BOOLEAN T,F;");
procedure B(Z);boolean Z;if Z
   then OUTTEXT("PROCEDURE ")
   else OUTTEXT("PROCEDURE ");
procedure C(Z);boolean Z;if Z
   then OUTTEXT("(Z:BOOLEAN);BEGIN IF Z THEN WRITE('")
```

```
   else OUTTEXT("(Z);BOOLEAN Z; IF Z THEN OUTTEXT("""");
procedure AA(Z);boolean Z;if Z
   then OUTTEXT("') ELSE WRITE('")
   else OUTTEXT(""") ELSE OUTTEXT(""");
procedure AB(Z);boolean Z;if Z
   then OUTTEXT("') END;")
   else OUTTEXT(""");");
procedure CB(Z);boolean Z;if Z
   then OUTTEXT("'")
   else OUTTEXT("""");
procedure BA(Z);boolean Z;if Z
   then OUTTEXT("A")
   else OUTTEXT("A");
procedure BB(Z);boolean Z;if Z
   then OUTTEXT("B")
   else OUTTEXT("B");
procedure BC(Z);boolean Z;if Z
   then OUTTEXT("C")
   else OUTTEXT("C");
procedure AC(Z);boolean Z; if Z
   then OUTTEXT("BEGIN T:=TRUE;F:=FALSE; ⊛ ;WRITELN END.")
   else OUTTEXT("T:=TRUE;F:=FALSE; ⊛ END");
T:=true;F:=false; ⊛
end
```

Verifikation

π_{PAS} und π_{SIM} enthalten jeweils alle Teilstrings - sowohl der Zerlegung von π_{PAS} als auch der Zerlegung von π_{SIM} - in den Prozeduren A bis AC. Da sich die Teilstrings der Zerlegungen von π_{PAS} und π_{SIM} eins zu eins entsprechen, können die Teilstrings alternativ in den Prozeduren abgelegt werden. Jede Prozedur von π_{PAS} hat dann den allgemeinen Aufbau:

```
procedure  <name>  (Z:boolean);
begin if Z then WRITE('<Teilstring s aus π SIM> ')
          else WRITE('<dem Teilstring s entsprechender
                      Teilstring s' aus π PAS> ')
end;
```

Einige Teilstrings von π_{PAS} sind gleich ihren Entsprechungen in π_{SIM}. Die Prozeduren, die diese Teilstrings bearbeiten, enthalten natürlich Redundanz. Beispiel:

```
procedure BA(Z:boolean);
begin if Z then WRITE('A') else WRITE('A') end;
```

Die Redundanz wird aber zugunsten eines einheitlichen Prozedurschemas in Kauf genommen. Die Auswahl, welche Alternative ausgegeben werden soll, wird beim Aufruf der Prozeduren durch ihren aktuellen Parameter getroffen. Das PASCAL-Programm enthält die SIMULA-Teilstrings immer im then-Zweig der Prozeduren und die PASCAL-Teilstrings immer im else-Zweig. Beim SIMULA-Programm sind die Verhältnisse genau umgekehrt. Dadurch wird erreicht, daß eine Prozedur, die im PASCAL-Programm mit true aufgerufen wird, auch im SIMULA-Programm mit true aufgerufen werden kann. Daraus folgt, daß die Anweisungsteile von π_{SIM} und π_{PAS} im wesentlichen identisch sind. Aus dem bisher Gesagten und der Tatsache, daß π_{SIM} und π_{PAS} bis auf die alternativen Prozeduren in π_4 und π_6 identisch sind, folgt: π_{PAS} reproduziert π_{SIM} und umgekehrt.

4.5.k-fach selbstreproduzierende Programme

In Abschnitt 4.2. haben wir Reproduktion als Abschwächung der Selbstreproduktion kennengelernt. Wir wollen nun die k-fache Selbstreproduktion von Programmen als Verschärfung der einfachen Selbstreproduktion einführen:

(4.5.1) Definition: Sei $k > 1$. Sei π aus S.(syntakt. korrekt).

a) (i) Weist π keine Eingabe auf, so heißt π k-fach selbstreproduzierend, falls π seinen Programmtext in S k-mal ausgibt.

(ii) Weist π Eingabe auf, so heißt π k-fach selbstreproduzierend, falls π bei jeder zulässigen Eingabe seinen Programmtext in S k-mal ausgibt.

b) $SR^k(S)$ bezeichnet die Menge aller k-fach selbstreproduzierenden Programme aus S.

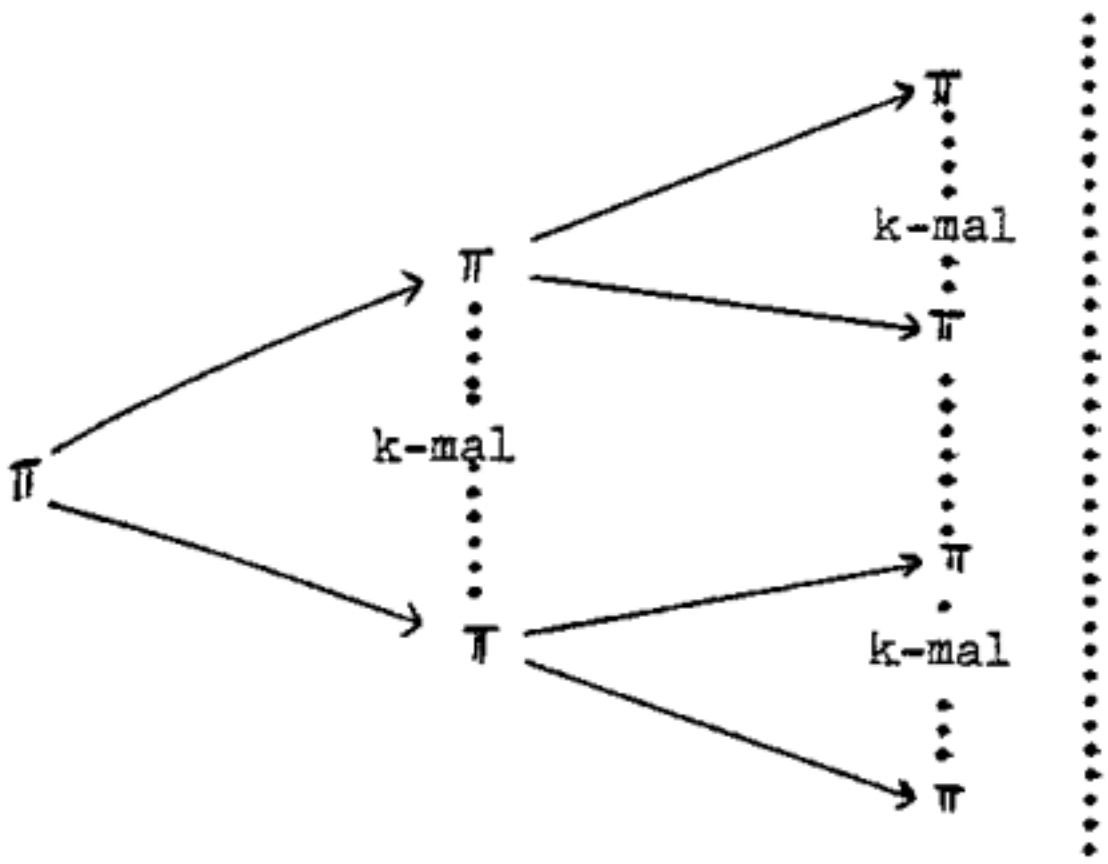


Abb.: 4.5.A

Die Existenz k-fach selbstreproduzierender Programme folgt bereits aus Korollar (2.8.9).

(4.5.2) Satz: Die Programmiersprache PASCAL enthält für jedes $k > 1$ ein k-fach selbstreproduzierendes Programm $\pi(k)$.

Wir geben ein Beispiel eines für jedes $k > 1$ k-fach selbstreproduzierenden Programms an. Dieses Beispiel beweist Satz (4.5.2).

(4.5.3) Beispiel:

$\pi(k)$ =

```
program PIK (OUTPUT);
var I:integer;
procedure AA;begin WRITE('PROGRAM PIK(OUTPUT);VAR I:INTEGER
   ;PROCEDURE AA;BEGIN WRITE(''') end;
procedure C;begin WRITE('PROCEDURE ') end;
procedure A;begin WRITE(';BEGIN WRITE(''') end;
procedure B;begin WRITE(''') END;') end;
procedure AC;begin WRITE('''') end;
procedure BA;begin WRITE('A') end;
```

```
procedure BB;begin WRITE('B') end;
procedure BC;begin WRITE('C') end;
procedure AB;begin WRITE('BEGIN FOR I:=1 TO 5 DO BEGIN AA;A
   A;AC;B;C;BC;A;C;B;C;BA;A;A;AC;B;C;BB;A;AC;B;B;C;BA;BC;A;
   AC;AC;B;C;BB;BA;A;BA;B;C;BB;BB;A;BB;B;C;BB;BC;A;BC;B;C;B
   A;BB;A;AB;B;AB;WRITELN END END.') end;
begin
for I:=1 to 5 do
begin
AA;              AA;AC;B;
   C;BC;    A;     C;    B;
   C;BA;    A;     A;AC;B;
   C;BB;    A;AC;  B;    B;
   C;BA;BC;A;AC;AC;    B;
   C;BB;BA;A;    BA;    B;
   C;BB;BB;A;    BB;    B;
   C;BB;BC;A;    BC;    B;
   C;BA;BB;A;    AB;    B;AB;WRITELN
end
end.
```

Verifikation

Die Verifikation von $\pi(k)$ ergibt sich direkt aus der Verifikation von Programm π_6 aus 3.3.2.. Der Unterschied zwischen $\pi(k)$ und π_6 besteht im wesentlichen nur in der for-Schleife

```
for I:=1 to k do
begin ... end   ,
```

in die der Ausgabealgorithmus eingebettet ist. Der Ausgabealgorithmus von π_6 wird also in $\pi(k)$ gerade k-mal ausgeführt. Dadurch wird $\pi(k)$ zum k-fach reproduzierenden Programm.

(4.5.4) Bemerkung:

I. Jedes k-fach selbstreproduzierende Programm π ist natürlich selbstreproduzierend, aber nicht streng selbstreproduzierend. Außerdem ist ein k-fach selbstreproduzierendes Programm zyklisch selbstreproduzierend mit der Zykluslänge 1.

II. Satz (4.5.2) gilt in analoger Formulierung für die Programmiersprache SIMULA. Zum Beweis übersetzt man das Programm $\pi(k)$ in ein entsprechendes SIMULA-Programm. Das geht ohne Schwierigkeiten, da $\pi(k)$ keine pascalspezifischen Konstruktionen enthält.

4.5.1. Implementierung von $\pi(k)$

Zur Implementierung von $\pi(k)$ sind die gleichen Bemerkungen wie in Abschnitt 3.3.3. zu machen. Anhang A.10. zeigt die Implementierung von $\pi(k)$ mit k=5.

4.6. Reproduktionshierarchie bei Programmen

In den vorangegangenen Abschnitten haben wir für eine beliebige Programmiersprache S die Mengen

$$R(S)\ ,\ U(S)\ ,\ Z(S) \text{ und } SR^k(S)$$

definiert. Für diese Mengen gilt

(1) $$SR^k(S) \subset SR(S) \subset Z(S) \subset U(S) \subset R(S) \quad ,$$

wobei SR(S) die Menge der selbstreproduzierende Programme aus S ist. Im allgemeinen werden die Inklusionen echt sein, wie wir am Beispiel der Programmiersprache PASCAL gesehen haben. Es gilt nämlich

$\pi_{PAS(k)}$ aus Abschnitt 4.2. ist reproduzierend, aber nicht unendlich reproduzierend.

$\overset{\infty}{\pi}_o$ aus Abschnitt 4.2. ist unendlich reproduzierend, aber nicht zyklisch selbstreproduzierend.

π_o^{zyk} aus Abschnitt 4.3. ist zyklisch selbstreproduzierend, aber nicht selbstreproduzierend.

π_6 aus Abschnitt 3.3.2. ist selbstreproduzierend, aber nicht k-fach selbstreproduzierend für ein $k > 1$.

Daraus folgt:

$$SR^k(PASCAL) \subsetneq SR(PASCAL) \subsetneq Z(PASCAL) \subsetneq U(PASCAL) \subsetneq R(PASCAL)$$

Abbildung 4.6.A erläutert dieses Ergebnis graphisch.

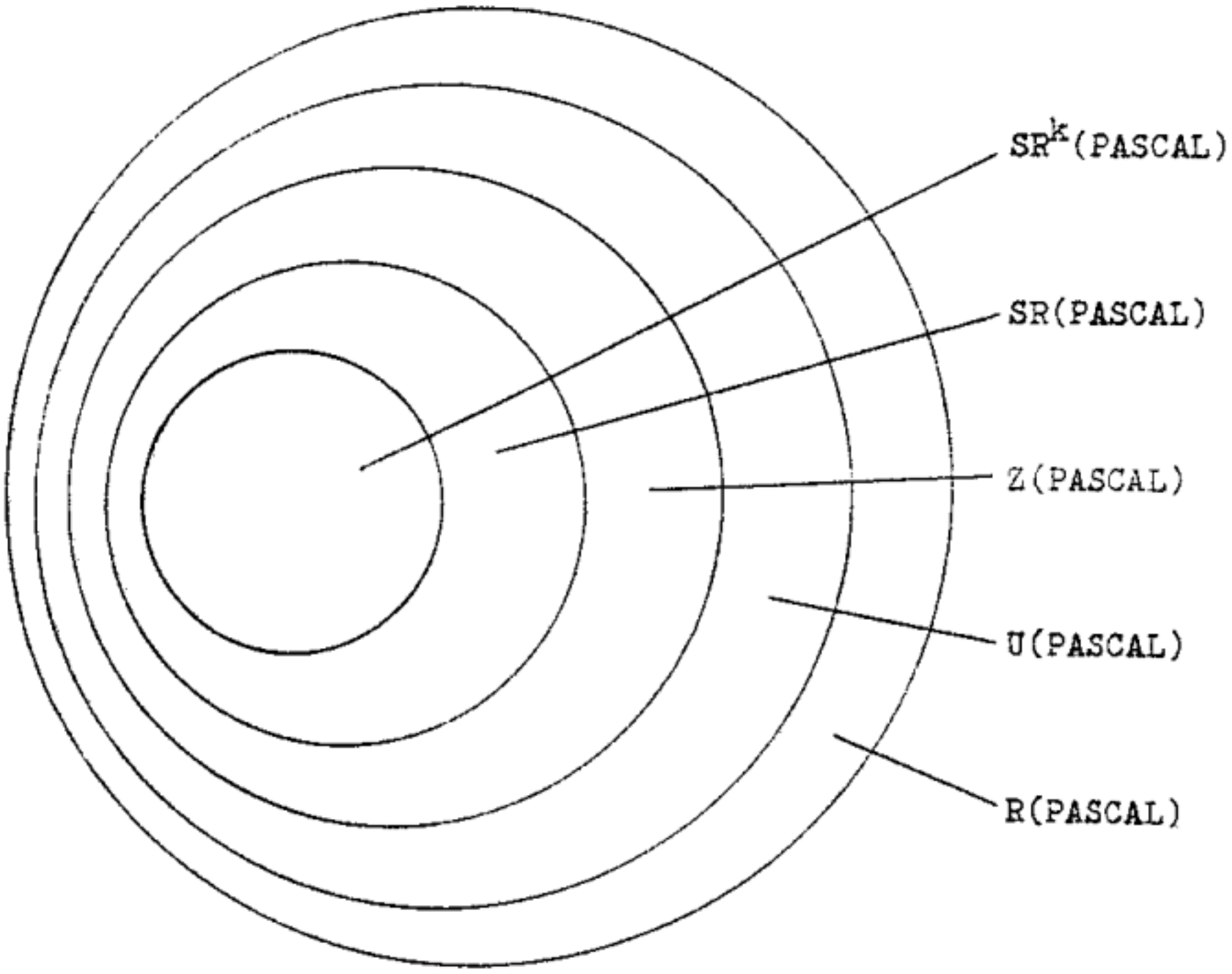


Abb.: 4.6.A

(4.6.1) Definition: Für jede Programmiersprache S heißt die Inklusionenreihe (1) Reproduktionshierarchie von S.

5. Selbstreproduzierende Programme mit Zusatzeigenschaften

5.1. Einleitung

In Kapitel 3 haben wir einige Beispiele für selbstreproduzierende Programme kennengelernt. Diesen Beispielprogrammen ist gemeinsam, daß sie außer der Ausgabe ihres eigenen Textes keinerlei Funktion ausführen. Eine interessante Fragestellung ist aber etwa:

(1) „Gibt es in der Programmiersprache S Programme, die mehr leisten als nur Selbstreproduktion?"

oder konkreter

„Gibt es in S selbstreproduzierende Programme, die zusätzlich einen Suchalgorithmus realisieren, oder die Primzahlzerlegung ausführen, oder ein Datenbanksystem verwalten, oder ... ?"

Angenommen, Frage (1) ließe sich für die Programmiersprache S positiv beantworten, so könnte man etwas schärfer fragen:

(2) „Existiert zu jedem Programm π aus S ein selbstreproduzierendes Programm $\tilde{\pi}$ aus S, das die gleiche Funktion realisiert wie π?"

Ist die letzte Frage für die Programmiersprache S zu bejahen, so ist intuitiv klar, daß ein selbstreproduzierendes Programm $\tilde{\pi}$, das eine gegebene Funktion realisiert, umfangreicher und komplizierter ist als ein nicht selbstreproduzierendes Programm π, das die gleiche Funktion realisiert. Daher ist es vielleicht einfacher, auf der Suche nach $\tilde{\pi}$ zunächst ein nicht selbstreproduzierendes Programm π zu entwickeln, und dieses anschließend in eine selbstreproduzierende Version $\tilde{\pi}$ zu transformieren. In diesem Zusammenhang drängt sich die folgende Frage auf:

(3) „Gibt es für eine gegebene Programmiersprache S einen Algorithmus, der zu jedem Programm π aus S ein selbstreproduzierendes Programm $\tilde{\pi}$ aus S liefert, das die gleiche Funktion realisiert wie π ?"

Will man die Beantwortung der Fragen (1) bis (3) in Angriff nehmen, so muß zunächst geklärt werden, was unter der „von einem Programm π aus S realisierten Funktion" zu verstehen ist. Wegen der Vielfalt an realen Programmiersprachen, deren unterschiedlichen Datentypen und der unterschiedlichen Interpretation auf verschiedenen Rechenanlagen dürfte es unmöglich sein, den obigen Begriff formal exakt und zudem noch allgemeingültig zu definieren. Für unsere Zwecke sollen jedoch die folgenden Überlegungen und Definitionen genügen.

Auf realen Rechenanlagen verarbeiten Programme aus konkreten Programmiersprachen in der Regel Daten, die auf mehreren Eingabedateien stehen, und geben Ergebnisse auf mehrere Ausgabedateien aus. Auf jeder dieser Dateien stehen Zeichenketten, die als ganze Zahlen, reelle Zahlen, Texte u.s.w. von einem Programm π aus der Sprache S interpretiert werden können. Diese Interpretation kann natürlich nur dann erfolgen, wenn die Zeichen, die auf den Dateien stehen, aus dem für Daten für Programme aus S zulässigen endlichen Alphabet A_S stammen. Der Inhalt einer Datei kann als Wort aus A_S^* aufgefaßt werden. Dieser Sichtweise entspricht die folgende Definition, die zudem den Fall zuläßt, daß während der Laufzeit von π einige Dateien sowohl als Eingabe- als auch als Ausgabedatei benutzt werden.

(5.1.1) Definition: Sei π ein Programm aus S mit $p \geq 0$ Ein- und Ausgabedateien, von denen q, $0 \leq q \leq p$, Dateien als Ausgabedateien benutzt werden. Die von π realisierte Funktion f_π ist eine partielle Funktion von $(A_S^p)^*$ nach $(A_S^q)^*$, die jeder Belegung der p Ein- und

Ausgabedateien mit Worten aus (A_S^*) genau eine Belegung der q Ausgabedateien mit Worten aus (A_S^*) zuordnet.

(5.1.2) Beispiel: Gegeben sei das PASCAL-Programm

```
π_0 = program X(INPUT,OUTPUT);
      var I : integer;
          Y : real;
      begin
      for I:=1 to 1Ø do
      begin READ(Y); WRITELN(SQRT(Y)) end
      end.
```

π_0 liest also 10 reelle Zahlen aus der Eingabedatei INPUT ein und gibt deren Quadratwurzeln auf die Ausgabedatei OUTPUT aus. Sei A_{PAS} die Menge aller Zeichen, die ein PASCAL-Programm verarbeiten kann. Dann liefert Definition (5.1.1):

π_0 realisiert die Funktion $f_{\pi_0} : A_{PAS}^* \longrightarrow A_{PAS}^*$

$$x \longmapsto f_{\pi_0}(x), \quad x \in A_{PAS}^*$$

Läßt sich der Anfang von x nicht als Folge von 10 reellen Zahlen interpretieren, so ist $f_{\pi_0}(x)$ undefiniert. Andernfalls ist $f_{\pi_0}(x)$ ein Wort aus A_{PAS}^*, dessen Anfang sich als Folge von ebenfalls 10 reellen Zahlen interpretieren läßt. Diese Folge ist gleich der Folge der Quadratwurzeln der reellen Zahlen der Eingabefolge.

Eine exakte Beschreibung von f_{π_0} würde die explizite Einführung von Konvertierungsfunktionen von $\mathbb{R}$ nach A_{PAS}^* und von A_{PAS}^* nach $\mathbb{R}$ voraussetzen.

(5.1.3) Bemerkung:

I. Definition (5.1.1) ist natürlich nicht formal exakt, sondern eher als „praxisnah" zu bezeichnen.

II. Definition (5.1.1) entspricht im wesentlichen der Definition der von PL(A)-Programmen realisierten Funktion in (2.4.1) . Im Falle p und/oder q gleich 0

ist wie unter (2.4.2) zu verfahren.

Aus Definition (5.1.1) folgt, daß ein selbstreproduzierendes Programm $\tilde{\pi}$ aus S, das ohne zusätzliche Eingabe die „gleiche" Funktion realisiert wie ein anderes, nicht selbstreproduzierendes Programm π aus S, natürlich eine ganz andere Funktion realisiert als π, da die Ausgaben von π und $\tilde{\pi}$ verschieden sind. Um diesen verwirrenden Sprachgebrauch zu umgehen, geben wir Definition (5.1.4) an. Zuvor sei jedoch bemerkt, daß man jede Funktion $F : M^n \longrightarrow M^m$, $m,n \in \mathbb{N}_0$, als m-Tupel $F = (F_1,\ldots F_m)$ von Funktionen $F_i : M^n \longrightarrow M$, $i=1,\ldots,m$, auffassen kann. Dabei ist M eine beliebige Menge. Es gilt: $F(x) = (F_1(x),\ldots,F_m(x))$ für alle $x \in M^n$.

(5.1.4) Definition: Sei π ein Programm aus S. Ein Programm $\tilde{\pi}$ aus S heißt selbstreproduzierende Version von π, falls für die von π und $\tilde{\pi}$ realisierten Funktionen

$f_\pi : (A_S^*)^{p_1} \longrightarrow (A_S^*)^{q_1}$ und $f_{\tilde{\pi}} : (A_S^*)^{p_2} \longrightarrow (A_S^*)^{q_2}$

(i) oder (ii) gilt.

(i) $p_1 = p_2$

und $q_1 = q_2$

und $\exists$ genau ein $j \in \{1,\ldots,q_2\}$ mit

$(f_{\tilde{\pi}})_i(\bar{x}) = (f_\pi)_i(\bar{x})$ für $i \neq j$

$(f_{\tilde{\pi}})_j(\bar{x}) = (f_\pi)_j(\bar{x}) \circ \alpha \circ \tilde{\pi} \circ \beta$ [1]

wobei $\bar{x} \in (A_S^*)^{p_1}$, $\alpha,\beta \in A_S^*$

(ii) $p_2 = p_1$

und $q_2 = q_1+1$

und $(f_{\tilde{\pi}})_i(\bar{x}) = (f_\pi)_i(\bar{x})$ für $i \in \{1,\ldots,q_1\}$

$(f_{\tilde{\pi}})_{q_2}(\bar{x}) = \alpha \circ \tilde{\pi} \circ \beta$, wobei $\bar{x} \in (A_S^*)^{p_1}$, $\alpha,\beta \in A_S^*$

Definition (5.1.4) stellt sicher, daß $\tilde{\pi}$ unabhängig von der Eingabe seinen eigenen Text ausgibt. $\tilde{\pi}$ hängt seinen Text entweder an ein Ausgabewort, das auch π ausgibt (Fall(i)), oder $\tilde{\pi}$ gibt seinen Text als zusätzliches Wort aus (Fall(ii)).

---

[1] „$\circ$" bezeichnet die Konkatenation von Worten; hier aus A_S^*.

(5.1.5) Bemerkung: I.a. ist die selbstreproduzierende Version $\tilde{\pi}$ eines Programms π aus S nicht eindeutig.

Mit Hilfe von Definition (5.1.4) sind wir in der Lage, die Fragestellungen (2) und (3) exakter zu formulieren:

(2) „Existiert zu jedem Programm π aus einer gegebenen Programmiersprache S eine selbstreproduzierende Version von π ?"

(3) „Gibt es für eine gegebene Programmiersprache S einen Algorithmus, der für jedes Programm π aus S eine selbstreproduzierende Version $\tilde{\pi}$ von π liefert?"

Wir werden im folgenden die Fragen (1) bis (3) für die Programmiersprachen SIMULA und PASCAL explizit beantworten.

5.2.Selbstreproduktionssatz für die Programmiersprache PASCAL

Frage (1) aus 5.1. läßt sich für die Programmiersprache PASCAL durch das folgende Beispiel beantworten.

(5.2.1) Beispiel: Wir geben eine selbstreproduzierende Version $\tilde{\pi}_0$ zu dem Programm π_0 aus Beispiel (5.1.2) an. Wir gehen dabei aus von unserem kürzesten PASCAL-Programm π_6 aus Abschnitt 3.3.2. und versuchen, π_6 und π_0 zu einer selbstreproduzierenden Version $\tilde{\pi}_0$ zu kombinieren. Zu diesem Zweck vergegenwärtigen wir uns noch einmal das Programm π_6. π_6 enthält in den Prozeduren A,...,AC seinen eigenen Text in Form von Teilstrings. Mehrfach vorkommende Teilstrings sind natürlich nur einmal gespeichert. Der Text von π_6 läßt sich aber durch Aneinanderreihen dieser Teilstrings zusammensetzen. Der erste Teilstring s_1 von π_6 enthält die das Programm einleitenden Phrasen bis zum ersten '. Der letzte, in der Prozedur AB enthaltene Teilstring s_9 beinhaltet den kompletten Anweisungsteil von π_6. Abbildung 5.2.A verdeutlicht diesen Zusammenhang.

π_6 =

```
program PI6(OUTPUT);                    = s1
procedure AA; begin WRITE('               s1          ') end;

procedure C; begin ..................... end;
           :                                :
           :                                :
procedure BC; begin .................... end;
procedure AB; begin WRITE('      s9         ') end;

begin
AA;AA;AC;B;
..........
AB;WRITELN      = s9
end
end.
```

Abb.: 5.2.A

Idee: Wir integrieren den Programmkopf von π_o

```
program X(INPUT,OUTPUT);
var I : integer;
    Y : real;
```

in den String s_1 und den Anweisungsteil von π_o

```
for I:=1 to 1Ø do
begin READ(Y); WRITELN(SQRT(Y)) end;
```

in den String s_9.
Wir erhalten die Teilstrings s_1' und s_9' mit

s_1' =
```
program PI6X(INPUT,OUTPUT);
var I : integer; Y : real; procedure AA;
begin WRITE('
```

s_9' =
```
begin
for I:=1 to 1Ø do
begin READ(Y);WRITELN(SQRT(Y)) end;
```

```
AA;AA;AC;B; ..... AB; WRITELN
end end.
```

Es liegt auf der Hand, daß die Ersetzung von s_1 und s_9 durch s_1' bzw. s_9' in Programm π_6 wieder zu einem syntaktisch korrekten selbstreproduzierenden Programm $\tilde{\pi}_0$ führt. $\tilde{\pi}_0$ führt zuerst die for-Schleife von π_0 aus und reproduziert sich anschließend selbst. Es gilt für die von $\tilde{\pi}_0$ realisierte Funktion $f_{\tilde{\pi}_0} : A^*_{PAS} \longrightarrow A^*_{PAS}$:

$$f_{\tilde{\pi}_0}(x) = f_{\pi_0}(x) \circ \pi_0 \quad \text{für alle } x \in A^*_{PAS}$$

Damit genügt $\tilde{\pi}_0$ Definition (5.1.4)(i), und es gilt: $\tilde{\pi}_0$ ist eine selbstreproduzierende Version von π_0. Anhang A.11. zeigt Programm π_6' in implementierter Form, die aus der implementierten Form von Programm π_6 abgeleitet ist.

Die Konstruktion von $\tilde{\pi}_0$ aus den beiden Programmen π_6 und π_0 zeigte keine Aspekte, die darauf hindeuten, daß diese Konstruktion von irgendwelchen speziellen Eigenschaften von π_0 abhängt. Die Konstruktion müßte sich also für beliebige PASCAL-Programme verallgemeinern lassen. Um diese Verallgemeinerung komfortabel durchführen zu können, benötigen wir noch zwei Vereinbarungen:

In [10] ist eine kontextfreie Grammatik G_{PAS}, soweit dies überhaupt möglich ist (vergleiche 2.3.), für die Programmiersprache PASCAL angegeben worden. Wir werden uns im folgenden an dieser Grammatik orientieren.

(5.2.2) Vereinbarung: Sei ν ein nichtterminales Zeichen aus G_{PAS} und π ein gültiges PASCAL-Programm, so bezeichnen wir mit $\nu\pi$ den Teilstring des Programmtextes π, der sich aus dem nichtterminalen Zeichen ν ableiten läßt. Kommt das Zeichen ν im Ableitungsbaum von π nicht vor, so identifizieren wir $\nu\pi$ mit dem leeren Wort.

Der Teilstring $\nu\pi$ ist natürlich abhängig von der Position von ν im Ableitungsbaum von π . Dieser Um-

stand wird für uns aber keine Bedeutung erlangen.

Durch die hier eingeführte Schreibweise lassen sich aus den Produktionen der Grammatik G_{PAS} Gleichungen gewinnen.

Beispiel: G_{PAS} enthält die Produktion

⟨program⟩::=⟨program heading⟩⟨block⟩

Für jedes gültige PASCAL-Programm π gilt damit die Gleichung

π = ⟨program⟩π
= ⟨program heading⟩π ⟨block⟩π

Bei der Kombination von Programm π_6 und Programm π_0 zu $\tilde{\pi}_0$ waren keine Konflikte mit Bezeichnern aufgetreten. Das heißt, sämtliche Prozedurnamen von π_6 waren von den Variablennamen aus π_0 verschieden. Dies kann i.a. jedoch nicht vorausgesetzt werden. Um den Test, ob in einem PASCAL-Programm π Bezeichner auftreten, die mit einem Prozedurnamen aus π_6 identisch sind, zu vereinfachen, normieren wir die Prozedurnamen aus π_6, indem wir festlegen, daß jeder Prozedurname aus π_6 nur mit Hilfe des Buchstaben A gebildet werden darf. Alle Prozedurnamen aus π_6 sind also Elemente aus $\{A\}^+$ und unterscheiden sich nur in ihrer Länge. Wir werden später sehen, daß es bei einigen Programmen nötig ist, neue Prozeduren zu generieren. Die Namen dieser Prozeduren werden wir ebenfalls aus $\{A\}^+$ wählen. Durch diese Normierung der Prozedurnamen werden einige Namen sehr lang. Um Schreibarbeit zu sparen, treffen wir die folgende Vereinbarung.

(5.2.3) Vereinbarung:

(i) Sei $\underbrace{A \ldots\ldots A}_{k\text{-mal}}$ ein Element aus $\{A\}^+$,

dann schreiben wir statt dessen kürzer A^k.

(ii) Wir kürzen r aufeinanderfolgende Aufrufe

$\underbrace{A^j; \ldots\ldots ;A^j;}_{r\text{-mal}}$ der Prozedur A^j ab durch $(A^j;)^r$.

Unter Benutzung von Vereinbarung (5.2.3) präsentiert sich π_6 nach der Umbenennung der Prozedurnamen in der folgenden Weise:

π_6 =

```
program PI6(OUTPUT);
procedure A;begin WRITE('PROGRAM PI6(OUTPUT);PROCEDURE A;BE
   GIN WRITE(''') end;
procedure A2;begin WRITE('PROCEDURE ') end;
procedure A3;begin WRITE(';BEGIN WRITE(''') end;
procedure A4;begin WRITE(''') END;') end;
procedure A5;begin WRITE('''') end;
procedure A6;begin WRITE('A') end;
procedure A7;begin WRITE('BEGIN A;A;A5;A4;A2;(A6;)2A3;A2;A4
   ;A2;(A6;)3A3;A3;A5;A4;A2;(A6;)4A3;A5;A4;A4;A2;(A6;)5A3;A
   5;A5;A4;A2;(A6;)6A3;A6;A4;A2;(A6;)7A3;A7;A4;A7;WRITELN E
   ND.') end;
begin
A;A;A5;A4;
A2;(A6;)2A3;A2;A4;
A2;(A6;)3A3;A3;A5;A4;
A2;(A6;)4A3;A5;A4;A4;
A2;(A6;)5A3;A5;A5;A4;
A2;(A6;)6A3;A6;A4;
A2;(A6;)7A3;A7;A4;A7;
WRITELN
end.
```

Nach diesen Vorbemerkungen sind wir nun in der Lage, den Selbstreproduktionssatz für PASCAL-Programme zu beweisen.

(5.2.4) Satz (Selbstreproduktionssatz für PASCAL-Programme)

Zu jedem syntaktisch korrekten PASCAL-Programm π existiert eine selbstreproduzierende Version $\tilde{\pi}$.

Beweis: Der Beweis gliedert sich in zwei Teile. In Teil A wird eine Konstruktion für ein Programm $\tilde{\pi}$ für beliebiges π angegeben. In Teil B wird gezeigt, daß das so konstruierte $\tilde{\pi}$ eine selbstreproduzierende Version von π ist. Der Beweis setzt die Grammatik G_{PAS} voraus.

Sei nun π ein beliebiges gültiges PASCAL-Programm. Enthält π Bezeichner aus $\{A\}^+$, so werden diese Bezeichner durch andere Bezeichner ersetzt, die nicht aus $\{A\}^+$ sind. Das resultierende Programm ist textuell von π verschieden und wird mit π' bezeichnet. Enthält π keine Bezeichner aus $\{A\}^+$,[1] so wird $\pi' := \pi$ gesetzt.

Teil A:

Nach G_{PAS} hat π' den folgenden Aufbau:

$$\pi' = \langle\text{program heading}\rangle\pi' \langle\text{block}\rangle\pi',$$

wobei $\langle$program heading$\rangle\pi'$ und $\langle$block$\rangle\pi'$ ungleich dem leeren Wort sind. Entsprechendes gilt für π_6 und das zu konstruierende $\tilde{\pi}$:

$$\pi_6 = \langle\text{program heading}\rangle\pi_6 \langle\text{block}\rangle\pi_6$$

$$\tilde{\pi} = \langle\text{program heading}\rangle\tilde{\pi} \langle\text{block}\rangle\tilde{\pi}$$

Wir erhalten das Programm $\tilde{\pi}$, indem wir

$\langle$program heading$\rangle\tilde{\pi}$ aus $\langle$program heading$\rangle\pi_6$
und $\langle$program heading$\rangle\pi'$

bzw.

$\langle$block$\rangle\tilde{\pi}$ aus $\langle$block$\rangle\pi_6$
und $\langle$block$\rangle\pi'$

kombinieren.

---

1) Menge aller Wörter, die aus endlich vielen A zusammengesetzt sind.

(i) Kombination von ⟨program heading⟩ $\tilde{\pi}$

Aus G_{PAS} folgt, daß für ⟨program heading⟩π' die allgemeine Beziehung

⟨program heading⟩π' = program $\mu(\mu_1,\ldots,\mu_r)$;

wobei $\mu,\mu_1,\ldots,\mu_r, r \geq 0$, PASCAL-gültige Bezeichner sind,

gilt. μ stellt den Namen des Programms dar, die μ_i bezeichnen die von π' benutzten Dateien. Benutzt π' die Standarddatei OUTPUT, so sei o.B.d.A. μ_r = OUTPUT.

Für π_6 gilt:

⟨program heading⟩ π_6 = program PI6(OUTPUT);

Damit kombinieren wir:

⟨program heading⟩ $\tilde{\pi}$ = program P($\mu_1,\ldots,\mu_k$,OUTPUT);

mit $$k = \begin{cases} r-1, & \text{falls } \mu_r = \text{OUTPUT} \\ r & \text{sonst.} \end{cases}$$

(ii) Kombination von ⟨block⟩ $\tilde{\pi}$

Aus G_{PAS} folgt, daß für ⟨block⟩π' gilt:

⟨block⟩π' = ⟨label declaration part⟩ π'
⟨constant declaration part⟩ π'
⟨type definition part⟩ π'
⟨variable declaration part⟩ π'
⟨procedure and function declaration part⟩ π'
⟨statement part⟩ π'

Alle Strings bis auf ⟨statement part⟩π' können leer sein.
Die Programme π' und $\tilde{\pi}$ haben einen entsprechenden Aufbau.
Da ⟨label declaration part⟩ π_6,...,⟨variable declaration part⟩ π_6 gleich dem leeren Wort sind, setzen wir:

⟨label declaration part⟩ $\tilde{\pi}$:= ⟨label declaration part⟩ π'
⟨constant declaration part⟩ $\tilde{\pi}$:= ⟨constant declaration part⟩ π'
⟨type definition part⟩ $\tilde{\pi}$:= ⟨type definition part⟩ π'
⟨variable declaration part⟩ $\tilde{\pi}$:= ⟨variable declaration part⟩ π'

In 3.2.5. war bemerkt worden, daß es in SIMULA nicht möglich ist, Texte, die das Zeichen " enthalten, en bloc auszudrucken. Die gleichen Schwierigkeiten macht in PASCAL das Zeichen '. Enthält π' das Zeichen ' ein- oder mehrmals, so muß der Text π' entsprechend zerlegt werden.

Es wird gesetzt:

S:= ⟨label declaration part⟩π'
⟨constant declaration part⟩π'
⟨type definition part⟩π'
⟨procedure and function declaration part⟩π'

T:= (⟨statement part⟩π' ohne die klammernden terminalen Zeichen begin und end.)

Mittels der Strings S und T läßt sich der Programmtext π' wie folgt darstellen:

π' = ⟨program heading⟩$\pi' \circ S \circ$ begin $\circ T \circ$ end.

1.Fall: S ist ungleich dem leeren Wort (d.h. π hat einen nichtleeren Vereinbarungsteil)

Zerlegung von S in eine Folge von $n \geq 1$ Teilstrings s_i mit

(i) $s_i = '$ oder $s_i \neq '$, $i \in [n]$

(ii) $s_i \neq ' \implies s_{i+1} = '$, $i \in [n-1]$

(iii) $s_1 \circ s_2 \circ \ldots\ldots \circ s_n = S$

Sei p die Anzahl der Teilstrings von S, die ungleich ' sind. Es gilt: $1 \leq p < n$
Für jeden Teilstring ungleich ' , mit Ausnahme von s_1, wird eine Prozedur generiert:

procedure A^{7+j-1};begin WRITE('s_i') end; $j = 2,\ldots,p$

wobei s_i der j-te Teilstring ungleich ' ist.
Sei AP die Menge der Namen der generierten Prozeduren.
Dann gilt:

$AP = \{A^{7+1}, A^{7+2}, \ldots\ldots, A^{7+p-1}\}$

Sei $\mathcal{S} = \{s_1, \ldots\ldots, s_n\}$ die Menge der Teilstrings der Zerlegung von S.

Einführung der Funktionen σ und $\tilde{\sigma}$:

$$\sigma : [p] \longrightarrow \mathcal{S}$$
$$j \longmapsto s_k \text{ , mit } s_k \text{ ist der j-te Teilstring} \neq '$$

$$\tilde{\sigma} : [n] \longrightarrow AP \cup \{A^5\}$$
$$j \longmapsto \begin{cases} \perp, \text{ falls } j=1 \text{ [1)]} \\ A^{7+i-1}, \text{ falls } s_j \text{ der i-te Teilstring von } S \neq ' \text{ ist} \\ A^5, \text{ falls } s_j = ' \end{cases}$$

(Zur Erinnerung: A^5 ist die Prozedur, die in π_6 das Zeichen ' ausgibt.)

Da weder ⟨label declaration part⟩ π'
noch ⟨constant declaration part⟩ π'
noch ⟨type definition part⟩ π'
noch ⟨procedure and function declaration part⟩ π'

mit dem Zeichen ' anfangen oder enden können, gilt:

$$\sigma(1) = s_1 \quad \text{und} \quad \sigma(p) = s_n$$

Der String T wird in einen String T' transformiert, indem T mit dem Zeichen ; konkateniert wird: $T' := T;$
T' wird analog zu S in $m \geq 1$ Teilstrings t_i zerlegt. Für die Zerlegung von T' gilt:

(i) $t_i = '$ oder $t_i \neq '$, $i \in [m]$
(ii) $t_i \neq ' \implies t_{i+1} = '$ $i \in [m-1]$
(iii) $t_1 \circ t_2 \circ \ldots\ldots \circ t_m = T'$

Sei q die Anzahl der Teilstrings von T', die ungleich ' sind. Es gilt: $1 \leq q < m$.
Für jeden Teilstring t_i ungleich ', mit Ausnahme von t_m, wird eine Prozedur generiert:

---

1)Notation: $\perp$ bedeutet undefiniert.

procedure $A^{7+j+p-1}$;begin WRITE('t_i') end;
für j = 2,...,q-1, wobei t_i der j-te Teilstring ≠ ' ist

bzw.

procedure A^{7+p};begin WRITE('BEGIN t_j') end;
für j = 1.

Dabei ist t_j der j-te Teilstring von T' ungleich '.

Entsprechend AP, φ, σ und $\tilde{\sigma}$ werden AQ, $\mathcal{T}$, τ und $\tilde{\tau}$ definiert:

$AQ := \{A^{7+p}, \ldots\ldots, A^{7+p+q-1}\}$

$\mathcal{T} := \{t_1, \ldots\ldots t_m\}$

$\tau : [q] \longrightarrow \{t_i\}_{i \in [n]}$

$j \longmapsto t_k$, mit t_k ist der j-te Teilstring von T' $\neq$ '

$\tilde{\tau} : [m] \longrightarrow AQ \cup \{A^5\}$

$$j \longmapsto \begin{cases} \perp, \text{ falls } j = q \\ A^{7+p+i-1}, \text{ falls } t_j \text{ der i-te Teilstring} \neq \text{ ' ist} \\ A^5, \text{ falls } t_j = \text{ ' } \end{cases}$$

Das erste Zeichen nach dem einleitenden begin von ⟨statement part⟩π' kann nicht gleich ' sein. Daraus ergibt sich nach Definition von T': $\tau(1) = t_1$
Das letzte Zeichen von T' ist ; . Daraus folgt: $\tau(q) = t_m$

Es ist nun möglich, die beiden noch fehlenden Programmteile von π, nämlich

⟨procedure and function declaration part⟩π und
⟨statement part⟩π

anzugeben:

⟨procedure and function declaration part⟩ $\tilde{\pi}$
= ⟨procedure and function declaration part⟩ π'
procedure A^{7+1};begin ..... end;
⋮
procedure $A^{7+p+q-1}$;begin ..... end;
procedure A;begin WRITE('⟨program heading⟩ $\tilde{\pi}$ $\sigma(1)$') end;
⋮
procedure A^{7};begin WRITE('$\tau(q)$ ⊛ END.') end;

⟨statement part⟩ $\tilde{\pi}$
= begin $t_1,\ldots,t_m$ ⊛ end.

⊛ steht dabei als Abkürzung für die Aufruffolge der Prozeduren A bis A^{p+q+2}, die die Ausgabe von $\tilde{\pi}$ bewirkt.
Damit ergibt sich insgesamt:

$\tilde{\pi}$ = program P($\mu_1,\ldots,\mu_k$,OUTPUT);
s_1 ..... s_n
procedure A^{7+1};begin WRITE('$\sigma(2)$') end;
procedure A^{7+2};begin WRITE('$\sigma(3)$') end;
.....
procedure A^{7+p-1};begin WRITE('$\sigma(p)$') end;
procedure A^{7+p};begin WRITE('BEGIN $\tau(1)$') end;
procedure A^{7+p+1};begin WRITE('$\tau(2)$') end;
.....
procedure $A^{7+p+q-2}$;begin WRITE('$\tau(q-1)$') end;
procedure A;begin WRITE('PROGRAM P($\mu_1,\ldots,\mu_k$,OUTPUT);$\sigma(1)$'
) end;
procedure A^{2};begin WRITE('PROCEDURE ') end;
procedure A^{3};begin WRITE(';BEGIN WRITE('''') end;
procedure A^{4};begin WRITE('''') END;') end;
procedure A^{5};begin WRITE('''') end;
procedure A^{6};begin WRITE('A') end;
procedure A^{7};begin WRITE('$\tau(q)$ ⊛ END.') end;
begin
t_1 ..... t_m

```
A;
ẽ(2); ..... ;ẽ(n);
A2;(A6;)7+1 A3;A7+1;A4;
A2;(A6;)7+2 A3;A7+2;A4;
.....
A2;(A6;)7+p-1 A3;A7+p-1;A4;
A2;(A6;)7+p A3;A7+p;A4;
A2;(A6;)7+p+1 A3;A7+p+1;A4;
.....
A2;(A6;)7+p+q-2 A3;A7+p+q-2;A4;
A2;A6;A3;A;A4;
A2;(A6;)2 A3;A2;A4;
A2;(A6;)3 A3;A3;A5;A4;
A2;(A6;)4 A3;A5;A4;A4;
A2;(A6;)5 A3;A5;A5;A4;
A2;(A6;)6 A3;A6;A4;
A2;(A6;)7 A3;A7;A4;
π̃(1);.....;π̃(m-1);
A7;WRITELN
end.
```

Zeilennummer

k_{7+1}
k_{7+2}
....
k_{7+p-1}
k_{7+p}
k_{7+p+1}
.....
$k_{7+p+q-2}$
k_1
:
k_7

2.Fall: Der String S ist gleich dem leeren Wort (d.h. der Vereinbarungsteil von π ist leer)

Dieser Fall ist ein Spezialfall von Fall 1. Wir erhalten das resultierende Programm aus dem in Fall 1 angegebenen Programm $\tilde{\pi}$ durch Streichung der den String S betreffenden Konstruktion. Im Einzelnen

- Streichung der Prozeduren A^{7+1} bis A^{7+p-1}
- Änderung der Prozedur A in
  procedure A;begin WRITE('PROGRAM P(μ_1,...,μ_k,OUTPUT);')
  end;
- Streichung der Programmzeile mit dem Inhalt
  ẽ(2);.....;ẽ(n);
- Streichung der Programmzeilen
  k_{7+1} bis k_{7+p-1}

Teil B:

1.Fall: S ist ungleich dem leeren Wort.

Die Konstruktion in Teil A liefert ein syntaktisch korrektes PASCAL-Programm $\tilde{\pi}$. Es bleibt zu zeigen, daß $\tilde{\pi}$ eine selbstreproduzierende Version von π ist.

π realisiert eine Funktion

$$f_\pi : (A_{PAS}^*)^r \longrightarrow (A_{PAS}^*)^u \quad \text{mit} \quad o \leq u \leq r.$$

Der Vereinbarungsteil von π wird in Form des Textes $S = s_1 \ldots\ldots s_n$ in das Programm $\tilde{\pi}$ unverändert aufgenommen. Im Anweisungsteil von $\tilde{\pi}$ wird der in der Form $T' = t_1 \ldots\ldots t_m$ übernommene Anweisungsteil von π zuerst ausgeführt. Alle anderen Anweisungen sind nur Aufrufe von Prozeduren, die nicht in S vereinbart sind. Bei jedem Aufruf dieser Prozeduren wird genau eine Textkonstante auf die Datei OUTPUT ausgegeben. Da $\tilde{\pi}$ nur endlich viele Prozeduraufrufe aufweist, wird insgesamt ein endlicher Text, also ein Wort y aus A_{PAS}^*, auf die Datei OUTPUT ausgegeben. Die Aufrufe dieser Ausgabeprozeduren erfolgen jedoch erst, nachdem der Anweisungsteil T' abgearbeitet worden ist. $\tilde{\pi}$ realisiert also die folgende Funktion:

$$f_{\tilde{\pi}} : (A_{PAS}^*)^r \longrightarrow (A_{PAS}^*)^u \qquad o \leq u \leq r$$

mit

$$(f_{\tilde{\pi}})_i(\overline{x}) = \left\{ \begin{array}{ll} (f_\pi)_i(\overline{x}) & \text{für } i \in [r-1] \\ (f_\pi)_i \circ y & \text{für } i = r \end{array} \right\} \quad \overline{x} \in (A_{PAS}^*)^r$$

falls μ_r = OUTPUT, d.h. $\tilde{\pi}$ gibt y auf eine Ausgabedatei aus, die auch π benutzt.

bzw.

$$f_{\tilde{\pi}} : (A_{PAS}^*)^{r+1} \longrightarrow (A_{PAS}^*)^u \qquad o \leq u \leq r+1$$

mit

$$(f_{\tilde{\pi}})_i(\overline{x}) = \left\{ \begin{array}{ll} (f_\pi)_i(\overline{x}) & \text{für } i \in [r] \\ y & \text{für } i = r+1 \end{array} \right\} \quad \overline{x} \in (A_{PAS}^*)^{r+1}$$

falls $\mu_r \neq$ OUTPUT, d.h. $\tilde{\pi}$ gibt y auf die Datei OUTPUT

aus, die von π nicht als Ausgabedatei benutzt wird.

$\tilde{\pi}$ erfüllt also genau dann Definition (5.1.4), wenn der Text $\tilde{\pi}$ Teilstring von y ist.

Es gilt aber sogar $y = \tilde{\pi}$, denn:

Nach Abarbeitung von t_1 ..... t_m gibt $\tilde{\pi}$ zunächst durch Aufruf der Prozedur A seinen eigenen Programmkopf ⟨program heading⟩$\tilde{\pi}$ und s_1 aus. Die nächsten Prozeduraufrufe

$\tilde{\sigma}(2)$; ... bis ... ;$\tilde{\sigma}(n)$; erledigen die Ausgabe von s_2 ... bis ... s_n.

Die Prozeduraufrufe der Zeilen k_{7+1} bis $k_{7+p+q-2}$ und k_1 bis k_7 bewirken die Ausgabe der Prozedurvereinbarungen von A^{7+1} bis $A^{7+p+q-2}$ bzw. von A bis A^7. Es fehlt nur noch die Ausgabe von ⟨statement part⟩$\tilde{\pi}$. Diese Ausgabe wird jedoch durch die Folge

$\tilde{\tau}(1)$; .....;$\tilde{\tau}(m-1)$;A^7; bewirkt. Das nachfolgende WRITELN leert nur den Puffer der Datei OUTPUT.

Es gilt also $y = \tilde{\pi}$. Damit ist $\tilde{\pi}$ selbstreproduzierende Version von π.

2.Fall: S ist gleich dem leeren Wort

Fall 2 ergibt sich als Spezialfall von Fall 1. Es gilt somit auch in diesem Fall, daß das erhaltene Programm $\tilde{\pi}$ selbstreproduzierende Version von π ist.

Damit ist der Satz vollständig bewiesen.

%

Dem Beweis von Satz (5.2.4) läßt sich direkt ein Algorithmus entnehmen, der zu jedem beliebigen PASCAL-Programm π eine selbstreproduzierende Version $\tilde{\pi}$ findet.

(5.2.5) Algorithmus:

Eingabe: Das PASCAL-Programm π.

1.Schritt: Test, ob keiner der in π vorkommenden Bezeichner aus $\{A\}^+$ ist. Gegebenenfalls Umbenennung vornehmen.

2.Schritt: Formulierung von ⟨program heading⟩$\tilde{\pi}$ aus ⟨program heading⟩π unter Verwendung der Standarddatei OUTPUT.

3.Schritt: Zerlegung von

S = ⟨label declaration part⟩π
⟨constant declaration part⟩π
⟨type definition part⟩π
⟨variable declaration part⟩π

in Teilstrings $s_1,\ldots,s_n$ und Ermittlung der Anzahl p der Teilstrings s_j ungleich ' .
Anschließend Formulierung der p-1 Prozeduren A^{7+1} bis A^{7+p-1} und Aufstellung der Wertetabellen von σ und $\tilde{\sigma}$.

4.Schritt: Ist das letzte Zeichen von T = (⟨statement part⟩π ohne klammerndes begin end.) gleich ; , so wird T' :=T gesetzt, andernfalls T' :=T;.
Zerlegung von T' in $t_1 \ldots t_m$ und Ermittlung der Zahl q. Anschließend Formulierung der q-1 Prozeduren A^{7+p} bis $A^{7+p+q-2}$ und Aufstellung der Wertetabellen von τ und $\tilde{\tau}$.

5.Schritt: Einsetzen der erhaltenen Funktionswerte, Prozedurenund Teilstrings $s_1,\ldots,s_n,t_1,\ldots,t_m$ in das im Beweis angegebene Programmschema.

Aufwand: Der Algorithmus verhält sich linear zur Länge $l(\pi)$ des Programms π .

(5.2.6) Beispiel: Gegeben sei das in [28] Seite 17 zu findende Programm.

```
π = program CONVERT(OUTPUT);
    const ADDIN=32;MULBY=1.8;LOW=Ø;HIGH=39;
          SEPARATOR='----------';
    var DEGREE : LOW .. HIGH;
    begin
    WRITELN(SEPARATOR);
    for DEGREE:= LOW to HIGH do
```

```
begin WRITE(DEGREE,'C',ROUND(DEGREE*MULBY+ADDIN),'F');
      if ODD(DEGREE) then WRITELN
end;
WRITELN;
WRITELN(SEPARATOR)
end.
```

Anwendung von Algorithmus (5.2.5):

1.Schritt: π enthält keinen Bezeichner aus $\{A\}^+$. Es sind also keine Umbenennungen nötig.

2.Schritt: ⟨program heading⟩$\tilde{\pi}$ wird auf

```
program CONVERTX(OUTPUT);
```

gesetzt.

3.Schritt: $S = s_1\ s_2\ s_3\ s_4\ s_5$ mit

s_1 = `const ADDIN=32;MULBY=1.8;LOW=Ø;HIGH=39;SEPARATOR=`

s_2 = `'`

s_3 = `----------`

s_4 = `'`

s_5 = `;var DEGREE:LOW..HIGH;`

Es gilt n=5, p=3

Die resultierenden Prozeduren sind:

```
procedure A8;begin WRITE(' ----------') end;
procedure A9;begin WRITE(';VAR DEGREE:LOW..
    HIGH;') end;
```

Wertetabellen von σ und $\tilde{\sigma}$:

$\sigma(1) = s_1$	$\tilde{\sigma}(1) = \perp$
$\sigma(2) = s_3$	$\tilde{\sigma}(2) = A^5$
$\sigma(3) = s_5$	$\tilde{\sigma}(3) = A^8$
	$\tilde{\sigma}(4) = A^5$
	$\tilde{\sigma}(5) = A^9$

4.Schritt: $T = t_1t_2t_3t_4t_5t_6t_7t_8t_9$ mit

t_1 = `WRITELN(SEPARATOR);FOR DEGREE:=LOW TO HIGH DO BEGIN WRITE('DEGREE,`

t_2 = `'`

t_3 = `C`

t_4 = '
t_5 = ,ROUND(DEGREE*MULBY+ADDIN),
t_6 = '
t_7 = F
t_8 = '
t_9 =);IF ODD(DEGREE) THEN WRITELN END;
WRITELN;WRITELN(SEPARATOR);

Es gilt: m=9 , q=5

Die resultierenden Prozeduren sind:

```
procedure A10;begin WRITE('BEGIN WRITELN(SEPARA
   TOR);FOR DEGREE:=LOW TO HIGH DO BEGIN WRITE(
   DEGREE),') end;
procedure A11;begin WRITE('C') end;
procedure A12;begin WRITE('ROUND(DEGREE*MULBY+A
   DDIN),') end;
procedure A13;begin WRITE('F') end;
```

Wertetabellen von τ und $\tilde{\tau}$:

$\tau(1) = t_1$	$\tilde{\tau}(1) = A^{10}$	$\tilde{\tau}(6) = A^5$
$\tau(2) = t_3$	$\tilde{\tau}(2) = A^5$	$\tilde{\tau}(7) = A^{13}$
$\tau(3) = t_5$	$\tilde{\tau}(3) = A^{11}$	$\tilde{\tau}(8) = A^5$
$\tau(4) = t_7$	$\tilde{\tau}(4) = A^5$	$\tilde{\tau}(9) = \perp$
$\tau(5) = t_9$	$\tilde{\tau}(5) = A^{12}$	

5.Schritt:

$\tilde{\pi}$ =
```
program CONVERTX(OUTPUT);
   const ADDIN=32;MULBY=1.8;LOW=Ø;HIGH=39;SEPARATOR=' ___
         _______';
var DEGREE:LOW..HIGH;
procedure A8;begin WRITE(' __________') end;
procedure A9;begin WRITE(';VAR DEGREE:LOW..HIGH;') end;
procedure A10;begin WRITE('BEGIN WRITELN(SEPARATOR);FOR
   DEGREE:=LOW TO HIGH DO BEGIN WRITE(DEGREE)') end;
procedure A11;begin WRITE('C') end;
procedure A12;begin WRITE(',ROUND(DEGREE*MULBY+ADDIN),'
   ) end;
procedure A13;begin WRITE('F') end;
procedure A;begin WRITE('PROGRAM CONVERTX(OUTPUT);CONST
   ADDIN=32;MULBY=1.8;LOW=Ø;HIGH=39;SEPARATOR=') end;
```

```
procedure A2;begin WRITE('PROCEDURE ') end;
procedure A3;begin WRITE(';BEGIN WRITE('''') end;
procedure A4;begin WRITE('''') END;') end;
procedure A5;begin WRITE('''') end;
procedure A6;begin WRITE('A') end;
procedure A7;begin WRITE(');IF ODD(DEGREE) THEN WRITEL
   N END;WRITELN;WRITELN(SEPARATOR); ⊛ END.') end;
begin
WRITELN(SEPARATOR);
for DEGREE:= LOW to HIGH do
begin WRITE(DEGREE,'C',ROUND(DEGREE*MULBY+ADDIN),'F');
      if ODD(DEGREE) then WRITELN
end;
WRITELN;
WRITELN(SEPARATOR);
A;
A5;A8;A5;A9
A2;(A6;)8 A3;A8;A4;
A2;(A6;)9 A3;A9;A4;
A2;(A6;)10 A3;A10;A4;
A2;(A6;)11 A3;A11;A4;
A2;(A6;)12 A3;A12;A4;
A2;(A6;)13 A3;A13;A4;
A2;A6;A3;A;A4;
A2;(A6;)2 A3;A2;A4;
A2;(A6;)3 A3;A5;A3;A4;
A2;(A6;)4 A3;A5;A4;A4;
A2;(A6;)5 A3;A5;A5;A4;
A2;(A6;)6 A3;A6;A4;
A2;(A6;)7 A3;A7;A4;
A10;A5;A11;A5;A12;A5;A13;A5;A7;
WRITELN
end.
```

⊛

Die Bedeutung von Satz (5.2.4) ist in erster Linie theoretischer und nicht praktischer Art. Satz (5.2.4) beweist zwar die Existenz einer selbstreproduzierenden Version $\tilde{\pi}$ für jedes gültige PASCAL-Programm π und gibt auch eine Konstruktion für ein syntaktisch richtiges $\tilde{\pi}$ an, jedoch garan-

tiert die syntaktische Korrektheit noch nicht die Realisierbarkeit von $\tilde{\pi}$ auf einer konkreten Rechenmaschine.

Die Implementierung eines von Algorithmus (5.2.5) gelieferten Programms $\tilde{\pi}$ kann zu folgenden Schwierigkeiten führen:

(i) Der längste Prozedurname von $\tilde{\pi}$ ist $A^{7+p+q-2}$. $A^{7+p+q-2}$ ist p+q+5 Zeichen lang. Jedes dieser Zeichen ist signifikant, da auch der Prozedurname $A^{7+p+q-3}$ mit der Länge p+q+4 in $\tilde{\pi}$ vorkommt. Die zulässige Signifikanz von Bezeichnern ist bei implementierten PASCAL-Compilern aber beschränkt. Die Zahlen p und q sind jedoch nur endlich, was zur Folge hat, daß bei großen p und q einige Prozeduren nicht unterschieden werden können.

(ii) Die Länge der durch (5.2.5) entstehenden Textkonstanten in den Prozeduren A^j, $j \in [p+q-2]$, ist für die Gesamtheit aller PASCAL-Programme $\tilde{\pi}$ nicht beschränkt. Damit ist auch die Länge einer Programmzeile nicht beschränkt. Die Länge einer Programmzeile ist auf realen Rechenanlagen aber oft durch die Länge des Eingabepuffers des implementierten PASCAL-Compilers beschränkt.

Die Schwierigkeiten (i) und (ii) lassen sich bei vielen in der Praxis vorkommenden Programmen vermeiden, indem man Algorithmus (5.2.5) um zwei „praxisorientierte" Schritte ergänzt.

<u>6.Schritt:</u> Sei g die Anzahl der signifikanten Zeichen von Bezeichnern bei dem vorliegenden PASCAL-Compiler. Sei $a := 7+p+q-2 = p+q+5$ die Länge des Prozedurnamen A^{p+q+5}. a ist gleichzeitig die Anzahl der Prozedurnamen vom Typ A^j, $j \in [a]$. Dann wähle man zwei natürliche Zahlen $c \leq 26$ und $b \leq g$ mit

$$(1) \qquad \sum_{k=1}^{b} c^k \geq a$$

Nun können die Prozedurnamen A^1 bis $A^{7+p+q-2}$ durch neue Namen ersetzt werden, die maximal b Zeichen lang

sind und aus den c ersten Buchstaben des Alphabets zusammengesetzt sind. Außer dem Buchstaben A können also c-1 weitere Buchstaben zur Bildung von Prozedurnamen herangezogen werden. Zu diesen c-1 Buchstaben müssen aber c-1 neue Prozeduren generiert werden, die jeweils einen neuen Buchstaben ausdrucken:

```
procedure ...;begin WRITE('B') end;
procedure ...;begin WRITE('C') end;   } c-1
          :
          :
          :
```

Für diese Prozeduren werden aber auch Namen benötigt. Daher müssen die Zahlen c und b auch die Relation

(1') $\sum_{k=1}^{b} c^k \geq a+c-1$ erfüllen.

7.Schritt: Sei d die Eingabepufferlänge des zur Verfügung stehenden PASCAL-Compilers. Seien v_i, $i \in [p+q-2]$, die Textkonstanten der Prozeduren A^i (diese Prozeduren sind möglicherweise in Schritt 6 bereits umbenannt worden). Für jedes $i \in [p+q-2]$ muß die folgende Berechnung ausgeführt werden:

Ist $l(\text{WRITE}(' v_i '))$ $\leq$ d,[1] so bleibt A^i unverändert. Andernfalls wird v_i in k_i Teilstrings v_{ij} zerlegt mit $l(\text{WRITE}(' v_{ij} ')) \leq d, \forall\ j \in [k_i]$. Dabei sei k_i minimal gewählt.

Die Prozedur A^i wird durch k_i neue Prozeduren ersetzt:

```
procedure ...;begin WRITE(' v_i1 ') end;
          :                          :
procedure ...;begin WRITE(' v_ik_i ') end;
```

Entsprechend der neu eingeführten Prozeduren wird die Aufruffolge der Prozeduren, die die Ausgabe von $\tilde{\pi}$ bewirkt, ergänzt.

---

[1]) vgl. Fußnote auf Seite 19

Die Schritte 6 und 7 führen jedoch nicht immer zum Ziel.

Ist die Eingabepufferlänge d des zur Verfügung stehenden PASCAL-Compilers relativ gering, so werden in Schritt 7 in der Regel sehr viele neue Prozeduren generiert. Sicherlich wird dabei auch die Textkonstante der - möglicherweise in Schritt 6 umbenannten - Prozedur A^7, die den Ausgabealgorithmus für $\tilde{\pi}$ enthält, auf mehrere neue Prozeduren aufgespalten. Neue Prozeduren bedingen einen längeren Ausgabealgorithmus, wenn $\tilde{\pi}$ selbstreproduzierend bleiben soll. Das bedeutet aber, daß noch mehr Prozeduren zur Aufnahme des Ausgabealgorithmus nötig sind. Noch mehr Prozeduren bewirken aber eine erneute Verlängerung des Ausgabealgorithmus, was noch mehr Prozeduren bewirkt, u.s.w.
Ist d nun relativ gering, so kann es geschehen, daß sich dieser Prozeß nicht stabilisiert und Schritt 7 zu einem unendlichen Programm führt. Das ist genau dann der Fall, wenn die Textkonstanten der den Ausgabealgorithmus enthaltenden Prozeduren durchschnittlich weniger Prozeduraufrufe enthalten, als zur Ausgabe der Prozedurvereinbarung einer Ausgabeprozedur erforderlich ist. Das sprunghafte Anwachsen der Anzahl der Ausgabeprozeduren kann zu einer wiederholten Durchführung von Schritt 6 führen. Noch komplizierter werden die Verhältnisse, wenn die Ausgabe von $\tilde{\pi}$ formatiert werden soll.

(5.2.7) Beispiel: Das in Beispiel (5.2.6) enthaltene Programm $\tilde{\pi}$ soll implementiert werden.

6.Schritt: Programm $\tilde{\pi}$ weist 13 Prozedurnamen vom Typ A^i auf. Der vorhandene PASCAL-Compiler wertet nur die ersten 8 Zeichen eines Bezeichners als signifikant. Einige der A^i müssen also umbenannt werden. Die Prozeduren A^{11} und A^{13} stellen zufällig die beiden Buchstaben C und F zur Verfügung. Zur Umbenennung der 13 Prozeduren $A^1,\ldots,A^{13}$ sollen aber insgesamt 4 Buchstaben herangezogen werden. Es wird deshalb zusätzlich die Prozedur

```
procedure BB;begin WRITE('B') end;
```

in das Programm $\tilde{\pi}$ aufgenommen.

Mit den 4 Buchstaben A,B,C und F lassen sich
4 verschiedene Namen der Länge 1,
16 verschiedene Namen der Länge 2 und
64 verschiedene Namen der Länge 3 bilden.
Auch wenn Schritt 7 weitere Prozeduren liefern sollte, ist anzunehmen, daß mit c=4 und b=3 genügend viele Namen zur Verfügung stehen. Wir werden versuchen, mit Namen der Längen 1 und 2 auszukommen.

Die Prozeduren A^i, $i \in [13]$, werden wie folgt umbenannt:

A^1	in	AA	A^8	in	AF
A^2	in	A	A^9	in	CB
A^3	in	B	A^{10}	in	CC
A^4	in	C	A^{11}	in	BC
A^5	in	F	A^{12}	in	CF
A^6	in	BA	A^{13}	in	BF
A^7	in	AC			

7.Schritt: Die Länge jeder Programmzeile sei auf 132 Zeichen beschränkt. Damit braucht nur die Textkonstante der Prozedur AC auf mehrere Prozeduren aufgespalten zu werden. Der für diese Prozeduren notwendige Verwaltungsaufwand im Algorithmus von $\tilde{\pi}$ bewirkt, daß insgesamt 4 neue Prozeduren FA,FB,FC und FF eingeführt werden müssen.

Anhang A.12. zeigt das durch die Schritte 6 und 7 veränderte Programm $\tilde{\tilde{\pi}}$. Zur Formatierung der Ausgabe wurde in das Programm die aus 3.3.4. bekannte Prozedur Q eingefügt.

5.3.Selbstreproduktionssatz für die Programmiersprache SIMULA

Die in den Kapiteln 3 und 4 vorgestellten Beispielprogramme in SIMULA und PASCAL entsprechen sich weitgehend. Zum selbstreproduzierenden PASCAL-Programmm π_6 aus (3.3.2)

existiert ein nahezu identisches SIMULA-Programm π_4 in 3.2.7. . Da der Beweis von Satz (5.2.4) im wesentlichen auf der Existenz von π_6 beruht, ist anzunehmen, daß bezüglich der Programmiersprache SIMULA ein entsprechender Satz gilt. Der Beweis dieses Satzes wird sich wie der Beweis von (5.2.4) in zwei Teile A und B gliedern. In Teil A wird die Konstruktion der selbstreproduzierenden Version $\tilde{\pi}$ eines beliebigen SIMULA-Programms π erfolgen. Trotz der Entsprechung von π_4 und π_6 muß diese Konstruktion explizit angegeben werden, da SIMULA im Gegensatz zu PASCAL eine blockorientierte Programmiersprache ist. Das aus Teil A resultierende Programm $\tilde{\pi}$ wird aber weitgehend dem in Teil A von dem Beweis zu (5.2.4) konstruierten Programm entsprechen. Zum Nachweis, daß $\tilde{\pi}$ selbstreproduzierende Version von π ist, wird in Teil B ein Verweis auf Teil B vom Beweis zu (5.2.4) genügen. Der Beweis wird sich an der in [19] angegebenen SIMULA-Grammatik orientieren. Diese Grammatik sei mit G_{sim} bezeichnet. Für G_{SIM} übernehmen wir die Vereinbarung (5.2.2). Außerdem übernehmen wir Vereinbarung (5.2.3) für die Prozedurnamen von π_4. Damit präsentiert sich π_4 in der Form:

π_4 =

```
begin
procedure A;OUTTEXT("BEGIN PROCEDURE A;OUTTEXT("""");
procedure A2;OUTTEXT("PROCEDURE ");
procedure A3;OUTTEXT(" ;OUTTEXT("""");
procedure A4;OUTTEXT("""");");
procedure A5;OUTTEXT("""");
procedure A6;OUTTEXT("A");
procedure A7;OUTTEXT("A;A;A5;A4;A2;(A6;)2 A3;A2;A4;A2;(A6;)3
     A3;A3;A5;A4;A2;(A6;)4 A3;A5;A4;A4;A2;(A6;)5 A3;A5;A5;A4
    ;A2;(A6;)6 A3;A6;A4;A2;(A6;)7 A3;A7;A4;A7 END");
A;A;A5;A4;
A2;(A6;)2 A3;A2;A4;
A2;(A6;)3 A3;A3;A5;A4;
A2;(A6;)4 A3;A5;A4;A4;
A2;(A6;)5 A3;A5;A5;A4;
A2;(A6;)6 A3;A6;A4;
```

$A^2;(A^6;)^7\ A^3;A^7;A^4;A^7$
end

(5.3.1) Satz (Selbstreproduktionssatz für SIMULA-Programme)

Zu jedem syntaktisch korrekten SIMULA-Programm π existiert eine selbstreproduzierende Version $\tilde{\pi}$.

Beweis:

Teil A:

Das Programm π_4 hat laut Grammatik G_{SIM} den folgenden Aufbau:

π_4 = begin ⟨Vereinbarung⟩ -Folge π_4
⟨Anweisung⟩ -Folge π_4 end

Sei π ein beliebiges SIMULA-Programm. Dann hat π den Aufbau:

π = ⟨Klassenbezeichnung⟩ π
⟨akt. Parameterteil⟩ -option π
begin ⟨Vereinbarung⟩ -Folge π
⟨Anweisung⟩ -Folge π end

Dabei können ⟨Klassenbezeichnung⟩ π ,
⟨akt. Parameterteil⟩ -option π
bzw. ⟨Vereinbarung⟩ -Folge π

leer sein (vgl. π_4).
Besteht π nur aus

begin ⟨Anweisung⟩ -Folge π end ,

so nennt man π eine Verbundanweisung. Andernfalls ist π ein Block. Sowohl Verbundanweisungen als auch Blöcke sind an jeder Stelle von ⟨Anweisung⟩ -Folge π_4 zulässig, an der eine Anweisung zulässig ist. Dies wird im folgenden ausgenutzt.

Kombination von $\tilde{\pi}$ aus π_4 und π

Ist ⟨Klassenbezeichnung⟩π nicht leer, so ist zunächst zu testen, ob ⟨Klassenbezeichnung⟩π aus $\{A\}^+$ ist. Ist dies

der Fall, so wird ⟨Klassenbezeichnung⟩π umbenannt. Es resultiert das Programm π', das die gleiche Funktion realisiert wie π. Andernfalls wird $\pi' := \pi$ gesetzt. Wegen des Konzepts der lokalen Gültigkeit von Bezeichnern ist eine Umbenennung der in - falls vorhanden - ⟨Vereinbarung⟩-Folgeπ und ⟨Anweisung⟩-Folgeπ vorkommenden Bezeichner auf jeden Fall nicht nötig.

$\tilde{\pi}$ = **begin**
procedure A; ... ;
(zusätzliche Vereinbarungen)
procedure A^2; ...;
⋮
procedure A^7; ...;
π';
(zusätzliche Anweisungen)
⟨Anweisung⟩-Folge π_4
end

← mit geänderter Prozedur A^7 aus π_4

Es brauchen nur noch die Programmteile

(zusätzliche Vereinbarungen)

und (zusätzliche Anweisungen)

realisiert zu werden.

Sei T gleich π' mit angefügtem ; . Also T = π';. Der String T wird analog zum String T' aus dem Beweis zu (5.2.4) in eine Folge von $m \geq 1$ Teilstrings t_i zerlegt. Für diese Zerlegung von T gilt:

(i) $t_i = "$ oder $t_i \neq "$, $i \in [m]$

(ii) $t_i \neq " \Longrightarrow t_{i+1} = "$, $i \in [m-1]$

(iii) $t_1 \circ t_2 \circ \ldots \circ t_m = T$

Sei q die Anzahl der Teilstrings t_j von T, die ungleich " sind. Es gilt: $1 \leq q < m$.
Für jeden Teilstring t_j ungleich ", mit Ausnahme von t_m, wird eine Prozedur generiert:

procedure A^{7+j};OUTTEXT(" t_i "); $j \in [q-1]$

wobei t_i der j-te Teilstring von T ungleich " ist.

Die Menge der erzeugten Prozedurnamen ist
$AQ = \{A^{7+1},.....,A^{7+q-1}\}$. Sei $\mathcal{T} = \{t_1,.....,t_m\}$.
Wie im Beweis zu (5.2.4) werden die Funktionen τ und $\tilde{\tau}$ definiert:

$$\tau : [q] \longrightarrow \mathcal{T}$$
$$j \longmapsto t_k \text{ , mit } t_k = \text{j-ter Teilstring} \neq "$$

$$\tilde{\tau} : [m] \longrightarrow AQ \cup \{A^5\}$$
$$j \longmapsto \begin{cases} \overline{1}, \text{ falls } j = q \\ A^{7+i}, \text{ falls } t_j = \text{i-ter Teilstring} \neq " \\ A^5, \text{ falls } t_j = " \end{cases}$$

t_1 ist ungleich ", da π nicht mit " anfangen kann. Da das letzte Zeichen von T gleich ; ist, ist auch $t_m = t_q$ ungleich ". t_m wird in die Prozedur A^7 integriert.

Der Programmteil (zusätzliche Vereinbarungen) besteht aus den q-1 Prozedurvereinbarungen von A^{7+1} bis A^{7+q-1}.
Der Programmteil (zusätzliche Anweisungen) besteht aus der Folge von Prozeduraufrufen, die nötig sind, die q-1 zusätzlichen Prozedurvereinbarungen auszugeben. Die Prozedur A^7 wird entsprechend erweitert.

$\tilde{\pi}$ =

```
begin
procedure A;OUTTEXT("BEGIN PROCEDURE A;OUTTEXT("""");
procedure A^(7+1);OUTTEXT("τ(1)");
.....
procedure A^(7+q-1);OUTTEXT("τ(q-1)");
procedure A^2;OUTTEXT("PROCEDURE ");
procedure A^3;OUTTEXT(" ;OUTTEXT("""");
procedure A^4;OUTTEXT("""");");
procedure A^5;OUTTEXT("""""");
procedure A^6;OUTTEXT("A");
procedure A^7;OUTTEXT("τ(m);A;A;A^5;A^4;A^2;(A^6;)^(7+1) A^3;A^(7+1);A^4
   ;.....A^2;(A^6;)^(7+q-1) A^3;A^(7+q-1);A^4;A^2;(A^6;)^2 A^3;A^2;A^4
```

```
    ;A2;(A6;)3 A3;A3;A5;A4;A2;(A6;)4 A3;A5;A4;A4;A2;(A6;)5
   A3;A5;A5;A4;A2;(A6;)6 A3;A6;A4;A2;(A6;)7 A3;A7;A4;T(1);
   .......;T(m-1);A7 END");
```

t_1 ..... t_m

```
A;A;A5;A4;
A2;(A6;)7+1 A3;A7+1;A4;

.........................
A2;(A6;)7+q-1 A3;A7+q-1;A4;
A2;(A6;)2 A3;A2;A4;
A2;(A6;)3 A3;A3;A5;A4;
A2;(A6;)4 A3;A5;A4;A4;
A2;(A6;)5 A3;A5;A5;A4;
A2;(A6;)6 A3;A6;A4;
A2;(A6;)7 A3;A7;A4;
T(1); ..... ;T(m-1);A7
end
```

Teil B:

Die Konstruktion von $\tilde{\pi}$ in Teil A führt zu einem syntaktisch korrekten SIMULA-Programm. Wegen der sehr engen Entsprechung von dem SIMULA-Programm π_4 und dem PASCAL-Programm π_6 entspricht $\tilde{\pi}$ dem im Beweis von (5.2.4) erzeugten PASCAL-Programm weitgehend. Zum Nachweis, daß $\tilde{\pi}$ wirklich selbstreproduzierende Version von π ist, kann wegen der völligen Analogie auf den Beweis von (5.2.4),Fall 2, verwiesen werden.

%

Aus dem Beweis zu Satz (5.3.1) läßt sich direkt ein Algorithmus herleiten, der zu jedem gültigen SIMULA-Programm π eine selbstreproduzierende Version $\tilde{\pi}$ angibt.

(5.3.2) Algorithmus:

Eingabe: Das SIMULA-Programm π.

1.Schritt: Test, ob ⟨Klassenbezeichnung⟩π - falls überhaupt vorhanden - ein Bezeichner aus $\{A\}^+$ ist. Gegebenenfalls Umbenennung

2.Schritt: Zerlegung von T:= π; in Teilstrings $t_1,\ldots,t_m$ und Ermittlung der Anzahl q der Teilstrings t_j ungleich ". Anschließend Formulierung der q-1 Prozeduren A^{7+1} bis A^{7+q-1} und Aufstellung der Wertetabellen von τ und $\bar{\tau}$.

3.Schritt: Einsetzen der erhaltenen Prozeduren, der Funktionswerte und des Programms π in das im Beweis angegebene Programmschema.

Aufwand: Der Aufwand des Algorithmus verhält sich linear zur Länge $l(\pi)$ des Programms π.

Wegen der Analogie zu (5.2.5) sei an dieser Stelle auf ein Beispiel für die Anwendung von Algorithmus (5.3.2) verzichtet. Zur Gewinnung implementierbarer selbstreproduzierender Versionen muß (5.3.2) analog zu (5.2.5) um zwei „praxisorientierte" Schritte erweitert werden.

Kapitel 5 hat insgesamt ergeben, daß sich die eingangs (s.o. 5.1.) gestellten Fragen (1), (2) und (3) bezüglich der Programmiersprachen SIMULA und PASCAL positiv beantworten lassen. Bei der Beantwortung werden keine simula- bzw. pascalspezifischen Sprachelemente bemüht. Diese Tatsache läßt den Schluß zu, daß sich die Fragen (1) bis (3) im Falle jeder höheren Programmiersprache, die über

- Textkonstanten und
- Prozedurkonzept

verfügt, positiv beantworten lassen.

Auch bezüglich der in 3.4. behandelten SIEMENS-Assembler-Sprache fallen die Antworten auf die drei Fragen positiv

aus. Die Antworten wurden bereits durch Beispiel (3.4.1) geliefert. Wenige Zeilen Assembler-Kode genügen, um aus einem beliebigen Assembler-Programmabschnitt einen selbstreproduzierenden Programmabschnitt zu machen. Die die Selbstreproduktion ausmachenden Zeilen sind im wesentlichen immer gleich.

6. Selbstreproduktion bei loop-Programmen

6.1. Einleitung

In den Kapiteln 3,4 und 5 haben wir Beispiele für selbstreproduzierende Programme in höheren Programmiersprachen kennengelernt. Allen Beispielen ist gemeinsam, daß sie algorithmisch nicht sehr aufwendig sind. Der jeweilige Kontrollfluß aller bisherigen Beispielprogramme ist recht einfach. Es wäre also interessant zu klären, wie einfach Programmiersprachen strukturiert sein können, um noch selbstreproduzierende Programme zu ermöglichen. Die folgenden Betrachtungen werden also in erster Linie den in Programmiersprachen üblichen Kontrollstrukturen gelten und sich nicht auf eine konkrete Programmiersprache beziehen. Die Loslösung von konkreten Programmiersprachen wird dadurch zum Ausdruck gebracht, daß wir unsere Überlegungen auf der Basis der fiktiven Programmiersprache PL(A) aus Kapitel 2 durchführen.

In Kapitel 2 wurde die Menge der durch PL(A)-Programme realisierbaren Funktionen mit $\mathfrak{P}$ bezeichnet. Schränkt man die in PL(A) zur Verfügung stehenden Grundanweisungen und Kontrollstrukturen etwa auf

$$\gamma_1 , \gamma_2 , \gamma_3 , \gamma_5 \longrightarrow \begin{array}{l} \varkappa_1 : P;Q \\ \varkappa_2 : \underline{\text{if}}\ p\ \underline{\text{then}}\ \underline{\text{goto}}\ L \end{array}$$

oder auf

$$\gamma_1 , \gamma_2 , \gamma_3 , \gamma_5 \longrightarrow \begin{array}{l} \varkappa_1 : P;Q \\ \varkappa_3 : \underline{\text{if}}\ p\ \underline{\text{then}}\ P\ \underline{\text{else}}\ Q\ \underline{\text{fi}} \\ \varkappa_4 : \underline{\text{while}}\ X=\epsilon\ \underline{\text{do}}\ P\ \underline{\text{od}} \end{array}$$

ein, so erhält man Programmiersprachen, die nur „goto-Programme" bzw. nur „while-Programme" ermöglichen. Die Theorie zeigt jedoch (vgl. dazu etwa [5]), daß die Menge der mit while-Programmen realisierbaren Funktionen gleich der Menge der mit goto-Programmen realisierbaren Funktionen gleich der Menge $\mathfrak{P}$ ist. (Unsere bisherigen Beispielpro-

gramme für selbstreproduzierende Programme in SIMULA und PASCAL benutzen Prozeduren. Wollte man diese Programme in PL(A)-Programme transformieren, so würden goto-Programme entstehen.) Erst die Einschränkung von PL(A) auf

$$\gamma_1, \gamma_2, \gamma_3, \gamma_4 \longrightarrow \varkappa_1 : P;Q$$
$$\varkappa_2 : \underline{\text{loop}}\ X\ \underline{\text{case}}\ a_1 \longrightarrow P_1,\ \ldots\ a_n \longrightarrow P_n,\ \underline{\text{end}}\ ,$$

also auf reine „loop-Programme", führt zu Programmen, mit denen sich nicht mehr alle Funktionen aus $\mathcal{P}$ realisieren lassen. Wir werden im folgenden untersuchen, unter welchen Voraussetzungen selbstreproduzierende loop-Programme möglich sind.

6.2. Definition der Programmiersprache LP(A)

(6.2.1) Definition: Sei $A = \{a_1, \ldots, a_n\}$ ein endliches Alphabet. Sei PL(A) die in 2.2. definierte, zu A gehörige Programmiersprache. LP(A) ist die Programmiersprache, die entsteht, indem man aus PL(A) alle Programme streicht, die Grundanweisungen vom Typ $\gamma_5 : X:=\S(X)$, $X \in VR$, oder eine der Kontrollstrukturen

$$\varkappa_2 : \underline{\text{if}}\ p\ \underline{\text{then}}\ \underline{\text{goto}}\ L\ ,$$
$$\varkappa_3 : \underline{\text{if}}\ p\ \underline{\text{then}}\ P\ \underline{\text{else}}\ Q\ \underline{\text{fi}}$$

oder $\varkappa_4 : \underline{\text{while}}\ X=\varepsilon\ \underline{\text{do}}\ P\ \underline{\text{od}}$

enthalten.

(6.2.2) Bezeichnung:

I. Neben der Hintereinanderausführung von Anweisungen stellt die loop-Schleife $\varkappa_5$ das einzige Konstruktionselement für Programme aus LP(A) dar. Die Programme aus LP(A) werden daher auch als loop-Pro-

gramme bezeichnet.

II. Die Menge aller durch Programme aus LP(A) realisierbaren Wortfunktionen
$f : (A^*)^r \longrightarrow (A^*)^s$, $r,s \geqslant 0$
wird mit $\mathfrak{L}$ bezeichnet. $\mathfrak{L}$ heißt auch Menge der primitiv rekursiven Funktionen. [5]

Aus (6.2.1) ergibt sich, daß LP(A) eine echte Teilmenge von PL(A) ist. Daß auch $\mathfrak{L}$ eine echte Teilmenge von $\mathfrak{P}$ ist, ergibt sich schon aus der Tatsache, daß loop-Programme immer halten und somit die von loop-Programmen realisierten Funktionen total sind. Es läßt sich aber auch zeigen, daß $\mathfrak{L}$ eine echte Teilmenge von $\mathfrak{R}$ (vgl. (2.4.6)) ist, indem man die Existenz einer total rekursiven Funktion, die nicht primitiv rekursiv ist, nachweist(vgl. [5] Seite 41).

6.3.Eine kontextfreie Grammatik für LP(A)

Der Vollständigkeit halber sei hier eine kontextfreie Grammatik G'(A) für LP(A) angegeben. G'(A) entsteht durch Einschränkung der Grammatik G(A) für PL(A)-Programme aus 2.3..

(6.3.1) Angabe der Grammatik $G'(A) = (V'_T, V'_N, s_o, P')$

Die Menge der Terminalzeichen ist

$$V'_T = A \cup VR \cup \{\underline{\text{input}} , \underline{\text{output}} , \underline{\text{loop}} , \underline{\text{case}} , \underline{\text{end}} , \longrightarrow , ; , , , \sqcup , \varepsilon , \overline{\varepsilon} , : , = \} ,$$

wobei VR die Menge der zulässigen Variablennamen ist.

Die Menge der nicht terminalen Zeichen ist

$$V'_N = \{\langle\text{program}\rangle , \langle\text{statement}\rangle , \langle\text{simple statement}\rangle , \langle\text{identifier}\rangle , \langle\text{identifier list}\rangle \}$$

Das Startsymbol s_o ist ⟨program⟩

Die Menge P' der Produktionen ist gleich der Menge P der Produktionen der Grammatik G(A) aus (2.3.1) ohne die Produktionen mit den Nummern
6 , 8 , 9 , 10 , 13 , 14 , 15 und 20.

6.4.Erweiterung der Sprache LP(A)

In Kapitel 2 wurde die Existenz selbstreproduzierender Programme in PL(A) theoretisch bewiesen. Ein analoger Beweis für die Existenz von selbstreproduzierenden Programmen in LP(A) scheitert daran, daß es in $\mathfrak{L}$ keine universelle Funktion gibt (siehe [5] Seite 47). Wir werden uns daher dem Problem der Existenz selbstreproduzierender Programme in LP(A) von der praktischen Seite aus nähern. Das bedeutet allerdings nicht, daß es prinzipiell unmöglich ist, die Existenz selbstreproduzierender LP(A)-Programme auf theoretischem Wege zu beweisen.

Um das praktische Schreiben von selbstreproduzierenden LP(A)-Programmen für uns zu ermöglichen, erweitern wir die Programmiersprache LP(A) um eine zusätzliche Grundanweisung. Diese Erweiterung soll aber nicht bedeuten, daß es ohne sie prinzipiell unmöglich ist, selbstreproduzierende LP(A)-Programme zu schreiben.

I. Sei A ein endliches Alphabet. In der zu A gehörigen Sprache LP(A) soll es die Möglichkeit geben, Variable mit jedem beliebigen Wert aus A^* und nicht nur mit ε zu initialisieren. Wir führen daher die Grundanweisung γ_6 ein:

$$\gamma_6 : X:='a_{i_1} \ldots\ldots a_{i_k}' , \quad k \geq 1$$

$$X \in VR,\ a_{i_j} \in A \text{ für } j \in [k] .$$

Es soll nicht ausgeschlossen werden, daß $' \in A$ sein darf. Damit ist aber auch $a_{i_j} =' $ für beliebiges $j \in [k]$ zulässig.

In höheren Programmiersprachen ist es üblich, daß Texttrennzeichen, wenn sie innerhalb von Textkonstanten vorkommen, doppelt geschrieben werden müssen. Dieser Umstand hatte uns in den Kapiteln 3 und 5 das Schreiben von selbstreproduzierenden Programmen in PASCAL und SIMULA sehr erschwert. Wir treffen daher eine andere Vereinbarung:
Auf eine Grundanweisung γ_6 muß zwingend ein Semikolon fol-

gen. Das Ende einer Textkonstantenwird demnach durch ';' angezeigt. Der Einfachheit halber verbieten wir das Auftreten des Strings ';' als Teil einer Textkonstanten. Um die Koppelung des Semikolons an Textkonstanten deutlich zu machen, beziehen wir das Semikolon in die Definition von γ_6 ein.

$$\gamma_6 : X := '\,a_{i_1} \ldots\ldots a_{i_k}\,'; \quad , \quad k \geq 1$$

$$X \in VR,\ a_{i_j} \in A \text{ für } j \in [k] .$$

II. Ist $X \in VR$, so ist folgende Anweisung vom Typ γ_4 möglich:

X := X

Auf eine solche Anweisung wird in der Regel ein Semikolon folgen.

..... X := X; .....

Da wir nicht ausschließen wollen, daß das Semikolon ein Element aus A ist, könnte der String

X := X;

auch als Anweisung vom Typ γ_3 interpretiert werden. Um Doppeldeutigkeiten[1] zu vermeiden, ersetzen wir γ_3 durch die Grundanweisung γ_3' :

$$\gamma_3' : \quad X := X \mid a \qquad \forall\ X \in VR,\ a \in A.$$

Die Bedeutung von γ_3' ist die gleiche wie die von γ_3.

(6.4.2) Definition: Sei A ein endliches Alphabet. $\overline{LP(A)}$ ist diejenige Programmiersprache, die durch Erweiterung um die Grundanweisung γ_6 und durch Ersetzung der Grundanweisung γ_3 durch γ_3' aus der Sprache LP(A) entsteht.

Aus der Grammatik G'(A) für die Sprache LP(A) läßt sich leicht eine kontextfreie Grammatik $\overline{G'(A)}$ für die Sprache $\overline{LP(A)}$ herleiten:

- Die Menge der terminalen Zeichen V_T' wird um ' und | erweitert.

1) entstehen durch konkrete Wahl von A und traten daher in

- Die Produktion

  ⟨simple statement⟩ $\longrightarrow$ X:='a_{i_1} ..... a_{i_k}';

  für alle $X \in VR$, $a_{i_j} \in A$

  wird der Menge P' hinzugefügt.

- Die Produktion

  ⟨simple statement⟩ $\longrightarrow$ X:=Xa , für alle $X \in VR$, $a \in A$

  wird ersetzt durch

  ⟨simple statement⟩ $\longrightarrow$ X:=X|a , für alle $X \in VR$, $a \in A$

(6.4.3) Definition: (loop-Hierarchie der $\overline{LP(A)}$-Programme)
Sei A ein endliches Alphabet.

(i) Es sei $\overline{L_0(A)}$ die Klasse der Programme

π = input $X_1,.....,X_r$;
AW_π ;
output $Y_1,....,Y_s$, $r,s \geq 0$

aus $\overline{LP(A)}$, deren Anweisungsteil AW_π durch beliebiges Hintereinanderschreiben von Anweisungen vom Typ γ_1 , γ_2 , γ_3' , γ_4 und γ_6 entsteht.

(ii) Die Klasse $\overline{L_{i+1}(A)}$ enthalte alle Programme π , deren Anweisungsteile AW_π durch Hintereinandersetzen von Anweisungsteilen von Programmen aus $\overline{L_i(A)}$ und Anweisungsteilen der Form

loop X case $a_1 \longrightarrow AW_{\pi_1}$
⋮
$a_n \longrightarrow AW_{\pi_n}$
end

$a_j \in A$ für jedes $j \in [n]$

wobei AW_{π_1} ..... AW_{π_n} Anweisungsteile von Programmen aus $\overline{L_i(A)}$ sind, entstehen ($i \geq 0$).

6.5. Selbstreproduzierende Programme in $\overline{LP(A)}$

Sei π_{rep} ein - falls es existiert - selbstreproduzierendes $\overline{LP(A)}$-Programm. Im Anweisungsteil von π_{rep} muß schrittweise der Text π_{rep} aufgebaut werden. Die einzelnen Buchstaben von π_{rep} müssen auf der rechten Seite von Wertzuweisungen auftreten. Auf den rechten Seiten von Wertzuweisungen stehen aber außer Variablennamen nur Buchstaben aus A. Es muß also gelten: $\pi_{rep} \in A^*$. Um dies zu gewährleisten, wird folgende Forderung an das Alphabet A gestellt:

Forderung (1): Für jedes Programm $\pi \in \overline{LP(A)}$ gilt:

$$\pi \in A^*$$

Jedes Alphabet, das Forderung (1) erfüllt, muß alle Zeichen enthalten, die zur Konstruktion von $\overline{LP(A)}$-Programmen zulässig sind. Das kleinste, Forderung (1) erfüllende Alphabet ist demnach

$$A_{min} := \{a,c,d,e,i,l,n,o,p,s,t,u,\varepsilon,\bar{\varepsilon},\sqcup,:,=,;,,,\longrightarrow,|,'\}$$

Die Definition von $\overline{LP(A)}$ schränkt die Wahl der Variablenmenge VR in keiner Weise ein. Aus Forderung (1) ergibt sich jedoch, daß auch für die Variablennamen eines selbstreproduzierenden Programms $\pi_{rep} \in \overline{LP(A)}$ nur Buchstaben aus A in Frage kommen.

Forderung (2): $VR \subseteq \{A \setminus B\}^* \setminus \{case, loop, end, input, output\} \setminus \{\varepsilon\}$.
Dabei ist $B := \{\sqcup, :, =, \longrightarrow, ,, ;, \bar{\varepsilon}, \varepsilon, ', |\}$ die Menge der Sonderzeichen.

In einem $\overline{LP(A)}$-Programm, das Forderung (2) erfüllt, muß streng zwischen dem Variablennamen x und dem Zeichen x unterschieden werden, wobei x ein Buchstabe aus A ist. Die Definition von $\overline{LP(A)}$ schließt Fehlinterpretationen jedoch aus.

(6.5.2) Satz: Sei A endliches Alphabet mit $A_{min} \subset A$. Dann existiert in $\overline{L_2(A)}$ ein selbstreproduzierendes Programm, falls $\overline{LP(A)}$ Forderung (2) erfüllt.

Beweis: Betrachte das Programm π^2_{rep} =

```
input;
a:='input;a:=';;
c:=':=''';;
d:='''';;
e:='lnlnoocppnoodpnnoocepsnooipuncou';;
i:='loop e case l ⟶ loop a case i ⟶ t:=t|i,
                                n ⟶ t:=t|n,
                                p ⟶ t:=t|p,
                                u ⟶ t:=t|u,
                                t ⟶ t:=t|t,
                                ; ⟶ t:=t|;,
                                a ⟶ t:=t|a,
                                : ⟶ t:=t|:,
                                = ⟶ t:=t|=,
                   end,
          n ⟶ loop d case ' ⟶ t:=t|',
                   end,
          o ⟶ t:=t|;,
          c ⟶ t:=t|c,
          p ⟶ loop c case : ⟶ t:=t|:,
                          = ⟶ t:=t|=,
                          ' ⟶ t:=t|',
                   end,
          d ⟶ t:=t|d,
          e ⟶ t:=t|e,
          s ⟶ loop e case l ⟶ t:=t|l,
                          n ⟶ t:=t|n,
                          o ⟶ t:=t|o,
                          c ⟶ t:=t|c,
                          p ⟶ t:=t|p,
                          d ⟶ t:=t|d,
                          e ⟶ t:=t|e,
                          s ⟶ t:=t|s,
                          i ⟶ t:=t|i,
                          u ⟶ t:=t|u,
                   end,
          i ⟶ t:=t|i,
```

```
u ⟶ loop i case a ⟶ t:=t|a,
                  c ⟶ t:=t|c,
                  d ⟶ t:=t|d,
                  e ⟶ t:=t|e,
                  i ⟶ t:=t|i,
                  l ⟶ t:=t|l,
                  n ⟶ t:=t|n,
                  o ⟶ t:=t|o,
                  p ⟶ t:=t|p,
                  s ⟶ t:=t|s,
                  t ⟶ t:=t|t,
                  u ⟶ t:=t|u,
                  : ⟶ t:=t|:,
                  = ⟶ t:=t|=,
                  ' ⟶ t:=t|',
                  ; ⟶ t:=t|;,
                  , ⟶ t:=t|,,
                  ⟶ ⟶ t:=t|⟶,
                  ␣ ⟶ t:=t|␣,
               end,
        end;
        output t';;
loop e case l ⟶ loop a case i ⟶ t:=t|i,
                           n ⟶ t:=t|n,
                           p ⟶ t:=t|p,
                           u ⟶ t:=t|u,
                           t ⟶ t:=t|t,
                           ; ⟶ t:=t|;,
                           a ⟶ t:=t|a,
                           : ⟶ t:=t|:,
                           = ⟶ t:=t|=,
                        end,
n ⟶ loop d case ' ⟶ t:=t|',
                end,
o ⟶ t:=t|;,
c ⟶ t:=t|c,
p ⟶ loop c case : ⟶ t:=t|:,
                  = ⟶ t:=t|=,
                  ' ⟶ t:=t|',
                end,
```

```
d ⟶ t:=t|d,
e ⟶ t:=t|e,
s ⟶ loop e case l ⟶ t:=t|l,
                n ⟶ t:=t|n,
                o ⟶ t:=t|o,
                c ⟶ t:=t|c,
                p ⟶ t:=t|p,
                d ⟶ t:=t|d,
                e ⟶ t:=t|e,
                s ⟶ t:=t|s,
                i ⟶ t:=t|i,
                u ⟶ t:=t|u,
              end,
i ⟶ t:=t|i,
u ⟶ loop i case a ⟶ t:=t|a,
                c ⟶ t:=t|c,
                d ⟶ t:=t|d,
                e ⟶ t:=t|e,
                i ⟶ t:=t|i,
                l ⟶ t:=t|l,
                n ⟶ t:=t|n,
                o ⟶ t:=t|o,
                p ⟶ t:=t|p,
                s ⟶ t:=t|s,
                t ⟶ t:=t|t,
                u ⟶ t:=t|u,
                : ⟶ t:=t|:,
                = ⟶ t:=t|=,
                ' ⟶ t:=t|',
                ; ⟶ t:=t|;,
                , ⟶ t:=t|,,
              ⟶ ⟶ t:=t|⟶,
                ␣ ⟶ t:=t|␣,
              end,
    end;
    output t
```

π^2_{rep} ist offensichtlich ein gültiges $\overline{LP(A)}$-Programm.

Zu zeigen: π^2_{rep} reproduziert sich selbst.

Da π^2_{rep} keine Eingabe besitzt, ist die Behauptung, daß sich π^2_{rep} selbstreproduziert, gleichwertig mit der Aussage, daß der Inhalt der Variablen t bei Ausführung der letzten Anweisung „output t" gleich π^2_{rep} ist.

I. Im folgenden bezeichne [x] den Inhalt der Variablen mit dem Namen x. $x \in \{a,c,d,e,i,t\} \subset A_{min}$.

II. Auf Grund der Definition der loop-Schleife ist klar, daß die 4 inneren loop-Schleifen jeweils den Inhalt ihrer Laufvariablen an [t] hängen und [t] somit verlängern. Es sei hier folgende Abkürzung vereinbart:

$$([t] := [t][y]) := \left\{ \begin{array}{l} \text{loop y case } \ldots \longrightarrow \ldots \\ \quad \vdots \\ \text{end} \end{array} \right.$$

$y \in \{a,c,d,e,i\}$.

Die Abkürzung [t] := [t] x hingegen bedeutet das Anhängen des Zeichens $x \in A$ an den Inhalt von [t] .

III. Mit Hilfe der unter II. getroffenen Abkürzungen läßt sich π^2_{rep} in der folgenden Form formulieren:

```
input;
a:='input;a:=';;
c:=':=''';;
d:='''';;
e:='lnlnoocppnoodpnnooepsnooipunoou';;
i:='loop e case l ⟶ [t]:=[t][a],
                  n ⟶ [t]:=[t][d],
                  o ⟶ [t]:=[t];,
                  c ⟶ [t]:=[t]c,
                  p ⟶ [t]:=[t][c],
                  d ⟶ [t]:=[t]d,
                  e ⟶ [t]:=[t]e,
                  s ⟶ [t]:=[t][e],
```

```
                i ⟶ [t]:=[t]i,
                u ⟶ [t]:=[t][i],
            end;output t';;
```

[i]

Die äußere loop-Schleife arbeitet nun die Laufvariable e ab. e enthält genau die stringweise Kodierung von π^2_{rep}. Die Kodierung ist durch Hinschreiben der Alternativenliste gegeben. Die äußere loop-Schleife selbst dekodiert [e] und erzeugt sukzessive π^2_{rep}. In π^2_{rep} kommen nur Zeichen aus A_{min} vor. Daher ist $\pi^2_{rep} \in \overline{LP(A)}$ für jedes $A \supset A_{min}$, falls $\overline{LP(A)}$ Forderung (2) erfüllt.
Da die Schachtelungstiefe der loop-Schleifen in π^2_{rep} 2 ist, folgt der Satz.

%

Aus dem Programm π^2_{rep} aus dem obigen Beweis läßt sich ein Programm π^1_{rep} gewinnen, das mit der loop-Schachtelungstiefe 1 auskommt.

Wir konstruieren π^1_{rep} aus π^2_{rep}, indem wir die äußere loop-Schleife eliminieren. Die äußere loop-Schleife arbeitet den Inhalt der Variablen e ab. Der Inhalt von e ist 31 Zeichen lang. Wir listen nun für jedes der 31 Zeichen den Anweisungsteil der Alternativenliste auf, den es kodiert. Da e seine Kodierung enthält und die Variable e bei Eliminierung der äußeren loop-Schleife überflüssig wird, reduziert sich die Anzahl dieser Anweisungsteile auf 25. Bei nunmehr 25 Zeichen und insgesamt 10 Alternativen ist klar, daß einige Alternativen mehrfach geschrieben werden müssen. Damit wird gleichzeitig deutlich, daß die äußere loop-Schleife von π^2_{rep} nur abkürzenden Charakter besitzt.

```
π¹rep = input;
        a:='input;a:=';;
        c:=':=''';;
        d:='''';;
        i:='⟨Alternative für 1⟩;
            ⟨Alternative für n⟩;
            ⟨Alternative für 1⟩;
```

```
        ⟨Alternative für n⟩;
        ⟨Alternative für o⟩;
        ⟨Alternative für o⟩;
        ⟨Alternative für c⟩;
        ⟨Alternative für p⟩;
        ⟨Alternative für p⟩;
        ⟨Alternative für n⟩;
        ⟨Alternative für o⟩;
        ⟨Alternative für o⟩;
        ⟨Alternative für d⟩;
        ⟨Alternative für p⟩;
        ⟨Alternative für n⟩;
        ⟨Alternative für n⟩;
            ⟨Alternative für o⟩;
            ⟨Alternative für o⟩;
            ⟨Alternative für i⟩;
            ⟨Alternative für p⟩;
            ⟨Alternative für u⟩;
            ⟨Alternative für n⟩;
            ⟨Alternative für o⟩;
            ⟨Alternative für o⟩;
            ⟨Alternative für u⟩;
            output t';;
    ⟨Alternative für l⟩;
    ⟨Alternative für n⟩;
    ⟨Alternative für l⟩;
    ⟨Alternative für n⟩;
    ⟨Alternative für o⟩;
    ⟨Alternative für o⟩;
    ⟨Alternative für c⟩;
    ⟨Alternative für p⟩;
    ⟨Alternative für p⟩;
    ⟨Alternative für n⟩;
    ⟨Alternative für o⟩;
    ⟨Alternative für o⟩;
    ⟨Alternative für d⟩;
    ⟨Alternative für p⟩;
    ⟨Alternative für n⟩;
    ⟨Alternative für n⟩;
```

```
⟨Alternative für o⟩;
⟨Alternative für o⟩;
⟨Alternative für i⟩;
⟨Alternative für p⟩;
⟨Alternative für u⟩;
⟨Alternative für n⟩;
⟨Alternative für o⟩;
⟨Alternative für o⟩;
⟨Alternative für u⟩;
output t
```

Dabei sind die Teile in spitzen Klammern zu ersetzen:

⟨Alternative für 1⟩ durch

```
loop a case i ⟶ t:=t|i,
            n ⟶ t:=t|n,
            p ⟶ t:=t|p,
            u ⟶ t:=t|u,
            t ⟶ t:=t|t,
            ; ⟶ t:=t|;,
            a ⟶ t:=t|a,
            : ⟶ t:=t|:,
            = ⟶ t:=t|=,
      end
```

⟨Alternative für n⟩ durch

```
loop d case ' ⟶ t:=t|',
      end
```

⟨Alternative für o⟩ durch

```
t:=t|;
```

⟨Alternative für c⟩ durch

```
t:=t|c
```

⟨Alternative für p⟩ durch

```
loop c case : ⟶ t:=t|:,
            = ⟶ t:=t|=,
            ' ⟶ t:=t|',
      end
```

⟨Alternative für d⟩ durch

```
t:=t|d
```

⟨Alternative für i⟩ durch

```
t:=t|i
```

⟨Alternative für u⟩ durch

```
loop i case a ⟶ t:=t|a,
            c ⟶ t:=t|c,
            d ⟶ t:=t|d,
            e ⟶ t:=t|e,
            i ⟶ t:=t|i,
            l ⟶ t:=t|l,
            n ⟶ t:=t|n,
            o ⟶ t:=t|o,
            p ⟶ t:=t|p,
            s ⟶ t:=t|s,
            t ⟶ t:=t|t,
            u ⟶ t:=t|u,
            : ⟶ t:=t|:,
            = ⟶ t:=t|=,
            ' ⟶ t:=t|',
            ; ⟶ t:=t|;,
            , ⟶ t:=t|,,
          ⟶ ⟶ t:=t|⟶,
            ␣ ⟶ t:=t|␣,
       end
```

Offensichtlich gilt: π^1_{rep} reproduziert sich selbst. Da $\pi^1_{rep} \in A^*_{min}$, können wir den folgenden Satz formulieren:

(6.5.3) Satz: Sei A endliches Alphabet mit $A_{min} \subset A$. Dann existiert in $\overline{L_1(A)}$ ein selbstreproduzierendes Programm, falls $\overline{LP(A)}$ Forderung (2) erfüllt.

Nach der recht einfachen Konstruktion von π^1_{rep} aus π^2_{rep} könnte man versucht sein, aus π^1_{rep} ein selbstreproduzierendes Programm π^0_{rep} gewinnen zu wollen, das ganz ohne loop-Schleifen auskommt. Zu diesem Zweck müßten in π^1_{rep} die noch verbleibenden loop-Schleifen

⟨Alternative für l⟩
⟨Alternative für n⟩
⟨Alternative für p⟩
⟨Alternative für u⟩

eliminiert werden. Die erste dieser loop-Schleifen ließe sich ohne weiteres in eine Folge von Grundanweisungen zerlegen, da sie,wie schon die äußere loop-Schleife von π^2_{rep}, nur abkürzenden Charakter hat. Schwierigkeiten ergeben sich bei ⟨Alternative für n⟩ . ⟨Alternative für n⟩ ließe sich ohne loop wie folgt schreiben:

t:=t|'

Da auf ⟨Alternative für n⟩ ein Semikolon folgt, würde die Textkonstante von i den Teilstring t:=t|'; und damit die für Textkonstanten verbotene Kombination '; enthalten (vgl. 6.4.). ⟨Alternative für n⟩ läßt sich also nicht durch einen sequentiellen Programmteil ersetzen. Dieser Umstand liegt jedoch nur an der in 6.4. vorgenommenen Definition der Textkonstanten in $\overline{LP(A)}$-Programmen. Es ist durchaus denkbar, daß er sich bei einer anderen Definition der Textkonstanten vermeiden ließe. Ähnliches gilt für ⟨Alternative für p⟩ .

Anders liegen allerdings die Verhältnisse bei ⟨Alternative für u⟩ . Diese loop-Schleife dient dazu, den Inhalt der Variablen i an den Inhalt der Variablen t zu hängen. ⟨Alternative für u⟩ ist nun aber selbst textueller Bestandteil des Inhalts von i.

i:='.....⟨Alternative für u⟩.....'

⟨Alternative für u⟩läßt sich nicht durch eine Sequenz von Grundanweisungen ersetzen. Es läßt sich aus π^1_{rep} kein selbstreproduzierendes Programm gewinnen, das ohne loop-Schleifen auskommt. Es gilt sogar:

(6.5.4) Satz: Für jedes endliche Alphabet A gilt:
Es existiert kein selbstreproduzierendes Programm in $\overline{LP_0(A)}$

Beweis: Sei $\overline{LP(A)}$ über dem endlichen Alphabet A vorgelegt. Sei die notwendige Forderung (2) erfüllt.

Annahme: Es existiert selbstreproduzierendes Programm $\pi^0 \in \overline{LP_0(A)}$. Dann hat π^0 den Aufbau:

π^0 = input; $AW_\pi o$; output t

In $AW_\pi o$ muß der Text π^0 aufgebaut werden. Da keine loop-Schleifen zur Verfügung stehen, sind nur zwei Möglichkeiten zur Erzeugung des Textes π^0 in $AW_\pi o$ gegeben.

1.Fall: Der Text π^0 wird en bloc mit Hilfe einer Grundanweisung vom Typ γ_6 erzeugt. Dann gilt für π^0 die Textgleichung
π^0 = input;t:='π^0';output
Diese Gleichung kann aber von keinem endlichen Text π^0 erfüllt werden. Widerspruch!

2.Fall: Der Text π^0 wird in $AW_\pi o$ zeichenweise mit Anweisungen vom Typ γ'_3 aufgebaut. Da π^0 ungleich dem leeren Wort sein muß, kommt in $AW_\pi o$ mindestens einmal die Anweisung t:=t|x; mit $x \in A$ vor. Diese Anweisung umfaßt 7 Zeichen. Als String interpretiert ist sie textueller Bestandteil von π^0. Die Anweisung kann nur höchstens eines seiner 7 Zeichen an die Ausgabevariable t hängen. Daraus folgt, daß mindestens 6 Zeichen unbearbeitet bleiben. Für diese 6 Zeichen sind dann weitere 6 Befehle vom Typ γ'_3 notwendig. Diese 6 Anweisungen hinterlassen ihrerseits 36 unbearbeitete Zeichen, u.s.w.

Um also einen Text der Länge $k > 0$ mit Anweisungen vom Typ γ'_3 zu erzeugen, braucht man mindestens ein Programm der Länge k·7.

Damit gibt es kein endliches Programm der Länge k, das nur mit Anweisungen vom Typ γ'_3 einen endlichen Text der Länge k aus dem leeren Wort aufbauen kann, insbesondere nicht seinen eigenen Text. π^0 ist also nicht endlich und damit kein Programm. Widerspruch!

%

(6.5.5) Bemerkung: In Kapitel 2 wurde die Existenz selbstreproduzierender PL(A)-Programme nachgewiesen (Satz (2.8.7)). Dieser Nachweis beruhte neben dem

Rekursionstheorem (Satz (2.3.4)) auf der Existenz einer universellen Funktion für die Funktionenklasse $\mathcal{P}_1^1$.
Die Klasse aller Wortfunktionen

$$\varphi: (A^*)^r \longrightarrow (A^*)^s \quad , \quad r,s \geq 0 \quad ,$$

die sich mittels $\overline{LP(A)}$-Programmen berechnen lassen, ist eine Funktionenklasse, die nur aus totalen Funktionen besteht; $\overline{LP(A)}$-Programme halten immer an. In [5] Seite 47 wird gezeigt, daß es für $\overline{LP(A)}$-berechenbare Funktionen keine universelle $\overline{LP(A)}$-berechenbare Funktion geben kann. Trotzdem gibt es in $\overline{LP(A_{min})}$ selbstreproduzierende Programme. Universalität kann also keine notwendige Voraussetzung für Selbstreproduktion sein. Es muß also auch andere (direktere!) Wege als der in Kapitel 2 beschrittene Weg geben, um die Existenz selbstreproduzierender Programme theoretisch nachzuweisen.

6.6. Selbstreproduktionssatz für $\overline{LP(A)}$-Programme

In Kapitel 5 zeigten die Sätze (5.2.4) und (5.3.1) die Existenz selbstreproduzierender Versionen beliebiger SIMULA- bzw. PASCAL-Programme. Ein analoger Satz läßt sich auch für $\overline{LP(A)}$-Programme beweisen. Ersetzt man in Abschnitt 5.1. die Sprechweise „Eingabedatei" und „Ausgabedatei" wieder durch „Eingabevariable" bzw. „Ausgabevariable", so ist auch die selbstreproduzierende Version für $\overline{LP(A)}$-Programme erklärt.

Sei A ein beliebiges endliches Alphabet, sei $\pi \in \overline{LP(A)}$. Möglicherweise existiert in $\overline{LP(A)}$ keine selbstreproduzierende Version $\tilde{\pi}$ zu π (etwa weil die in $\overline{LP(A)}$ verwendeten Variablennamen nicht aus A^* sind), sondern erst in der Sprache $\overline{LP(B)}$ mit einem entsprechend „großen" Alphabet $B \supset A$. Nach Definition (5.1.4) wäre ein solches $\tilde{\pi} \in \overline{LP(B)}$ keine selbstreproduzierende Version von π, da es mit einem anderen Datenbereich arbeitet. Definition (5.1.4) ist aber erfüllt, wenn man π ebenfalls als Programm aus $\overline{LP(B)}$ auffaßt, was ja wegen $A \subset B$ durchaus zu vertreten ist: Die von $\pi \in \overline{LP(A)}$ rea-

lisierte Funktion ist gleich der Restriktion der von π als $\overline{LP(B)}$-Programm realisierten Funktion auf $(A^*)^r$, $r \geq 0$ (vgl. Definition (5.1.1)).

Dieser Sichtweise entspricht Satz (6.6.1).

(6.6.1) Selbstreproduktionssatz für $\overline{LP(A)}$-Programme

Seien A ein endliches Alphabet, $\pi \in \overline{LP(A)}$. Dann gibt es ein endliches Alphabet B, $A \subset B$, so daß gilt:

(a) Es existiert eine selbstreproduzierende Version $\tilde{\pi} \in \overline{LP(B)}$ von π (wobei π als Element aus $\overline{LP(B)}$ aufzufassen ist).

(b) $$\tilde{\pi} \in \begin{cases} \overline{LP_2(B)} \text{ , falls } \pi \in \overline{LP_j(A)} \text{ , } j=0,1 \\ \overline{LP_i(B)} \text{ , falls } \pi \in \overline{LP_i(A)} \text{ , } i \geq 2 \end{cases}$$

Beweis: Seien A beliebiges endliches Alphabet, $\pi \in \overline{LP(A)}$.
O.B.d.A. seien alle Variablennamen aus π nicht aus $\{c,d,e,i,t\}$.

Konstruktion der selbstreproduzierenden Version $\tilde{\pi}$

π hat den folgenden Aufbau

$\pi =$ input $y_1,\ldots,y_r$;
AW_π;
output $z_1,\ldots,z_w$, $r,w \geq 0$

Sei A_π die Menge aller Zeichen, aus denen das Programm π zusammengesetzt ist. Der String $S \in A_\pi^*$ sei wie folgt definiert:

S:=input $y_1,\ldots,y_r$;$\underbrace{AW_\pi}$;
(hiermit ist natürlich der Text des Anweisungsteils von π gemeint)

Es gilt: $\pi = S \circ$ output $z_1,\ldots,z_w$.

S wird in eine Folge von $n \geq 1$ Teilstrings s_i zerlegt mit:

(i) $s_i =$ '; oder ' ; ist nicht in s_i enthalten

(ii) s_i enthält ´; nicht $\Rightarrow$ s_{i+1}=´; , i<n

(iii) $s_1 \circ s_2 \circ \ldots\ldots \circ s_n = S$

(Zur Erinnerung: ´; dient als Endemarkierung von Textkonstanten und darf selbst nicht Teilstring einer Textkonstanten sein.)

Weitere Vorbereitungen:

- $\mathcal{S} := \{s_1, \ldots\ldots, s_n\}$ Menge der Teilstrings.
- q sei die Anzahl der Teilstrings von S, die ungleich ´; sind. Es gilt $1 \leq q \leq n$.
- $x_1, \ldots\ldots, x_{2q}$ seien Zeichen, die weder in $\{c,d,e,i,t\}$ noch als Variablennamen in π vorkommen.
- $B := A \cup A_\pi \cup A_{min} \cup \{x_1, \ldots\ldots, x_{2q}\}$
  B ist ein endliches Alphabet und läßt sich als solches in der Form $B := \{b_1, \ldots\ldots, b_\alpha\}$ schreiben, wobei α die Kardinalität von B ist.
- Definition der Funktionen σ und $\tilde{\sigma}$:

$$\sigma : [q] \longrightarrow \mathcal{S}$$

$$i \longmapsto s_j, \text{ wobei } s_j \text{ der i-te Teilstring ungleich ´; ist.}$$

$$\tilde{\sigma} : [n] \longrightarrow B^*$$

$$i \longmapsto \begin{cases} no, & \text{falls } s_i = ´; \\ x_j, & \text{falls } s_i \text{ der j-te Teilstring ungleich ´; ist.} \end{cases}$$

- Sei v eine zulässige Variable, dann sei die Anweisung

```
loop v case b_1 ⟶ t:=t|b_1,
                ..............
                b_α ⟶ t:=t|b_α,
           end
```

durch **loop** v abgekürzt.

Nach diesen Vorbereitungen läßt sich $\tilde{\pi} \in \overline{LP(B)}$ angeben:

$\tilde{\pi}$ =

$\boxed{= \underline{\text{input}}\ y_1,\ldots,y_n;\ AW_\pi;}$

```
σ1 ..... σn
x1:='σ(1)';;
x2:='σ(2)';;
.............
xq:='σ(q)';;
c:=':=';;
d:='';;
e:='σ̃(1) ..... σ̃(n)
    x1px(q+1)noo
    ..........
    xqpx(q+q)noo
    cppnoodpnnooepsnooipunoou';;
i:='loop e case x1 ⟶ t:=t|x1,
                x2 ⟶ t:=t|x2,
                ..............
                xq ⟶ t:=t|xq,
                c  ⟶ t:=t|c,
                d  ⟶ t:=t|d,
                e  ⟶ t:=t|e,
                i  ⟶ t:=t|i,
              x(q+1) ⟶ loop x1,
              x(q+2) ⟶ loop x2,
              ................
              x(q+q) ⟶ loop xq,
                n  ⟶ loop d case ' ⟶ t:=t|',
                                end,
                o  ⟶ t:=t|;,
                p  ⟶ loop c case : ⟶ t:=t|:,
                                = ⟶ t:=t|=,
                                ' ⟶ t:=t|',
                                end,
                s  ⟶ loop e,
                u  ⟶ loop i,
          end; output  z1,...,zw,t ';;
loop e case x1  ⟶ t:=t|x1,
            x2  ⟶ t:=t|x2,
            ..............
```

```
  x_q    ⟶ t:=t|x_q,
  c      ⟶ t:=t|c,
  d      ⟶ t:=t|d,
  e      ⟶ t:=t|e,
  i      ⟶ t:=t|i,
x_q+1    ⟶ loop x_1,
x_q+2    ⟶ loop x_2,
.....................
x_q+q    ⟶ loop x_q,
  n      ⟶ loop d case '⟶ t:=t|',
                      end,
  o      ⟶ t:=t|;,
  p      ⟶ loop c case : ⟶ t:=t|:,
                         = ⟶ t:=t|=,
                         ' ⟶ t:=t|',
                      end,
  s      ⟶ loop e,
  u      ⟶ loop i,
end;
output z_1,...,z_w,t
```

Behauptung: $\widetilde{\pi}$ ist selbstreproduzierende Version von π .

Programm $\widetilde{\pi}$ beginnt mit der Eingabe der Eingabevariablen von π und der Abarbeitung des vollständigen Anweisungsteils von π . Am Ende von $\widetilde{\pi}$ werden die Ausgabevariablen von π ausgegeben. Diese Ausgabevariablen $z_1,\ldots,z_w$ werden nicht durch Anweisungen außerhalb des Anweisungsteils von π verändert. $\widetilde{\pi}$ gibt zusätzlich die Variable t aus. Zum Nachweis, daß $\widetilde{\pi}$ selbstreproduzierende Version von π ist, genügt es also zu zeigen, daß am Ende der Programmausführung der Inhalt von t gleich $\widetilde{\pi}$ ist.

Die Variablen $x_1,\ldots,x_q$,c,d,e und i enthalten - bis auf einige einzelne Zeichen - den Programmtext $\widetilde{\pi}$ in zerlegter Form. Die Aufgabe der loop-Schleife von $\widetilde{\pi}$ ist es, die in den Variablen gespeicherten Teilstrings zum Text $\widetilde{\pi}$ zusammenzufügen. Die äußere loop-Schleife

loop e case ..... end;

wird durch die Variable e gesteuert. e enthält den Text $\tilde{\tilde{\pi}}$ in kodierter Form. Die Kodierung ist direkt aus der Alternativenliste ersichtlich. Für jedes Zeichen der Textkonstanten e wird genau ein Teilstring des Programms $\tilde{\tilde{\pi}}$ an den Inhalt der Variablen t gehängt. Die Textkonstante e läßt sich grob in drei Teile gliedern:

Teil I : $\tilde{\sigma}(1)$ ..... $\tilde{\sigma}(n)$

Teil II : $x_1 p x_{q+1} noo$
.....
$x_q p x_{q+q} noo$

Teil III: cppnoodpnnooepsnooipunoou

Teil I bewirkt, daß

input $y_1,\ldots,y_r$; AW_π;

an den Inhalt der zunächst leeren Variablen t gehängt wird.

Teil II verlängert den Inhalt von t um die Programmzeilen

x_1:='$\sigma(1)$';; bis
x_q:='$\sigma(q)$';;

Teil III bewirkt, daß der restliche Programmtext von $\tilde{\tilde{\pi}}$ an den Inhalt von t gehängt wird. Dies folgt aus der völligen Analogie zu π^2_{rep} aus dem Beweis von Satz (6.5.2).

Mit Teil III ist die Variable e vollständig abgearbeitet, und die Ausführung der äußeren loop-Schleife bricht ab. Damit stoppt auch das Gesamtprogramm,und $\tilde{\tilde{\pi}}$ wird als Inhalt von t ausgegeben. $\tilde{\tilde{\pi}}$ erfüllt also Definition (5.1.4)(ii) und ist somit selbstreproduzierende Version von π .

Die „reproduzierende" loop-Schleife in $\tilde{\tilde{\pi}}$ hat die loop-Schachtelungstiefe 2 und steht neben dem Anweisungsteil von π . Die loop-Schachtelungstiefe von $\tilde{\tilde{\pi}}$ ist daher mindestens 2, aber beschränkt durch die loop-Schachtelungstiefe von π . Damit ist auch die zweite Aussage des Satzes erfüllt.

%

(6.6.2) Bemerkung:

I. Aus dem Beweis von Satz (6.6.1) läßt sich direkt ein Algorithmus zur Ermittlung einer selbstreproduzierenden Version zu einem gegebenen $\overline{LP(A)}$-Programm gewinnen. Dieser Algorithmus ist jedoch stark verbesserungsbedürftig.

Beispiele:
- Die Wahl des Alphabets B läßt sich verfeinern. Man kommt mit „kleinerem" Alphabet B aus, als im Beweis angegeben.
- Die loop-Schleifen vom Typ

  loop v

  enthalten viele unnütze Alternativen, die im Beweis zugunsten einer einheitlichen Schreibweise in Kauf genommen werden.

II. Die im Beweis von (6.6.1) vorgenommene Konstruktion orientiert sich eng an dem Programm π^2_{rep} aus 6.5. Eine Konstruktion mit Hilfe des $\overline{LP_1(A_{min})}$-Programms π^1_{rep} würde zu selbstreproduzierenden Versionen $\tilde{\pi}$ führen mit

$$\tilde{\pi} \in \begin{cases} \overline{LP_1(B)} & , \text{ falls } \pi \in \overline{LP_0(A)} \\ \overline{LP_i(B)} & , \text{ falls } \pi \in \overline{LP_i(A)},\ i \geq 1. \end{cases}$$

Diese Konstruktion wäre aber noch schwieriger zu überschauen als die Konstruktion mittels π^2_{rep}.

7.Leben bei Programmen?

7.1.Einleitung

In den vorangegangenen Kapiteln wurde die Existenz selbstreproduzierender Programme sowohl in Assembler-Sprachen als auch in höheren Programmiersprachen nachgewiesen. Speziell Kapitel 5 zeigt, daß es nicht nur unendlich viele selbstreproduzierende Programme gibt, sondern daß sich jede Programmieraufgabe effektiv mit einem selbstreproduzierenden Programm bewältigen läßt; Grenzen sind nur durch die technisch physikalischen Gegebenheiten einer konkreten Rechenanlage gesetzt.

Elektronische Datenverarbeitungsanlagen arbeiten nicht hundertprozentig fehlerfrei. Schalt- und Übertragungsfehler sind jederzeit möglich, wenn auch sehr unwahrscheinlich.

Beispiel: Aus Gründen der Effizienzsteigerung und der besseren Nutzbarmachung einzelner hardware-Komponenten werden immer häufiger einzelne Rechner zu Rechnernetzen [21] zusammengeschaltet. Da die Entfernung der Einzelrechner zueinander oft mehrere Hundert Kilometer beträgt, stellt die Übertragung der Daten zwischen den Rechnern eines Netzes ein besonderes Problem dar. Je nach Art des gewählten physikalischen Übertragungsweges liegt die Fehlerrate bei der Datenübermittlung zwischen 10^{-4} und 10^{-7} bit/sec (10^{-5} bit/sec beim öffentlichen Telefonnetz) [21] [12] . Durch hard- und softwaremäßige Maßnahmen (z.B. Verwendung fehlerkorrigierender Kodes [11] , Kommunikationsprotokolle [21]), die in ihrer Gesamtheit als Fehlerkontrolle bezeichnet werden, kann die Rate der unerkannten Fehler niedriger gehalten werden. So liegt z.B. Die Bitfehlerrate für Bitübertragung beim ARPA-Netz [12] bei 10^{-12} bit/sec.

Es besteht die Möglichkeit, daß bei der Selbstreproduktion eines Programms π Fehler unterlaufen (z.B. bei der Übertragung der Kopie aus dem Arbeitsspeicher in den Hintergrundspeicher). Diese Fehler können dazu führen, daß effektiv ein anderes Programm $\pi' \neq \pi$ reproduziert wird. Falls π' ein syntaktisch korrektes Programm ist, kann π' durchaus wieder selbstreproduzierend sein und dabei eine andere Funktion realisieren als π . Dieser Sachverhalt erinnnert stark an Reproduktion und Mutation lebender Zellen in der Biologie. Reproduktion und Mutation gehören nach Ansicht der Biologie zu den Grundeigenschaften alles Lebendigen. Es drängt sich in diesem Zusammenhang die Frage auf, ob sich Programmen noch weitere lebenskennzeichnende Prozesse zuordnen lassen. Ist es vielleicht sogar möglich, in Anlehnung an die Biologie von lebendigen Programmen zu sprechen? Die Beantwortung dieser Fragestellungen stößt auf eine Vielzahl von Schwierigkeiten, von denen die beiden folgenden wohl zu den bedeutsameren gehören.

- Die moderne Biologie ist sich durchaus nicht einig, wenn es darum geht, die charakteristischen Merkmale des Lebens auf eindeutige Weise zu definieren (vgl. 7.2.).

- Biologisches Leben basiert auf einem äußerst vielschichtigen Zusammenspiel von biochemischen Reaktionen. Bestimmten Makromolekülen, namentlich Nukleinsäure- und Aminosäuremoleküle, fallen dabei Schlüsselpositionen zu [13] . Alle irdischen Lebensformen werden von dem Zusammenwirken von Nuklein- und Aminosäuren bestimmt. Die Frage, ob andere Lebensformen als die uns vertrauten denkbar sind, versucht die Biologie zu beantworten, indem sie das Problem untersucht, ob die Funktionen der Nuklein- bzw. Aminosäuren durch andere Makromolekülsorten ersetzt werden können [13] . Diesem Vorgehen läßt sich die Auffassung entnehmen, daß Leben immer chemophysikalischen Ursprungs ist. Leben bei Computerprogrammen wäre in diesem Sinne

unmöglich.

Die Suche nach „Leben" bei Programmen wird also sicherlich von philosophischen Problemen und Problemen der theoretischen Biologie begleitet sein. Es liegt daher auch nicht in der Absicht dieses und der beiden letzten Kapitel, „Leben" bei Programmen zu definieren. Die abschließenden Kapitel der vorliegenden Arbeit sind eher als ein erster Versuch zur Erschließung des dargelegten Problemkreises,verbunden mit einigen Denkmöglichkeiten, zu verstehen.

7.2.Biologisches Leben

Die moderne Biologie ist immer noch auf der Suche nach einer einheitlichen Definition des Lebens. Aus der Vielzahl rezenter und ausgestorbener Lebensformen lassen sich jedoch einige gemeinsame Eigenschaften alles Lebendigen extrahieren. So sind nach einer weitverbreiteten Auffassung

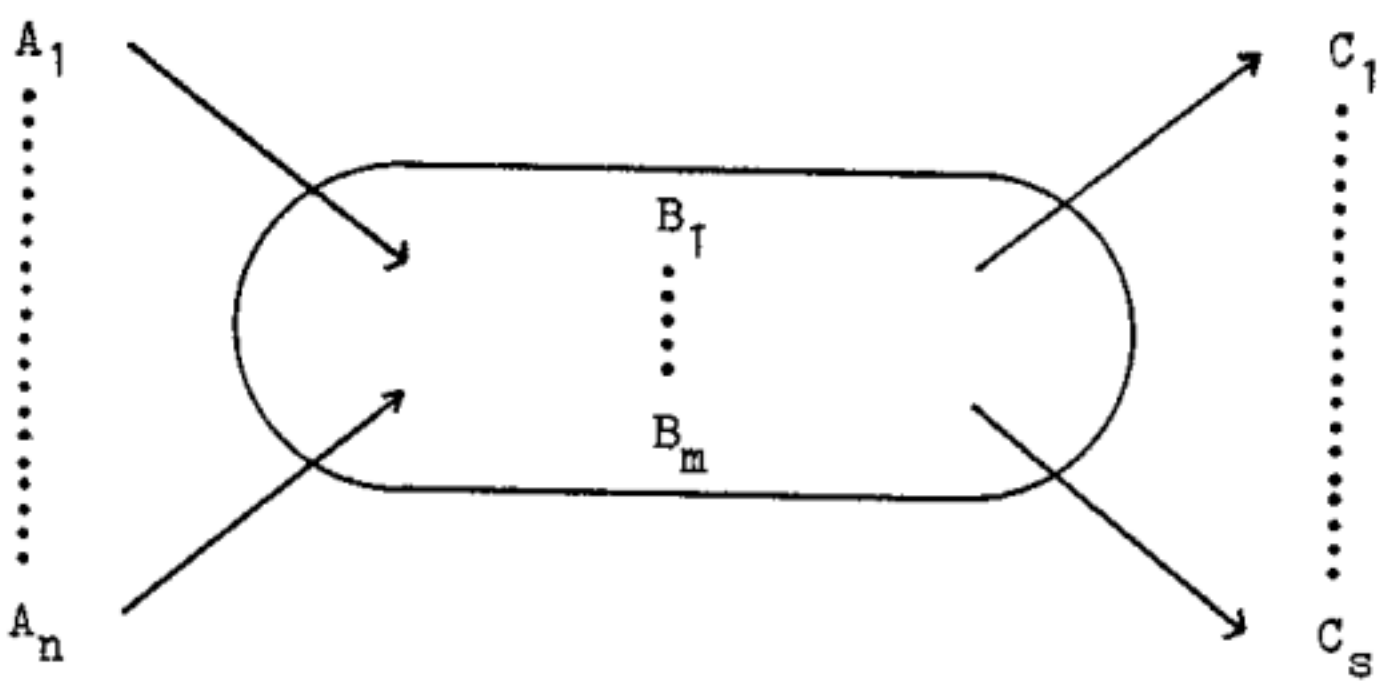
- Stoffwechsel (+ Regelung)
- Zellreproduktion und Mutation

die Schlüsselprozesse des Lebens. Diese Schlüsselprozesse dienen der Erhaltung der Individuen, der Vermehrung und dem Erbwandel (vgl. [13] Seite 24 ff). Es gibt Auffassungen von Leben, die auch Reizbarkeit und Bewegung als charakteristische Aspekte des Lebendigen anführen (vgl. [15] Seite 335).

Stoffwechsel

Die „Grundeinheit" des Lebens ist die Zelle. Alle Lebewesen sind aus Zellen aufgebaut. Sowohl Zellen als auch Lebewesen stellen begrenzte Stoffsysteme dar. Eine Zelle nimmt aus ihrer Umgebung ständig Stoffe auf, wandelt sie intern um und gibt sie in veränderter Form wieder

an ihre Umwelt ab. Da jede Zelle auf diese Weise ständig von Stoffen durchströmt wird, wird das System „Zelle" als Fließsystem (Abb. 7.2.A) bezeichnet [13] . Fließsysteme sind offene Systeme [2] . Die Stoffumwandlungen in Zellen laufen in geordneten Bahnen ab. Es stellt sich ein sogenanntes Fließgleichgewicht ein. (In [2] Seite 158 wird die Auffassung vertreten, daß alle charakteristischen Eigenschaften lebender Organismen direkte Konsequenzen des Fließgleichgewichts sind). Offene Systeme im Fließgleichgewicht streben unabhängig von den Anfangsbedingungen einem konstanten Zustand entgegen. Dieser Zustand heißt stationärer Zustand. Stofflich gleiche Fließsysteme streben, sofern sie sich in einer gleichen Umgebung befinden, dem gleichen stationären Zustand entgegen, auch wenn die Anfangsbedingungen unterschiedlich sein mögen. Man kann also von einer gewissen Selbstregulationsfähigkeit der Lebewesen bzgl. des Stoffwechsels sprechen.



Schema eines chemischen Fließsystems: Die Stoffe A_i treten in das System ein. Innerhalb des Systems werden mittels Binnenreaktionen die Stoffe B_j erzeugt. Die (Abfall-) Produkte C_k treten aus dem System aus.

Abb. 7.2.A

Der Stoffwechsel wird häufig, rein heuristisch, in den Baustoffwechsel und den Energiestoffwechsel differenziert. Während der Baustoffwechsel dem materiellen Aufbau bzw. dem Wachstum der Lebewesen dient, liefert der Energiestoffwechsel die zur Aufrechterhaltung der Lebensprozesse notwendige Energie.

Reproduktion und Mutation

Bei lebenden Organismen beruht die Vermehrung auf Zellteilung. Die Teilung einer Zelle erfolgt dabei so, daß die entstehenden Tochterzellen die gleiche Struktur und das gleiche Ablaufschema der Stoffwechselreaktionen erhalten wie die Elterzelle(n) (Singular bei ungeschlechtlicher, Plural bei geschlechtlicher Vermehrung). Aus dem oben über Fließsysteme Gesagten folgt, daß die Tochterzellen dem gleichen stationären Zustand zustreben wie die Elterzelle(n), vorausgesetzt, die Umweltbedingungen sind während und nach der Zellteilung identisch. Auf diese Weise „ererben" die Tochterzellen den Zustand der Elterzelle(n). Die Struktur einer Zelle und die in der Zelle ablaufenden Stoffwechselreaktionen werden durch Proteine bestimmt. Damit eine Tochterzelle den gleichen stationären Zustand wie die Elterzelle(n) bei konstanten Umweltverhältnissen erreichen kann, genügt es also, bei der Zellreproduktion dafür zu sorgen, daß

- in den Tochterzellen die gleichen Proteine gebildet werden wie in der (den) Elterzelle(n),
- sich diese Bildung in der zeitlich richtigen Reihenfolge vollzieht.

Die zur Synthese der Proteine notwendige Information enthält jede Zelle in Form speziell strukturierter Nukleinsäuremoleküle. Diese Moleküle werden als DNS, einzelne funktionelle Molekülabschnitte als Gene [27] bezeichnet. Damit die Tochterzellen in der Lage sind, die gleichen Proteine zu bilden wie die Elterzelle(n),

bekommt jede Tochterzelle bei der Reproduktion eine identische Kopie der DNS-Moleküle der Elterzelle, also identische Gene mit; bei geschlechtlicher Vermehrung handelt es sich um eine Kombination der DNS-Moleküle der Elterzellen (allele Gene). Die Elterzellen vererben also den Bauplan der in ihnen gebildeten Proteine. Unterlaufen bei der Replikation der DNS-Moleküle Fehler und werden die fehlerhaften Kopien an die Tochterzellen weitergegeben, so werden in den Tochterzellen andere Proteine erzeugt als in den Elterzellen. Es werden dann i.a. in den Tochterzellen andere Binnenreaktionen erfolgen. In den Tochterzellen wird sich trotz sonst gleicher Umweltbedingungen ein anderes Fließgleichgewicht als in den Elterzellen einstellen. Auch der stationäre Zustand der Tochterzellen wird ein anderer sein. Man bezeichnet bei lebenden Organismen solche sprunghaften Änderungen des Erbgutes (Genom) als Mutation. Mutationen erfolgen immer zufällig und ungerichtet. Mutationen werden nicht nur durch fehlerhafte Kopierprozesse hervorgerufen, sondern können auch bei bereits „fertigen" Zellen durch spontane Änderungen der DNS-Moleküle entstehen.

Diese beiden extrem kurzen und nicht annähernd vollständigen Überblicke über Stoffwechsel, Reproduktion und Mutation von Lebewesen sollen als Grundlage zu den folgenden Überlegungen dienen.

7.3.Selbstreproduzierende Programme und Leben

Im Gegensatz zu natürlichen Lebewesen sind Programme in erster Linie Informationen und als solche nicht stofflich. Damit Information verfügbar ist, muß sie in einer interpretierbaren Form zugänglich sein.
Beispiele:

- Lochstreifen
- Lochkarten
- Formulierung mit „Papier und Beistift"

:

Programme werden in der Regel erstellt, um sie von einer konkreten Rechenmaschine ausführen zu lassen. Im Rechner sind dann Programme digital dargestellt und für den Rechner interpretierbar, vorausgesetzt sie werden als syntaktisch und semantisch[1)] korrekt vom vorhandenen Rechner akzeptiert. Gehen wir einmal von der vertrauten Darstellung der Programme im Rechner aus, so müssen wir uns dennoch ständig im Klaren sein, daß Programme in ihrer Existenz an keine der möglichen Darstellungsformen gebunden sind. Wegen ihres mehr abstrakten als stofflichen Daseins benötigen Programme auch keinen Stoffwechsel, um ihre Existenz zu erhalten. Die Tatsache, daß Energie benötigt wird, um ein Programm π auf einer Rechenanlage auszuführen, kann natürlich nicht als (Energie-) Stoffwechsel betrachtet werden, da die Zufuhr von Energie

- nicht vom Programm π aktiv gesteuert wird,
- nicht der Erhaltung des Programms π , sondern der Interpretation von π dient.

Programme sind auf Rechenanlagen in irgendwelchen Speichermedien abgespeichert. Es gibt Speichertypen, gewisse Halbleiterspeicher (Charge-Coupled Devices), die in gewissen Zeitabständen eine Erneuerung der enthaltenen Information benötigen [11] . In solchen Speichern abgelegte Programme benötigen also regelmäßig Energie, um verfügbar zu bleiben. Auch eine solche Energiezufuhr kann man,aus den gleichen Gründen wie oben, nicht im entferntesten als Energiestoffwechsel ansehen.

Es hat also wenig Sinn, im Hinblick auf Programme, auch wenn man von der festen Darstellung auf einer konkreten Rechenanlage ausgeht, ein Analogon zum Stoffwechsel lebender Organismen zu suchen.

Anders liegen die Verhältnisse allerdings bzgl. Reproduktion und Mutation. Die vielen Beispiele selbstreproduzierender Programme in den vorangegangenen Kapiteln zeigen, daß es durchaus Programme gibt, die zur identischen Reproduktion fähig sind. Allen Beispielprogrammen in höheren Programmiersprachen war gemeinsam,

---

1)semantisch hier im Sinne von: keine Laufzeitfehler.

daß sie an irgendeiner Stelle ihren eigenen Bauplan in kodierter Form enthalten (vgl. etwa Inhalt der Komponente C[23] in π_3 aus Abschnitt 3.2.5., Textkonstante der Prozedur AB in π_4 aus Abschnitt 3.2.7.). Dieser Bauplan läßt einen Vergleich mit der DNS lebender Zellen zu. Der Zusammenbau der Kopien selbstreproduzierender Programme mittels des Bauplans ist vergleichbar (zugegeben sehr gewagt) mit der Proteinsynthese bei Zellen. Man sollte sich jedoch einschränkend vergegenwärtigen, daß die Selbstreproduktion von Programmen im strengen Sinne keine Autoreproduktion darstellt, wie dies bei Organismen der Fall ist. Dies liegt daran, daß die Reproduktion von Programmen von <u>außen</u> veranlaßt wird (Steuerung durch das Betriebssystem) und die Initiative nicht beim Programm selbst liegt.

Mögliche Schalt- und Übertragungsfehler in elektronischen Rechenanlagen können dazu führen, daß bei der Selbstreproduktion von Programmen Fehler entstehen (vgl. 7.1.). Mutationen sind also in diesem Sinne jederzeit möglich.

Insgesamt ergibt sich also, daß sich von den biologisches Leben ausmachenden Schlüsselprozessen bei Programmen nur Entsprechungen bzgl. Reproduktion und Mutation finden lassen. Systeme, die nur die Eigenschaften der Reproduktion und Mutation enthalten, sind daher im Sinne der Biologie nicht als lebendig zu bezeichnen. Insofern sind auch selbstreproduzierende Programme nicht lebendig. Selbstreproduzierende Programme lassen sich somit auch nicht mit lebendigen Organismen vergleichen. Die Biologie kennt jedoch Strukturen, die durchaus einen Vergleich mit selbstreproduzierenden Programmen zulassen.

<u>7.4.Selbstreproduzierende Programme und Viren</u>

Viren wurden lange Zeit als einfachste Organismen angesehen. Sie sind sehr viel einfacher gebaut als einzellige

Lebewesen. In Wirklichkeit stellen Viren jedoch keine vollständigen Organismen dar, sondern sind subzellulare Gebilde, die fast nur aus DNS bestehen. Bei manchen Viren sind die DNS-Moleküle noch mit einer Hülle aus Proteinen, Fettstoffen und anderen organischen Substanzen umgeben. Viren verfügen über keinen eigenen Stoffwechsel. Erst wenn Viren in eine lebende Zelle eindringen, zeigen sie Lebenserscheinungen in Form von Reproduktion und Mutation. Sie benötigen zu ihrer eigenen Vermehrung also den Stoffwechsel echter Organismen. Außerhalb lebender Organismen sind Viren tot, sie können sich dann sogar zu Kristallen anordnen, was von Lebewesen nicht bekannt ist. Von den Schlüsselprozessen des Lebens weisen Viren also nur Reproduktion und Mutation auf und das auch nur dann, wenn eine fremde Stoffwechselmaschinerie Baustoffe und Energie zur Verfügung stellt. Diese Zusammenhänge sind in ähnlicher Form auch bei selbstreproduzierenden Programmen festzustellen. Solange ein selbstreproduzierendes Programm sich nicht im Speicher einer Rechenanlage befindet, kommt ihm bis auf seinen Informationsgehalt keine Bedeutung zu. Erst im Rechner und dann auch erst , wenn das Programm wirklich läuft, ist ein selbstreproduzierendes Programm in der Lage zur Reproduktion und Mutation. Dem Programm steht dann Energie, die vom Rechner geliefert wird, zur Verfügung. Es bleibt jedoch bei aller Ähnlichkeit zu beachten, daß ein Virus aktiv seine Reproduktion einleitet, indem es in das Baustoff und Energie liefernde System „Zelle" eindringt. Das kann ein selbstreproduzierendes Programm nicht, auch wenn es sich im Speicherplatz und Energie liefernden System „Rechner" befindet. Es bleibt auf Aktivierung durch das Betriebssystem angewiesen.

8.Modelle für konkurrierendes Verhalten selbstreproduzierender Programme

8.1.Motivation

Wie die Biologie lehrt, sind lebende Organismen vielschichtigen Konkurrenzkämpfen unterworfen. Diese Konkurrenzkämpfe betreffen nicht nur einzelne Individuen, sondern ganze Arten (bzw. Populationen [24] Seite 337). Um erfolgreich zu sein, müssen diese Arten eine gewisse Variabilität ihres Erbgutes (Genom) aufweisen. Nur so können sie sich einerseits gegenüber der unbelebten Umwelt, die sich ständig verändert, und andererseits gegenüber der Konkurrenz anderer Arten behaupten. Die Eigenschaften der Arten und Individuen sind also ständig in Beziehung zur belebten und unbelebten Umwelt zu sehen (Ökologie, vgl. [27] Seite 199 ff).

Selbstreproduzierende Programme, die sich in einer Rechenanlage befinden, sind von der „Umwelt" Rechner (= hardware + Systemsoftware) umgeben. Das Speichermedium, das die Programme enthält, ist sogar ein Teil dieser Umwelt. Als „belebte" Umwelt lassen sich andere, ebenfalls im Rechner befindliche selbstreproduzierende Programme ansehen. Die Möglichkeit der Mutation (s.o. 7.3.) befähigt selbstreproduzierende Programme zur Evolution (s.u. 9.1.). Es ist also nicht auszuschließen, daß die Wechselwirkungen selbstreproduzierender Programme miteinander und mit dem umgebenden Rechnersystem zu anderen selbstreproduzierenden Programmen mit immer neuen Eigenschaften führen können. Die folgenden (spekulativen!) Modelle sollen versuchen, eine Vorstellung von derartigen auf Wechselwirkungen beruhenden Verhaltensweisen selbstreproduzierender Programme zu vermitteln.

8.2.Ein einfaches Grundmodell

Das nachfolgend beschriebene Grundmodell MOD1 geht von dem Hintergrund einer herkömmlichen Rechenanlage mit „von Neuman Architektur" aus [12] . Eine solche Rechenanlage zeichnet sich in moderner Sicht durch einen (Zentral-)Pro-

zessor und ein oder mehrere E/A-Kanäle (extrem: E/A-Prozessoren) aus. Es herrscht multiprogramming Betrieb: K Programme $\pi_1,\ldots,\pi_K$ können zeitlich verzahnt, jedoch nicht durchgehend parallel verarbeitet werden. Die Programme sind nur zeitlich alternierend aktiv. In diesem Zusammenhang sind auch Begriffe wie time slicing und time sharing zu erwähnen [1].

Selbstreproduzierende Programme werden in MOD1 durch 2 Größen charakterisiert.

- Die Laufzeit (sprich: Reproduktionszeit)
- Die räumliche Beziehung (bzgl. Speicher) zwischen Programm und dessen erzeugter Kopie.

8.2.1.Informelle Beschreibung von MOD1

(i) Programme: Programme werden mit ihrem Namen identifiziert. Hinter diesem Namen verschwindet die Programmstruktur vollkommen. Wichtiger sind hingegen andere Daten:

a) Die individuelle Anzahl der Zeittakte, die erforderlich sind, damit ein Programm sich reproduzieren kann.

b) Die Mindestentfernung, in der die Kopie im Speicher angelegt wird (vgl. (ii)).

Das Modell eines Programms ist somit ein Tripel, bestehend aus Programmname, Reproduktionszeit und Kopiedistanz.

(ii) Speicher: Der Speicher ist eindimensional, beidseitig unendlich und in Speicherzellen unterteilt. Je zwei benachbarte Speicherzellen sind direkt miteinander verbunden. Jede Speicherzelle ist in der Lage, genau ein Programm - unabhängig von dessen physikalischer Länge - aufzunehmen. Fast alle Speicherzellen sind leer.

(iii) Zeitverhalten: Anfangs wird der leere Speicher mit

einer festen Zahl NUM von selbstreproduzierenden Programmen $\pi_1,\ldots,\pi_{NUM}$ initialisiert. Jedes Programm π_j erhält zyklisch für einen Zeittakt die „Aktivität" zugeteilt. Diese zyklische „Aktivierung" ist möglich, da sich zu jedem Zeitpunkt nur endlich viele Programme im Speicher aufhalten. Wächst die Anzahl der Programme durch Reproduktionen an, so vergrößert sich der Zyklus entsprechend. Jedes Programm im Speicher ist nach einer individuellen Anzahl von Zeittakten, in denen es „aktiv" war (Reproduktionszeit), in der Lage, sich selbst zu reproduzieren.

(iv) Räumliches Verhalten: Ein Programm legt seine Kopie in einem individuellen Mindestabstand nach rechts oder links an. Ist die ausgewählte Speicherzelle besetzt, so werden alle weiteren folgenden Speicherzellen getestet. Die Kopie wird dann in die erste freie Zelle abgelegt. Diese existiert wegen (ii) immer. Es entstehen also hinsichtlich des Speicherplatzbedarfs keine Konflikte.

MOD1 vermeidet Kollisionen. Die Programme können sich nicht gegenseitig in Bedrängnis bringen. Es gibt keine ausgezeichneten Programme, die in der Lage sind, andere Programme zu zerstören, indem sie deren Speicher beanspruchen. In diesem Sinne sind alle Programme äquivalent. Die Vermeidung von Kollisionen wird durch die Unendlichkeit des Speichers unterstützt. Jedes Programm (Individuum) ist beständig und produziert während seines ewigen Daseins identische Kopien (Nachkommen). Die Menge der vorhandenen Programme (Population) steigt ständig an. Bildhaft gesprochen gibt es in MOD1 keinen „Kampf ums Dasein", sondern friedliche Koexistenz. In einem solchen Modell gibt es keine Evolution. Die Motoren der Evolution, Mutation und Selektion, sind ohne Bedeutung, da es keinen Selektionsdruck gibt (s.u.).

8.2.2.MOD1 als SIMULA-Programm

(i) Programme:

Umsetzung des Programme bestimmenden Tripels in die SIMULA-Struktur:

```
class PROGRAM;
      begin
      integer DELY,DISTANCE,IDENT;
      end;
```

DELY: Reproduktionszeit

DISTANCE: Mindestentfernung der Kopie

IDENT: Identifizierung des Programms

(ii) Speicher:

Jede Speicherzelle ist in der Lage, ein Objekt vom Typ PROGRAM aufzunehmen. Sie ist außerdem mit ihren beiden Nachbarzellen verbunden. Diesen Eigenschaften trägt die SIMULA-Struktur CELL Rechnung:

```
class CELL;
      begin
      ref (PROGRAM) CONTENS;
      ref (CELL) LEFT,RIGHT;
      integer TIMECOUNT;
      end;
```

(bzgl. TIMECOUNT s.u.)

Der Speicher selbst wird also als doppelt verkettete lineare Liste dargestellt ([28] Seite 233 ff). Anfangs wird ein völlig leerer Speicher der festen Länge N angelegt. Je nach Bedarf werden an seine Enden neue Speicherzellen angehängt, so daß der Speicher potentiell unendlich ist, aber zu jedem Zeitpunkt eine feste Länge aufweist. Die beiden Zeiger

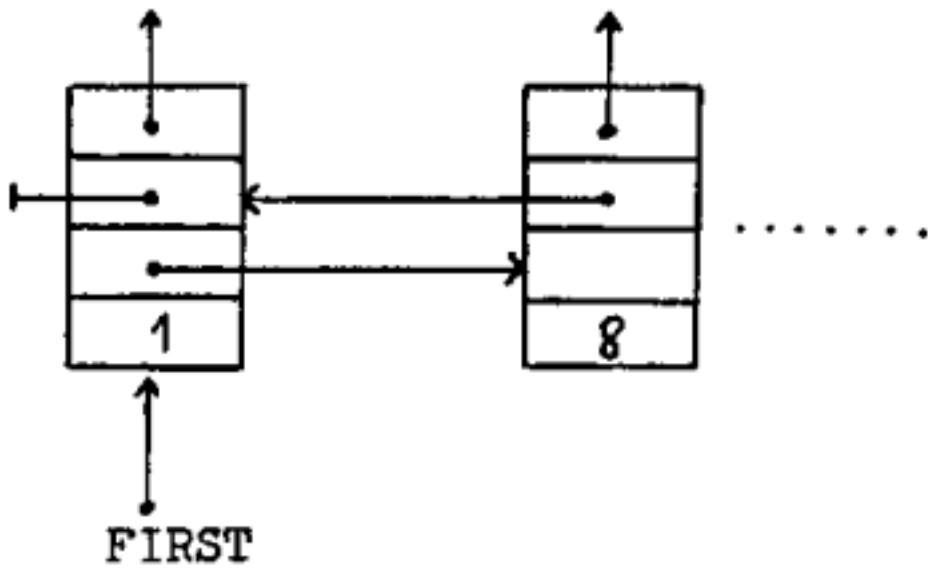
FIRST und LAST vom Typ ref (CELL) markieren das jeweils aktuelle linke bzw. rechte Ende der Liste. Die Funktionsprozeduren

ref (CELL) procedure ADDLEFT; und

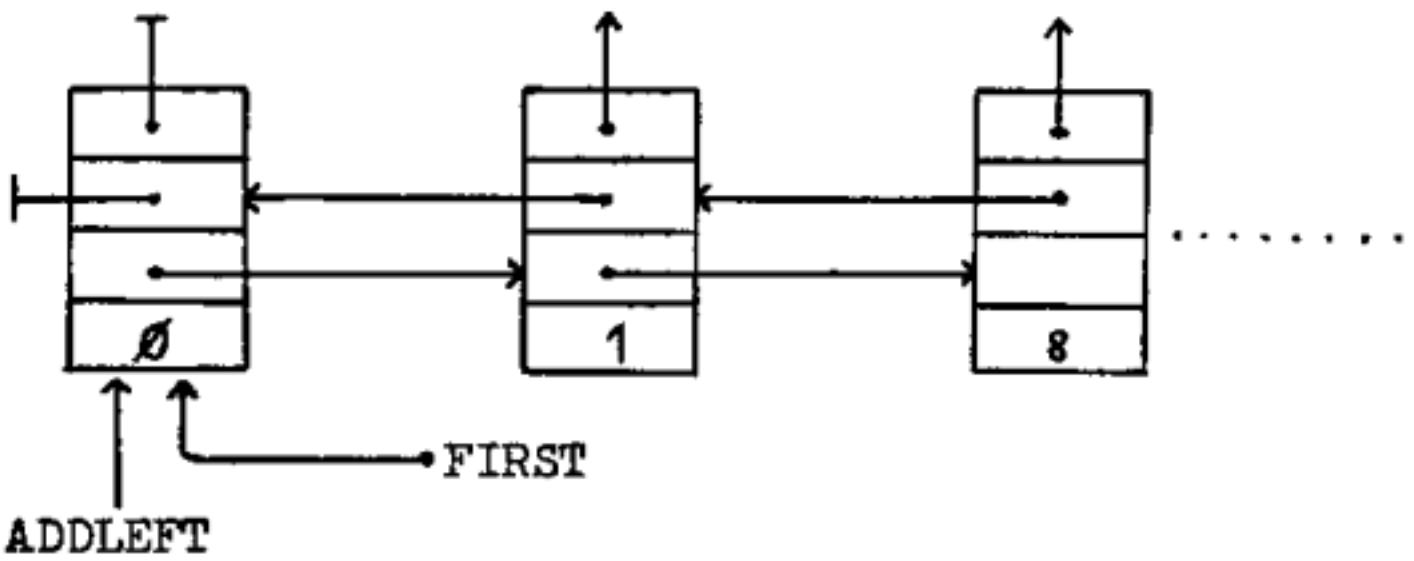
ref (CELL) procedure ADDRIGHT;

bewerkstelligen das Anhängen einer neuen Speicherzelle an die bisherige Liste.

Beispiel:



Situation vor Aufruf von ADDLEFT



ADDLEFT

Situation nach Aufruf von ADDLEFT

Analog ADDRIGHT.

(iii) Zeitverhalten:

Startsituation: Zu Beginn der Simulation wird der noch leere Speicher mit einer festen Anzahl von Programmen (Objekten des Typs PROGRAM) π_i, i=1,...,NUM,

initialisiert. Es gibt M ≤ NUM verschiedene Programme. Daraus ergibt sich, daß schon anfangs Programme mehrfach im Speicher vorhanden sein können. Da in den Speicherzellen die Programme nicht explizit stehen, sondern durch Verweise repräsentiert werden, müssen die Programme π_j irgendwo ausführlich gespeichert sein. Zu diesem Zweck dient ref (PROGRAM) array P[1:M];
Über den Zeiger P[j] besteht immer Zugriffsmöglichkeit auf das Programm π_j. Das Kopieren eines Programms π_j wird durch Setzen eines weiteren Zeigers auf π_j realisiert. Diese Vorgehensweise zwingt fast dazu, vom „Programmtyp" π_j zu sprechen, während die Zeiger auf π_j die eigentlichen Programme oder Exemplare dieses Typs darstellen. Der Sprachgebrauch ist hier nicht eindeutig festzulegen. Ohne sprachliche Verwirrung zu stiften, werden wir deshalb im folgenden π_j sowohl als Programm als auch als Programmtyp bezeichnen, je nachdem, welche Bezeichnung gerade angebracht erscheint. Das Feld integer array ST[1:M] enthält zu jedem Zeitpunkt der Simulation in den Komponenten ST[j] die momentane Anzahl der Exemplare des Programms P[j]. Zu Beginn der Simulation gilt also

$$\sum_{j=1}^{M} ST[j] = NUM$$

Einlesen der NUM Zahlenpaare (PI,WHERE) liefert die räumliche Anordnung der NUM Programme : Das Programm P[PI] steht in der vom Zeiger LEFT aus gerechnet WHERE-ten Speicherzelle. (siehe Abbildung 8.2.2.A)

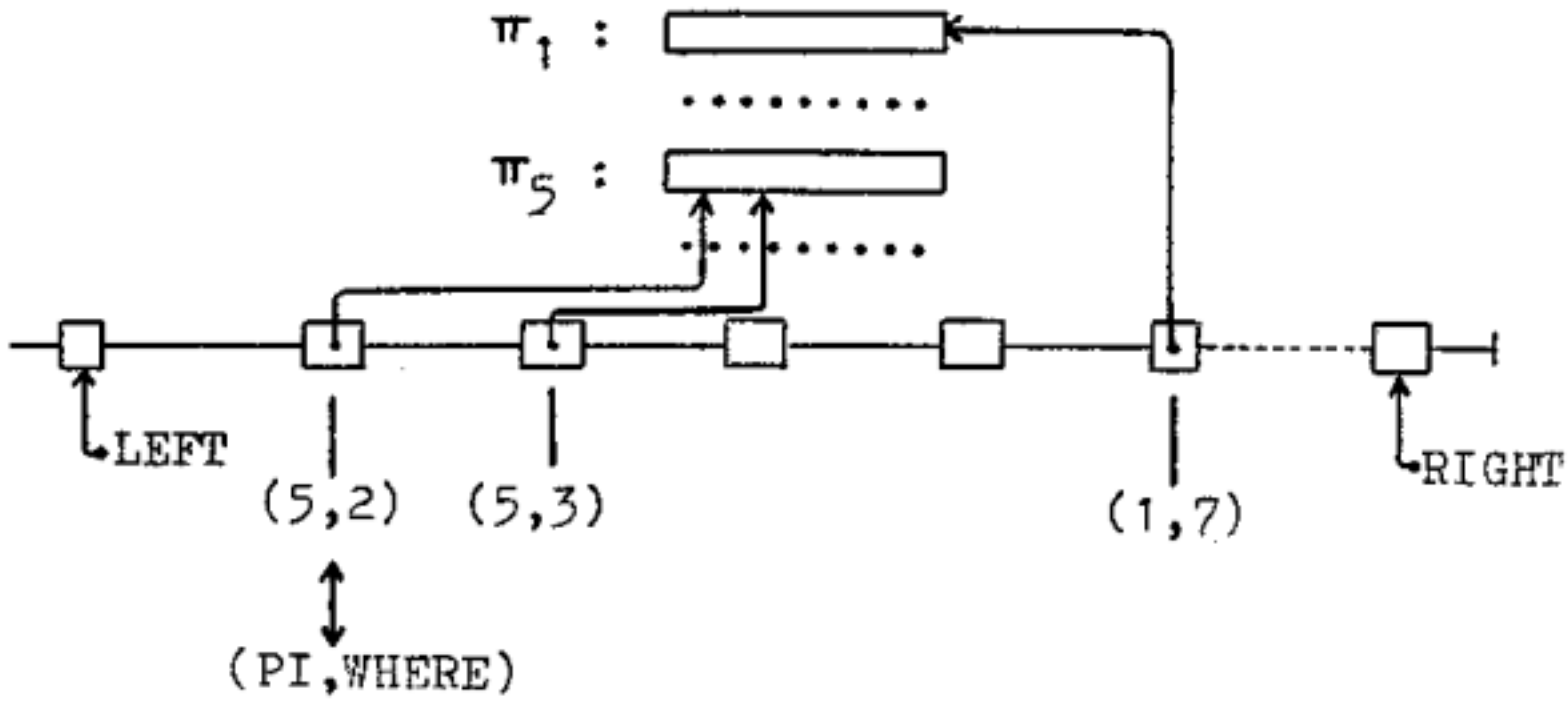


Abb. 8.2.2.A

Simulation: TIME-mal wird der Speicher von links nach rechts mittels des Zeigers C vom Typ ref (CELL) durchlaufen. Bei jeder Zelle wird getestet, ob sie leer ist oder nicht. Ist die Zelle leer, so geschieht nichts. Ist die Zelle durch ein Programm belegt, so wird weiter getestet, ob bei dieser Zelle

TIMECOUNT+1 = CONTENS.DELY

gilt. Im Ja-Fall wird eine Kopie des betreffenden Programms angelegt und TIMECOUNT wieder auf Ø gesetzt (ein neuer Reproduktionszyklus des betreffenden Programms kann beginnen). Andernfalls wird TIMECOUNT um 1 erhöht. Es kann geschehen, daß zum Anlegen einer Kopie der Speicher am rechten Ende verlängert werden muß; dann ist darauf zu achten, daß der Speicher nur bis zu seinem rechten Ende zu Beginn des betreffenden Durchlaufs durchlaufen wird. Erst der Durchlauf, der den nächsten Zyklus simuliert, erfaßt den nun zusätzlichen Speicherbereich. Dies wird durch die Variable

ref (CELL) OLD_LAST

unterstützt.

```
for T:=1 step 1 until TIME do
begin
  C:-FIRST;                  }
  OLD_LAST:-LAST;            }   Initialisierung eines Durch-
                                                        laufs
  while C=/=OLD_LAST.RIGHT do
  begin
    [Erhöhe C.TIMECOUNT um 1];
    if C.TIMECOUNT=C.CONTENS.DELY
    then [Kopiere das Programm C.CONTENS;
          Setze C.TIMECOUNT zurück auf Ø;]
    C:-C.RIGHT
  end
end;
```

(iv) Räumliches Verhalten:

Sei nun beim i-ten Durchlauf, i ≤ TIME, durch den Speicher der Zeiger C auf eine Speicherzelle gestoßen, deren Programm π_j = C.CONTENS reproduktionsbereit ist, das heißt:

C.TIMECOUNT+1 = C.CONTENS.DELY .

π_j wird nun kopiert. Ob die Kopie rechts oder links von der Zelle, auf die C zeigt, angelegt wird, entscheidet die Prozedur COPY mit Hilfe der SIMULA-Zufallszahlenfunktion RANDINT. [1)]

```
procedure COPY(C); ref (CELL) C;
if RANDINT(1,2,U) = 1
then COPY_LEFT(C)
else COPY_RIGHT(C);
```

Entsprechend der getroffenen Entscheidung wird also die Prozedur COPY_LEFT bzw. COPY_RIGHT aufgerufen, die den eigentlichen Kopierprozeß durchführt.

```
procedure COPY_RIGHT(C); ref (CELL) C;
begin
  ref (CELL) HELP;
  [Setze HELP auf C];
  [Bewege HELP um soviele Zellen nach rechts,
  wie die Komponente DISTANCE des Programms
  π_j (=C.CONTENS) angibt. Wird dabei vorzeitig
  das Ende des Speichers erreicht, so erweitere
  mittels ADDRIGHT den Speicher um entsprechend
  viele Zellen an seinem rechten Ende.];
  if HELP.CONTENS==none
  then [Lege die Kopie von π_j in der leeren
       Zelle ab, auf die HELP zeigt
       [= HELP.CONTENS:-P[j]]]
  else [Bewege HELP solange im Speicher weiter
       nach rechts, bis HELP auf eine leere
       Speicherzelle zeigt, oder das rechte
       Ende des Speichers erreicht ist. Ist
       letzteres der Fall, so setze
```

1) Siehe Beschreibung von RANDINT [25] Seite 4.-9

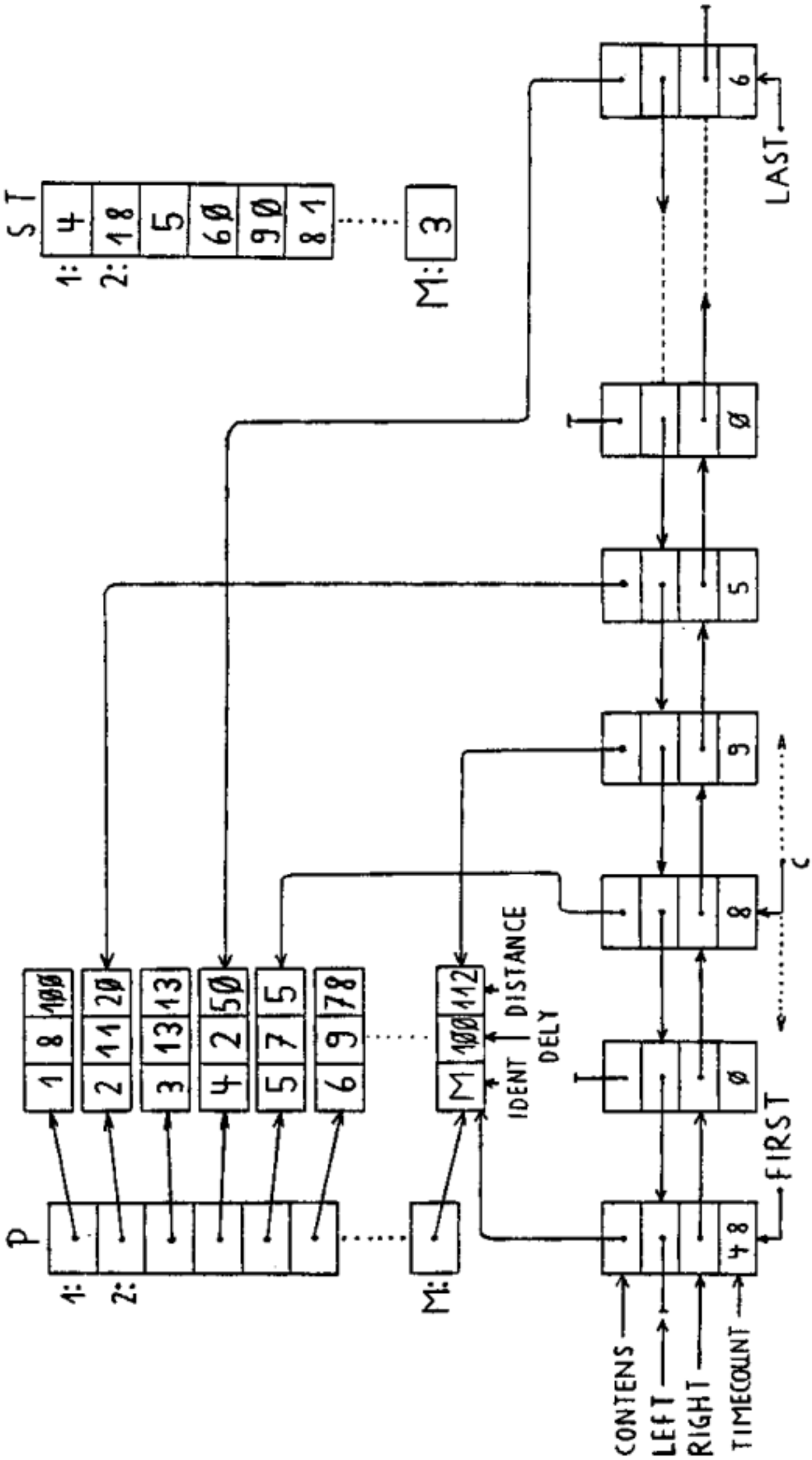

GLOBALE DATENSTRUKTUR IN MOD 1 (BEISPIEL)
ST
1: 4
2: 18
5
60
90
81
M: 3
LAST
6
T
0
5
9
8
C
P
1 8 100
2 11 20
3 13 13
4 2 50
5 7 5
6 9 78
M 100 112
IDENT
DISTANCE
DELY
I
0
FIRST
1:
2:
M:
48
CONTENS
LEFT
RIGHT
TIMECOUNT

Abb. 8.2.2.B

```
                    |HELP:-ADDRIGHT; Lege die Kopie von π_j|
                    |in der Zelle ab, auf die HELP zeigt.  |
          end;
```

Anhang C.1. zeigt MOD1 in ausführlicher Form als lauffähiges SIMULA-Programm. Die Datenstrukturen dieses Programms verdeutlicht Abbildung 8.2.2.B..

Eingabeparameter des SIMULA-Programms für MOD1:

- Die Anfangslänge des Speichers	N
- Die Anzahl der unterschiedlichen Programmtypen	M
- Die M Programme(Typen), charakterisiert durch M Zahlenpaare	DELY,DISTANCE
- Die Anzahl der sich anfangs im Speicher befindlichen Programme	NUM
- Die Verteilung der NUM Programme im Speicher, angegeben durch Num Zahlenpaare	PI,WHERE
- Die Anzahl der vorgesehenen Speicherdurchläufe	TIME

8.2.3.Absichten von MOD1

Simulationsmodelle bzw.-programme haben in der Regel experimentellen Charakter. Die Erkenntnisse, die sich mittels MOD1 gewinnen lassen, sind beschränkt und größtenteils vorherberechenbar; sie benötigen kein Experiment. Der einzige Nichtdeterminismus liegt in der zufälligen Wahl der Richtung, in der die Kopie eines Programms im Speicher angelegt wird. MOD1 ist als Grundmodell zu werten, auf das weitere Modelle mit mehr Möglichkeiten aufbauen. Außerdem demonstriert MOD1 einen gewissen Satz von Grundelementen, die auch den weiteren Modellen MOD2 und MOD3 zu eigen sind. Es handelt sich dabei um

(i) Modell für Programme

(ii) Modell für Speicher

(iii) zeitliches Verhalten

(iv) räumliches Verhalten

Durch Änderung einer oder mehrerer dieser 4 Komponenten lassen sich andere Grundmodelle erzielen; besonders (iv) dürfte sehr viele Variationsmöglichkeiten bieten. (i) bis (iv) sind nicht ganz unabhängig von einander. Die Charakterisierung der Programme durch die Größen DELY und DISTANCE bestimmt das zeitliche Verhalten (iii) (mittels DELY) und das räumliche Verhalten (iv) (mittels DISTANCE) wesentlich mit.

Die Wahl von (i) bis (iv) ist auch vor dem Hintergrund der unterstellten „von Neuman Architektur" des Rechners zu sehen . Eine charakteristische Eigenschaft der von Neuman-Rechner ist die Tatsache, daß es zu jedem Zeitpunkt nur jeweils einen Strom von Instruktionen und Daten gibt. Man spricht daher auch von SISD-Maschinen (single-instruction single-data stream) [12] . Für moderne Rechnersysteme trifft diese Charakterisierung jedoch nicht ganz zu, da durch Hinzunahme weiterer Prozessoren, speziell E/A-Prozessoren, das SISD-Prinzip durchbrochen wird; es liegt eigentlich MIMD-Organisation (multiple-instruction multiple-data stream) vor. Trotzdem wird man heutige Rechner kaum als MIMD-Rechner bezeichnen, da die Anzahl der gleichzeitig arbeitenden Prozessoren sehr klein ist. Die Idee, die hinter MIMD steht, ist jedoch eine große Zahl (ca. 100 bis 1000) [26] unabhängiger Prozessoren, um ein Höchstmaß an Parallelverarbeitung zu erzielen. Neben SISD- und MIMD-Maschinen gibt es außerdem noch die Klassen der SIMD (single-instruction multiple-data stream)- und der MISD (multiple-instruction single-data stream)-Maschinen. Nahezu alle heutigen Rechner sind im weitesten Sinne SISD-Maschinen. Es entsteht also die Frage:

Welche Modelleigenschaften müßte MOD1 haben, wenn

MOD1 als Grundmodell für unorthodoxe Rechenanlagen (=nicht SISD-Rechner) [6] gedacht wäre?

Voraussetzung ist narürlich die Existenz selbstreproduzierender Programme auf solchen Maschinen.

8.2.4.Einige Aspekte des SIMULA-Programms für MOD1

I. Wie in 8.2.3. erwähnt liefert MOD1 keine experimentellen Ergebnisse bzgl. des zahlenmäßigen Verhaltens der einzelnen Programmtypen. Da in MOD1 alle Programme beständig sind und alle reproduktionsfähigen Exemplare sich unbedingt reproduzieren, gilt für jeden Programmtyp π_j, $j \in [M]$:

Die Anzahl $S_j(T)$ der Exemplare nach dem T-ten Speicherdurchlauf beträgt

$$S_j(0) \cdot 2^{(T \div \text{DELY-Komponente von } \pi_j)} ,$$

wobei $S_j(0)$ die Anzahl der Exemplare von π_j vor Beginn der Simulation ist.

$S_j(T)$ kann für jedes π_j nach jedem WHEN_CON-ten Speicherdurchlauf mittels

procedure CONTROL;

in tabellarischer Form ausgedruckt werden.

II. In Anbetracht des ungehinderten numerischen Anwachsens der Programmzahl ist sicherlich die räumliche Anordnung der einzelnen Exemplare von größerem Interesse. Das räumliche Verhalten (iv) in MOD1 ist recht willkürlich gewählt und soll hier auch nicht näher analysiert werden; andere räumliche Verhalten sind denkbar. Dem SIMULA-Programm für MOD1 sind daher zwei Prozeduren beigefügt, die unabhängig vom gewählten räumlichen Verhalten dessen Analyse unterstützen. Es handelt sich dabei um

procedure DUMP;

DUMP gibt den Inhalt des gesamten Speichers von links nach rechts aus, indem für eine leere Zelle das Zeichen '*' und für eine besetzte Zelle die Komponente IDENT des gespeicherten Programms ausgedruckt wird.

procedure AVERAGE;

AVERAGE gibt in tabellarischer Form die durchschnittliche Entfernung der Exemplare der jeweiligen Programmtypen an. Grundlage ist der Abstand 1 für direkt benachbarte Speicherzellen.

Die Aufrufe von DUMP und AVERAGE werden durch die integer-Variablen WHEN_DUM und WHEN_AVE gesteuert. DUMP wird nach jedem WHEN_DUM-ten Speicherdurchlauf aufgerufen, entsprechend AVERAGE.

III. Aus I und II ergibt sich, daß die in Anhang C.1. wiedergegebene Implementierung von MOD1 drei modellunabhängige Eingabeparameter, nämlich

WHEN_CON

WHEN_DUM

WHEN_AVE

enthält, die die Ausgabe steuern.

IV. Aufwand:

Speicherplatz: In MOD1 ist die Anzahl der vorhandenen Programmtypen konstant. Damit haben auch die Felder ST und P während der Simulation eine konstante Größe. Nur die den Speicher darstellende Liste wird - bei hinreichend großer Anzahl TIME der Speicherdurchläufe - während der Simulation anwachsen. Wie stark dieses Wachstum während eines Speicherdurchlaufs ist, hängt von den vorhandenen Programm-

typen ab. Bei kleinen Reproduktionszeiten (Komponente DELY) der Programme und großen Entfernungen der Kopien (Komponente DISTANCE) erfolgt die Zunahme besonders schnell. Da die Gesamtzahl der vorhandenen Programmexemplare (siehe I.) exponentiell ansteigt, wächst auch die Länge des Speichers im Verlauf der Simulation insgesamt exponentiell; die vorgenannten Kriterien machen daher nur einen Faktor aus.

Laufzeit: Die Laufzeit des SIMULA-Programms für MOD1 hängt von allen Modellparametern ab. Eine genaue Abschätzung kann im Rahmen dieser Arbeit nicht mehr vorgenommen werden. Da der Zeitaufwand für einen Speicherdurchlauf von der Länge des Speichers abhängt, ergibt sich auf jeden Fall ein exponentieller Zusammenhang zwischen der Anzahl der Speicherdurchläufe und der Laufzeit von MOD1.

Das exponentielle Verhalten des Aufwands macht MOD1 für statistische Zwecke wenig brauchbar und unterstreicht die Bedeutung von MOD1 als ein lediglich Grundmodell. Das ungehinderte exponentielle Anwachsen an Programmexemplaren wird in den folgenden Modellen durch ein geändertes räumliches Verhalten und durch Einführung von konkurrierendem Verhalten verhindert.

8.3.Ein Modell mit konkurrierendem Verhalten

Im Grundmodell MOD1 kann jedes reproduktionsfähige Programm π seine Kopie $\overline{\pi}$ ungehindert in einer freien Speicherzelle ablegen. Insofern gibt es zwischen den Programmen keine echten Konfliktsituationen und damit keine Konkurrenz. Das Fehlen von Konkurrenz in MOD1 macht das Eintreten von Evolution unmöglich. Ein weiterer Aspekt von MOD1 ist die Beständigkeit der Programme.

In Form von MOD2 soll nun MOD1 dahingehend erweitert werden, daß Programme in der Lage sind, ihre Kopien in bereits durch andere Programme besetzte Speicherzellen zu schreiben. Dabei werden die alten Inhalte der Speicherzellen, also Programme, gelöscht. Es wird zweierlei erreicht:

- Es gibt Konflikte zwischen den Programmen. Die Konsequenz ist Konkurrenz.
- Das Überschreiben von Programmen bedeutet deren Vernichtung. Es tritt also eine Art „Sterben" von Programmen auf.

Konkurrierendes Verhalten ist ein spezielles Verhalten von Programmen untereinander. Wir führen daher ganz allgemein

(v) Verhalten der Programme untereinander

als weitere Modellkomponente ein. In 8.3.1. erfolgt eine detaillierte Beschreibung von MOD2.

8.3.1.Informelle Beschreibung von MOD2

(i) Programme: Die einzige charakteristische Größe eines Programms ist seine Laufzeit (=Reproduktionszeit). Die Laufzeit ist gleich der Anzahl von Zeittakten, die das Programm jeweils aktiv sein muß, um sich reproduzieren zu können.

Das Modell eines Programms ist somit ein 2-Tupel, bestehend aus der Programmidentifikation und der Laufzeit.

(ii) Speicher: Wie in MOD1.

(iii) Zeitverhalten: Wie in MOD1.

(iv) Räumliches Verhalten: Zu jedem Zeitpunkt t sind nur endlich viele Speicherzellen belegt. Unter den belegten Speicherzellen gibt es daher immer eine am weitesten links und eine am weitesten rechts stehende Speicherzelle l(t) bzw. r(t). l(t) und r(t) begrenzen den Speicherbereich, in dem sich belegte Zellen befinden. Die Speicherzelle, in der ein Programm π seine Kopie $\overline{\pi}$ ablegt, soll nicht beliebig von diesem abgegrenzten Speicherbereich entfernt liegen, sondern nur um eine konstante Anzahl von Zellen. Es ergibt sich also als Zielbereich für eine Kopie ein durch L(t) und R(t) abgegrenzter Speicherbereich (Abb. 8.3.1.A). Innerhalb dieses Speicherbereichs ist jede Zelle gleichwahrscheinliches Ziel für die Kopie $\overline{\pi}$.

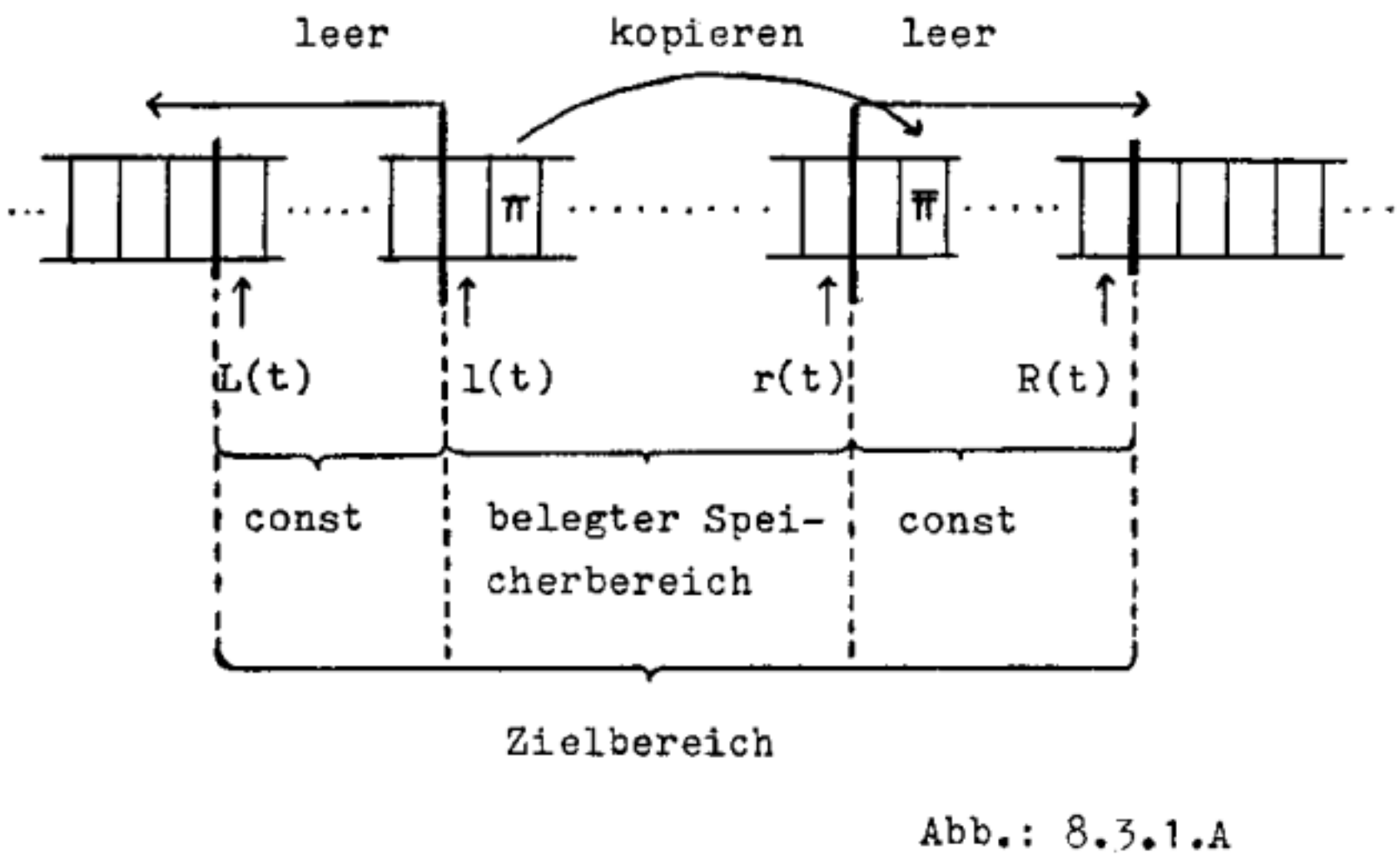


Abb.: 8.3.1.A

Ein derartiges räumliches Verhalten garantiert eine kontrollierte Ausbreitung der Programme. Es können nicht plötzlich beliebig weit entfernte und isolierte „Populationen" entstehen.

(v) Verhalten der Programme untereinander: Ist ein Programm π reproduktionsfähig, so wird gemäß (iv) eine beliebige Speicherzelle ausgewählt, in der die Kopie $\overline{\pi}$ abgelegt werden soll. Ist diese Speicherzelle leer, so tritt kein Konflikt auf. Andernfalls ist die Speicherzelle bereits mit einem Programm π' besetzt, und ein Entscheidungsmechanismus muß herangezogen werden um zu bestimmen, ob $\overline{\pi}$ das Programm π' überschreiben darf oder nicht. Fällt die Entscheidung positiv aus, so überschreibt $\overline{\pi}$ das Programm π', ist sie negativ, so hat $\overline{\pi}$ keine Ausweichmöglichkeit und wird eliminiert. Ein Konfliktfall endet also immer für eines der beteiligten Programme „tödlich".

Erst durch Angabe des in (v) genannten Entscheidungsmechanismus wird die Beschreibung von MOD2 vollständig. Es sind sicherlich sehr viele Entscheidungsmechanismen für MOD2 denkbar, die auf unterschiedlichsten Faktoren beruhen. Wir wollen, um MOD2 möglichst einfach zu halten, einen Entscheidungsmechanismus angeben, der nur auf den beiden jeweils aufeinandertreffenden Programmen beruht. Damit dieser Entscheidungsmechanismus nicht zu starr wird, wird er mit Wahrscheinlichkeiten belegt, was indirekt doch eine Berücksichtigung weiterer, allerdings unbekannter, Faktoren bedeutet.

(8.3.1.1) Definition: Sei $n \in \mathbb{N}$. Eine $n \times n$-Matrix $V = (v_{ij}) \in \mathcal{M}_n(\mathbb{R})$ (Ring der n-reihigen Matrizen über dem Körper der reellen Zahlen) heißt n-reihige Vorrangmatrix, falls gilt:

$$v_{ij} \in [0,1] \subset \mathbb{R} \quad \text{für alle } i,j \in [n]$$

Sei $\mathbb{P} := \{\pi_1, \ldots, \pi_M\}$ die Menge der in MOD2 vorkommenden Programmtypen. Eine M-reihige Vorrangmatrix zusammen mit einer entsprechenden Interpretation liefert einen Entscheidungsmechanismus für MOD2.

(8.3.1.2) Definition: Sei M die Anzahl der in MOD2 vorkommenden Programmtypen. Sei $V = (v_{ij})$ eine M-reihige Vorrangmatrix. Die Komponenten v_{ij} werden wie folgt interpretiert:
Soll die Kopie $\overline{\pi}$ eines Programms π vom Typ π_i in einer Speicherzelle abgelegt werden, in der sich bereits ein Programm π' des Typs π_j befindet, so bedeutet v_{ij}:
Mit der Wahrscheinlichkeit v_{ij} überschreibt $\overline{\pi}$ das Programm π'.
Mit der Wahrscheinlichkeit $(1-v_{ij})$ überschreibt $\overline{\pi}$ das Programm π' nicht, π' bleibt erhalten und $\overline{\pi}$ wird eliminiert.

Als Entscheidungsmechanismen für MOD2 sind genau die M-reihigen Vorrangmatrizen zugelassen. Die Vorrangmatrix ist also ein Parameter von MOD2. Durch die Vorrangmatrix erhält MOD2 einen nichtdeterministischen Charakter.

8.3.2.MOD2 als SIMULA-Programm

(i) Programme:

Im Gegensatz zu MOD1 werden Programme durch die einfachere SIMULA-Struktur

```
class PROGRAM;
      begin
      integer IDENT,DELY;
      end;
```

Identifizierung, Reproduktionszeit

dargestellt.

(ii) Speicher:

Die Realisierung des Speichers ist mit Schwierigkeiten verbunden. Einerseits muß der Speicher po-

tentiell unendlich sein (Listenkonzept), andererseits ist wegen 8.3.2.(iv) direkter Zugriff auf die Speicherzellen wünschenswert (Arraykonzept). Da Listenkonzept und Arraykonzept nicht miteinander vereinbar sind, muß bei einer Kompromißlösung irgendwo die Priorität gesetzt werden. Direkter Zugriff wirkt sich günstig auf die Laufzeit von MOD2 aus. Wir setzen deshalb hier die Priorität und stellen den Speicher als array dar. Um der geforderten Unendlichkeit des Speichers wenigstens gerecht zu werden, muß es sich dabei um dynamische arrays handeln. Dynamische arrays sind in SIMULA im Rahmen des Klassenkonzepts möglich:

```
class STORAGE(Q); integer Q;
      begin
      ref (CELL) array ELEMENT(1:Q);
      end;
```

wobei die einzelnen Speicherzellen (Typ : CELL) wie in MOD1 dargestellt werden:

```
class CELL;
      begin
      ref (PROGRAM) CONTENS;
      integer TIMECOUNT;
      end;
```

Zugriff auf den Speicher liefert der globale Zeiger ref (STORAGE) STOREPOINTER.
Zu Beginn der Simulation wird der Speicher mit N Speicherzellen initialisiert. Dies geschieht durch die Zuweisung

```
STOREPOINTER:-new STORAGE(N);
```

Während der Simulation wird der Speicher immer dichter mit Programmen „besiedelt" und muß von Zeit zu Zeit erweitert werden. Wann der Speicher erweitert wird, gibt die integer Variable PERCENT an: Ist der

Speicher zu PERCENT % belegt - getestet wird dies mit Hilfe von boolean procedure OVERFLOW - , so wird eine Erweiterung des Speichers vorgenommen. Die Erweiterung des Speichers wird durch die Prozedur

```
procedure NEW_STORAGE(MORE); integer MORE;
```

vorgenommen. NEW_STORAGE generiert ein neues Objekt der Länge N+2*MORE vom Typ STORAGE, kopiert den Inhalt des alten Speichers in dieses neue Objekt und setzt den globalen Zeiger STOREPOINTER entsprechend um (Abb. 8.3.2.A). Nach Ablauf von NEW_STORAGE ist der Speicher an jedem Ende um MORE Zellen erweitert und die Variable N, die immer die aktuelle Länge des Speichers angibt, um 2*MORE vergrößert.

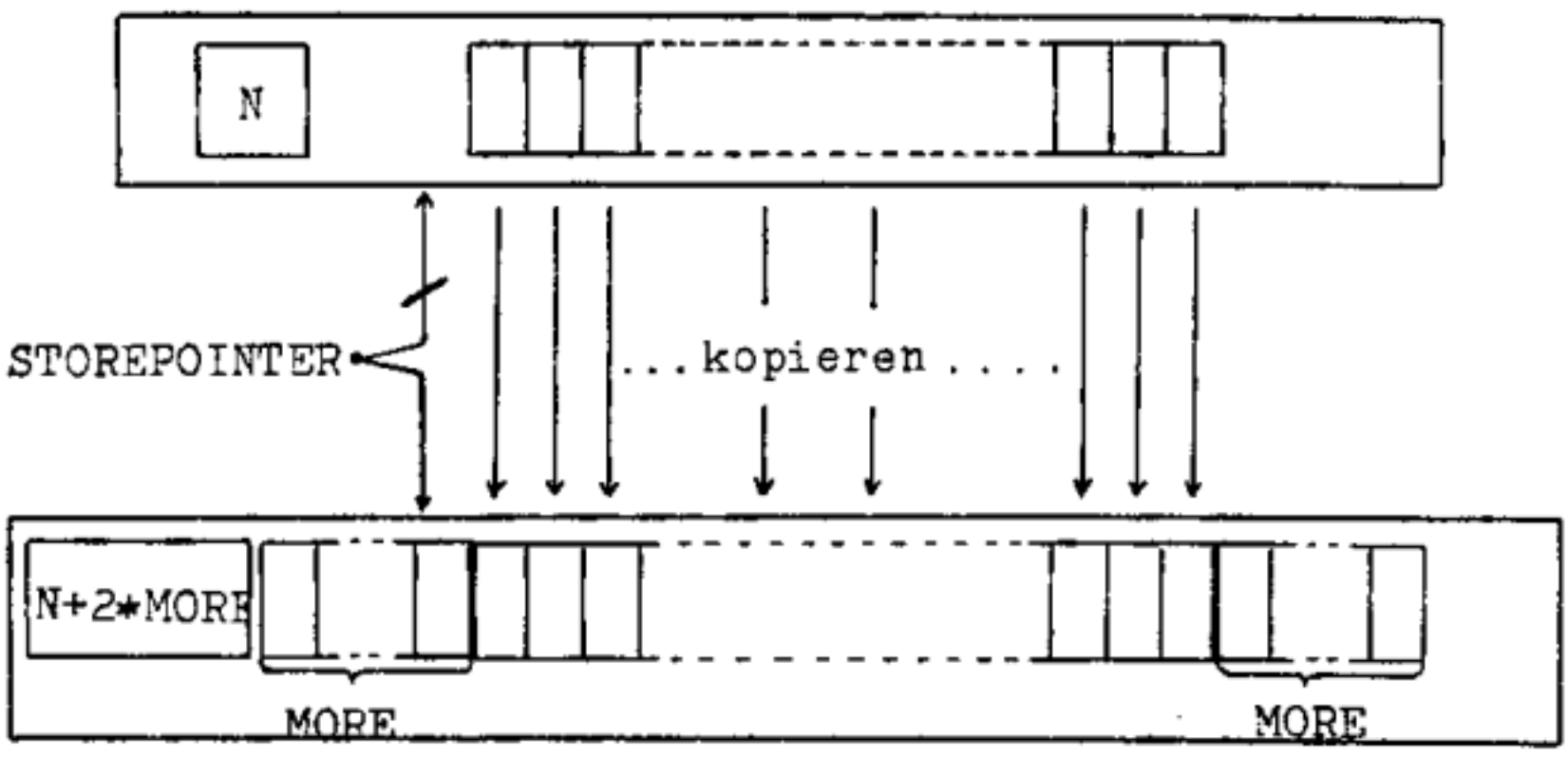


Abb.: 8.3.2.A

Die den Mechanismus der Speichererweiterung bestimmenden Größen PERCENT und MORE sind wichtige Parameter von MOD2 als SIMULA-Programm. Je kleiner PERCENT und je größer MORE gewählt sind, desto besser wird die Unendlichkeit des Speichers angenähert. Setzt man PERCENT > 100, so artet die Reali-

sierung von MOD2 in ein Modell mit endlichem Speicher aus.

(iii) Zeitverhalten:

Startsituation: Zu Beginn der Simulation werden wie in MOD1 die M Programme (besser: Programmtypen) π_1, ..., π_M eingelesen und im Feld

```
ref (PROGRAM) array P[1:M];
```

abgespeichert. Das Feld

```
integer array ST[1:M]
```

enthält zu jedem Zeitpunkt der Simulation in den Komponenten ST[j] die momentane Anzahl der im Speicher befindlichen Exemplare des Programms π_j. Die Anfangsbelegung von ST wird ebenfalls eingelesen, da sie angibt, mit wievielen Exemplaren der einzelnen Programme der Speicher initialisiert wird. Der leere Speicher mit der Startlänge N wird initialisiert, indem für jedes $j \in [M]$ die ST[j] Exemplare des Programms π_j in den Speicher geschrieben (durch Setzen von Verweisen) werden. Die zufällige Verteilung der Programme im Speicher wird mittels des Zufallszahlengenerators RANDINT(1,N,U) gewährleistet; jede Speicherzelle ist gleichwahrscheinlich. Es wird jedoch verhindert, daß bereits bei Initialisierung Programme überschrieben werden. Selbstverständlich muß gelten:

$$\sum_{j=1}^{M} ST[j] \leq N$$

Zum Abschluß der Initialisierung wird die M-reihige Vorrangmatrix eingelesen und im Feld

```
CONFLICT[1:M,1:M]
```

abgespeichert.

Simulation: TIME-mal wird der Speicher von rechts nach links bzw. von links nach rechts durchlaufen. Jede Durchlaufrichtung ist gleichwahrscheinlich.

Durch zufällige Wahl der Durchlaufrichtung soll eine Bevorzugung einzelner Programme vermieden werden. Während eines Durchlaufs wird für jede nicht leere Speicherzelle die Prozedur MATCH aufgerufen. MATCH testet, ob das in der betreffenden Speicherzelle befindliche Programm reproduktionsfähig ist und legt eventuell eine Kopie des Programms an. Damit ist es möglich, daß ein Aufruf von MATCH die prozentuale Speicherbelegung größer als PERCENT werden läßt und ein Aufruf von NEW_STORAGE notwendig wird:

```
for T=1 step 1 until TIME do
begin
   [Lege Durchlaufrichtung fest];
  if [Durchlaufrichtung = 'von rechts nach links']
  then
  begin
    for I:=1 step 1 until N do
    if [I-te Zelle ungleich leer]
    then
    begin
      MATCH(I);
      if OVERFLOW then NEW_STORAGE(MORE)
    end
  end
  else
  for I:=N step -1 until 1 do
  if [I-te Zelle ungleich leer]
  then
  begin
    MATCH(I);
    if OVERFLOW then NEW_STORAGE(MORE)
  end
end *** S I M U L A T I O N ***;
```

Funktionsweise der Prozedur MATCH:

```
procedure MATCH(I); integer I;
begin
  :
  boolean IS_COPY;
  IS_COPY:=false;
   [Erhöhe TIMECOUNT-Komponente der I-ten Speicher-
   zelle um 1 ;
  if [TIMECOUNT-Komponente der I-ten Speicherzelle
      gleich DELY-Komponente des in der I-ten Spei-
      cherzelle befindlichen Programms
  then
begin
  comment  *** Reproduktion des in der I-ten
               Zelle befindlichen Programms *** ;
   [Setze TIMECOUNT-Komponente der I-ten Spei-
   cherzelle auf Ø                              ];
   [Wähle eine zufällige Zelle des Speichers aus.
   Sei die Wahl auf die W-te Speicherzelle ge-
   fallen                                       ];
  comment *** siehe dazu unten (iv) *** ;
  if [W-te Speicherzelle gleich leer]
  then
  begin
    comment *** Ungehindertes Ablegen der Kopie *** ;
     [Schreibe Kopie in die W-te Speicherzelle] ;
   IS_COPY:=true
  end
  else
  begin
    comment *** W-te Speicherzelle ist bereits
                besetzt *** ;
     [Treffe Entscheidung mittels der Vorrangma-
     trix, ob die Kopie des in der I-ten Zelle
     befindlichen Programms das Programm in der
     W-ten Zelle überschreiben darf             ];
   comment *** siehe unten (v) *** ;
```

```
    if [Überschreiben nicht möglich]
    then comment *** es geschieht nichts ***
    else
    begin
       [Schreibe Kopie in die W-te Speicherzelle];
      IS_COPY:=true
    end
  end;
  comment *** Falls das Programm aus Speicherzelle
              I seine Kopie im Speicher ablegen
              konnte, muß noch in Abhängigkeit von
              der Durchlaufrichtung die Komponente
              TIMECOUNT der W-ten Speicherzelle
              gesetzt werden *** ;
  if IS_COPY
  then
  begin
    if W ≤ I
    then
    begin
      if [Durchlaufrichtung = 'von links nach rechts']
      then [TIMECOUNT-Komponente von Zelle W:=Ø]
      else [TIMECOUNT-Komponente von Zelle W:=-1]
    end
    else
    if [Durchlaufrichtung = 'von links nach rechts']
    then [TIMECOUNT-Komponente von Zelle W:=-1]
    else [TIMECOUNT-Komponente von Zelle W:=Ø]
    end
  end
end *** M A T C H ***;
```

(iv) Räumliches Verhalten:

Die Auswahl der Speicherzelle, in die ein Programm seine Kopie ablegt, erfolgt über die SIMULA-Zufallsfunktion RANDINT in Form des Aufrufs

RANDINT(1,N,U_CELL),

wobei N die aktuelle Länge des Speichers und U_CELL ein von RANDINT benötigter name-Parameter ist. Jede der N Speicherzellen ist gleichwahrscheinlich (vgl. genaue Beschreibung der Funktion RANDINT in [25]). Die Auswahl der Speicherzellen für die Kopie weicht geringfügig von der in 8.3.1.(iv) beschriebenen ab. Zusammen mit der Speichererweiterungsstrategie aus 8.3.2.(iii) ergibt sich jedoch der in 8.3.1.(iv) beschriebene Gesamteffekt.

(v) Verhalten der Programme untereinander:

In der SIMULA-Version von MOD2 wird die Vorrangmatrix $V = (v_{ij})$ nicht als Element aus $\mathcal{M}_M(\mathbb{R})$, sondern als Element aus $\mathcal{M}_M(\mathbb{N})$ dargestellt:

```
integer array CONFLICT[1:M,1:M]
```

Jedes v_{ij} (=CONFLICT[i,j]) wird als „v_{ij} Hundertstel" interpretiert. Daher nimmt jedes v_{ij} höchstens den Wert 100 an. $v_{ij} = 0$ kann aus programmtechnischen Gründen nicht zugelassen werden (Abweichung von 8.3.1.(v)).

Beispiel:

M = 5

Konflikt eines Programms π vom Typ π_2 mit einem Programm π' vom Typ π_3, d.h. π versucht π' zu überschreiben:
Entscheidung (vgl. Beschreibung der Prozedur MATCH):

```
if CONFLICT[2,3]<RANDINT(1,100,U_CONFLICT)
then [π überschreibt π' nicht]
else [π überschreibt π']
```

Anhang C.2. zeigt MOD2 als ausführlich kommentiertes SIMULA-Programm. Einen Überblick über die in diesem Programm benutzten Datenstrukturen gibt Abb. 8.3.2.B..

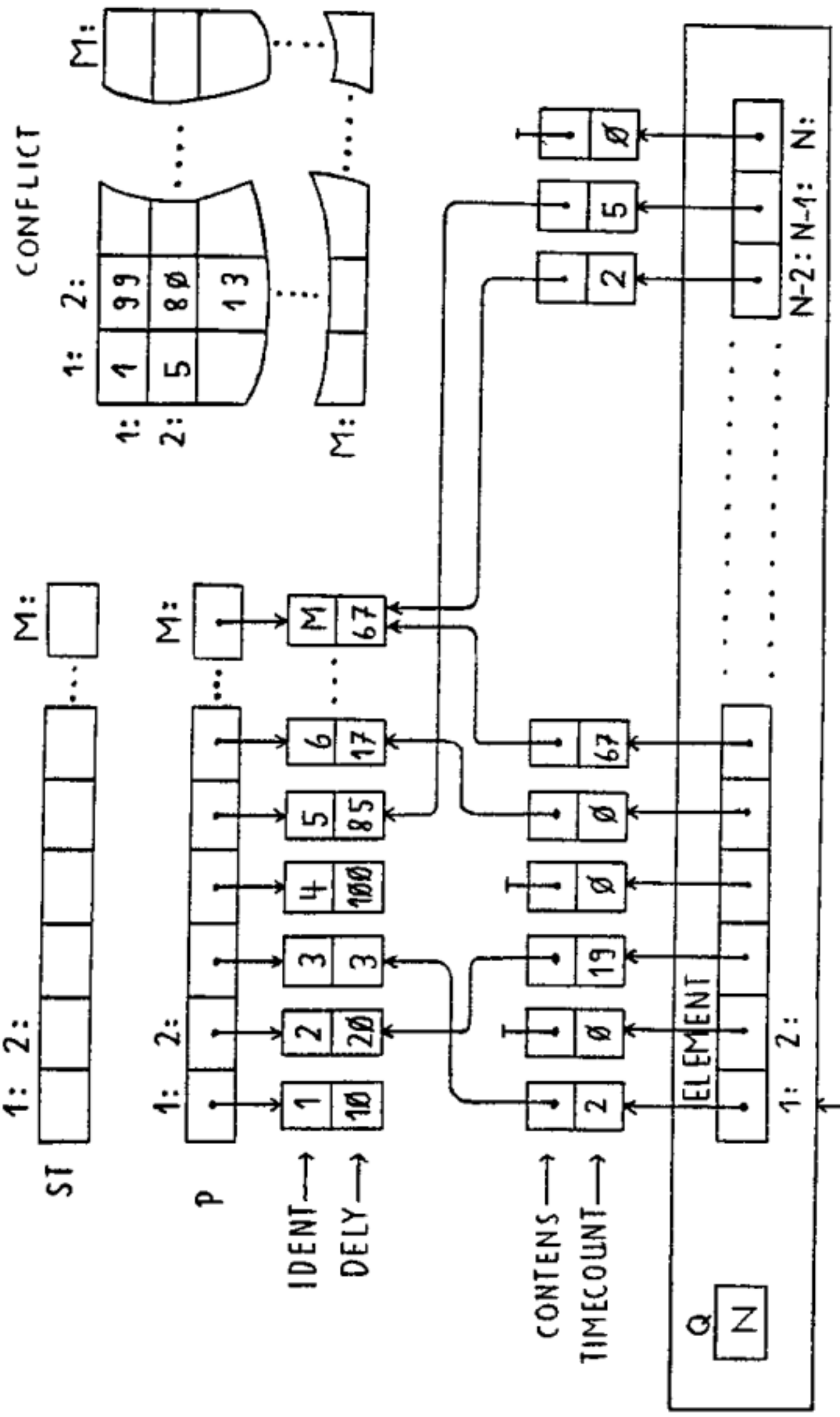

GLOBALE DATENSTRUKTUREN IN MOD2 (BEISPIEL)
STOREPOINTER
ELEMENT
1: 2:
N-2: N-1: N:
1: 2:
M:
CONFLICT
1: 2:
1: 2:
M:
M:
SI
P
IDENT
DELY
CONTENS
TIMECOUNT
Q
Z

Abb. 8.3.2.B

Eingabeparameter des SIMULA-Programms für MOD2:

- Die Anfangslänge des Speichers	N
- Die Anzahl der unterschiedlichen Programmtypen	M
- Die M Programmtypen, charakterisiert durch die Größe DELY, und ihre jeweilige Anfangshäufigkeit	DELY,ST[...]
- Die M×M Elemente der Vorrangmatrix. Jedes Element ist aus [100]	CONFLICT
- Die Anzahl der vorgesehenen Speicherdurchläufe	TIME
- Die Speicherparameter	MORE,PERCENT

8.3.3.Einige Aspekte des SIMULA-Programms für MOD2

I. Das SIMULA-Programm für MOD2 gestattet Simulationen sowohl mit endlichem als auch mit unendlichem Speicher (abhängig von PERCENT).

II. Zur Unterstützung der Ausgabe werden die Prozeduren DUMP und CONTROL in das Programm eingefügt. Daher enthält das Programm die beiden modellunabhängigen Parameter

WHEN_DUM und WHEN_CON (vgl. 8.2.4.I.undII.)

III. Mit Hilfe des Programms für MOD2 lassen sich gewisse Fragestellungen experimentell untersuchen.
Z.B.:

- Inwieweit kann eine relativ schwache Stellung eines Programmtyps in der Vorrangmatrix durch eine kleine Reproduktionszeit kompensiert werden, so daß sich dieser Programmtyp gegenüber seinen Konkurrenten behaupten kann?
- Seien M Programmtypen, repräsentiert durch ihre

DELY-Komponenten, und eine entsprechende Vorrangmatrix gegeben. Wie entwickelt sich das anzahlmäßige Verhältnis der Exemplare der einzelnen Programmtypen bei fortschreitender Simulationsdauer? Wie lange dauert es, bis der eine oder andere Programmtyp ausgemerzt ist? Kann ein Programmtyp nach endlicher Simulationsdauer alle anderen Programmtypen verdrängen?

- Fragen der obigen Art in Abhängigkeit von der „Populationsdichte" (gesteuert durch die Speicherparameter MORE und PERCENT).
- viele weitere Fragen.

Das SIMULA-Programm für MOD2 bietet ein weites Experimentierfeld, was schon aus der Vielzahl der Eingabeparameter ersichtlich ist. Leider kann keine der obigen Fragestellungen im Rahmen dieser Arbeit mehr näher untersucht werden.

IV. Aufwand:

Speicherplatz: Die Anzahl der in MOD2 vorhandenen verschiedenen Programmtypen bleibt während der gesamten Simulation konstant. Damit bleibt auch der durch die Felder CONFLICT, ST und P bedingte Speicherplatzaufwand konstant. Nur das den Speicher simulierende dynamische Feld, auf das der Zeiger STOREPOINTER verweist, kann während der Simulation größer werden. In welchem Maße dieses Feld wächst, hängt von den Parametern MORE und PERCENT ab (vgl. 8.3.2.(ii)). Ist PERCENT > 100 gewählt, so bleibt die Größe des Feldes immer konstant, andernfalls nimmt die Größe im Verlauf der Simulation zu. Die Zunahme erfolgt exponentiell mit der Anzahl der Speicherdurchläufe. Der Faktor (100-PERCENT)/100 (=relative Anzahl der freien Speicherzellen) sorgt jedoch dafür, daß diese Zunahme nicht so ungehemmt erfolgt wie im SIMULA-Programm zu MOD1. Zu beachten ist, daß eine Vergrößerung des Feldes immer mit der Generierung eines neuen Objekts vom Typ STORAGE verbunden ist (Aufruf von NEW_STORAGE). Das jeweils alte Objekt

bleibt im Speicher vorhanden.

Laufzeit: Die Laufzeit ist im wesentlichen von der Anzahl der Speicherdurchläufe und der Länge des den Speicher simulierenden Feldes abhängig. Da die Größe dieses Feldes exponentiell zur Anzahl der Speicherdurchläufe wächst, hängt auch die Laufzeit exponentiell von der Anzahl der Speicherdurchläufe ab. Auch in bezug auf die Laufzeit hat der Faktor (100-PERCENT)/100 eine hemmende Wirkung. Extremfälle:

a) Die Länge des Speichers ist nicht beschränkt, aber der Speicher ist zu jedem Zeitpunkt der Simulation relativ wenig belegt (PERCENT klein gewählt). Dann sind Konflikte relativ selten und die Programme können ihre Kopien nahezu ungehindert ablegen; die Gesamtzahl der Programmexemplare steigt fast ungehemmt exponentiell an. Die Situation ist dann mit der in MOD1 vergleichbar (vgl. 8.2.4.IV.).

b) Die Länge des simulierten Speichers ist konstant (PERCENT > 100). Dann sind von irgendeinem Speicherdurchlauf an alle Speicherzellen besetzt. Die Laufzeit ist dann im wesentlichen proportional zur Anzahl der Speicherdurchläufe, da die Laufzeit der Prozedur MATCH (eine andere wird nicht mehr aufgerufen) durch eine Konstante beschränkt ist.

V. Beim Aufruf von NEW_STORAGE wird der Speicher um die konstante Anzahl von 2*MORE Elementen erweitert. Günstiger wäre es wohl, wenn die Anzahl der zusätzlichen Elemente in einem konstanten prozentualen Verhältnis zur jeweils momentanen Länge des Speichers stehen würde.

9.Evolution bei Programmen

9.1.Motivation

Lebende Pflanzen- und Tierorganismen waren nicht immer so beschaffen wie heute. Vielmehr haben sie sich unter langsamer, aber stetiger Abwandlung ihrer Eigenschaften aus anderen (einfacheren) Lebewesen entwickelt. Man bezeichnet diesen Vorgang als biologische Evolution. Evolution findet auch heute noch, allerdings kaum merklich, statt. Eine kausale Erklärung für die biologische Evolution versucht die Evolutionstheorie anzugeben. Die moderne Evolutionstheorie läßt sich in wenigen Worten wie folgt darlegen (vgl. [24] [13] [27] [15]) :

Lebewesen erzeugen viel mehr Nachkommen, als zur Erhaltung ihrer jeweiligen Art notwendig wäre. Diese Nachkommen variieren in ihrem Genbestand (s. 7.2.). Auch Nachkommen derselben Eltern sind in der Regel nicht alle gleich. Die Veränderlichkeit des Genbestands wird durch die Fähigkeit der Gene zur Mutation (s. 7.2.) bewirkt. Die Mutationsrate lebender Organismen ist äußerst gering und liegt bei etwa 10^{-4} bis 10^{-7} pro Gen. (Diese Werte gelten unabhängig von der Generationsdauer der einzelnen Arten und sind selbst ein Ergebnis der Evolution : Sie bewirken eine ausreichende Anpassungsfähigkeit der Arten, ohne daß die Arten in ihrem Genbestand instabil werden). Da die Gene die Eigenschaften eines Individuums ausmachen, unterscheiden sich die überzahlreichen Nachkommen in ihren Eigenschaften. Die Lebewesen stehen untereinander in einem ständigen Wettbewerb um günstige Lebensbedingungen. Es herrscht ein permanenter Kampf ums Dasein (struggle for life). Es überleben nur die an die Umwelt bestangepaßten Nachkommen (survival of the fittest). Nur diese Individuen gelangen zur Fortpflanzung. Die anhaltende natürliche Auslese (natural selection) bewirkt, daß die weniger tauglichen Individuen einer Population ([24] S.337) zurückgedrängt und schließlich

ausgemerzt werden. Der Zwang zur bestmöglichen Anpassung an die Umwelt (Selektionsdruck) führt zu immer optimaleren Eigenschaften der Lebewesen (transformierende Selektion). Ungeachtet, ob eine solche Anpassung - bei gleichgebliebener Umwelt - bereits eingetreten ist, entstehen mit konstanter Rate neue Mutationen. Ist die Anpassung weit fortgeschritten, so nimmt die Wahrscheinlichkeit für „positive" Mutationen ab. In diesem Fall sorgt die Selektion dafür, daß die genetische Zusammensetzung einer Population konstant bleibt, indem auftretende „negative" Mutationen - wenn diese nicht schon letal verlaufen sind - wieder beseitigt werden (stabilisierende Selektion). Entstehen jedoch positive Mutationen, oder verändert sich die Umwelt erneut, so tritt wieder transformierende Selektion ein. Mutation und Selektion stellen die eigentlichen „Motoren" der Evolution dar. Für die Evolution können allerdings noch andere Faktoren eine Rolle spielen, z.B. Isolation, Zufallswirkung ([15] Seite 317 ff.), geschlechtliche oder ungeschlechtliche Vermehrung.

In 7.4. wurden selbstreproduzierende Programme mit Viren verglichen. Obwohl Viren keine Lebewesen sind, läßt sich an ihnen Evolution beobachten. Die Gründe dafür sind:

- Viren sind zu Mutationen fähig
- Viren befinden sich ebenfalls in einem Kampf ums Dasein und sind daher der Selektion unterworfen.

Es liegt der Schluß nahe, daß Evolution auch bei selbstreproduzierenden Programmen möglich ist, falls diese Mutation und Selektion gleichzeitig ausgesetzt sind. In 8.3. haben wir mit MOD2 ein Modell für konkurrierendes Verhalten (= Kampf ums Dasein) von Programmen entwikkelt. Die Programmtypen in MOD2 lassen sich durch ihre Reproduktionszeit und ihre Stellung in der Vorrangmatrix beschreiben. In MOD2 werden sich diejenigen Programme behaupten, die in der Vorrangmatrix eine günstige Stellung einnehmen, also relativ leicht Speicherplatz

für ihre Kopien finden, bzw. die eine kurze Reproduktionszeit aufweisen. Es herrscht in MOD2 also Selektionsdruck in Richtung

- kurze Reproduktionszeit
- günstige Stellung in der Vorrangmatrix.

Würde es einem Programmtyp gelingen, eine kürzere Reproduktionszeit zu erlangen, oder eine bessere Stellung in der Vorrangmatrix einzunehmen, so würden seine einzelnen Exemplare den in MOD2 vorhandenen Konkurrenzkampf besser bestehen können. Als Ursache für solche Veränderungen kommt Mutation in Frage. In 9.2. wird MOD2 dahingehend erweitert, daß Programme die Möglichkeit erhalten zu mutieren. Damit liegt dann ein Modell vor, in dem alle Voraussetzungen für das Eintreten von Evolution gegeben sind. Die SIMULA-Version dieses Modells stellt dann ein Rechnerprogramm zur Simulation von Evolution bei selbstreproduzierenden Programmen dar.

Bei dieser Gelegenheit sei erwähnt, daß Rechnerprogramme ganz allgemein ein adäquates Mittel zur Simulation von Evolutionsprozessen darstellen. Der Grund ist, daß Evolution (biologische, chemische, kosmische,...) in der Regel einen sehr langen Zeitraum benötigt, um merkliche Veränderungen hervorzurufen und daher Simulationsmodelle immer vor dem Problem stehen, diesen Zeitraum, in dem ja ständig etwas „geschieht", zu simulieren. Nur mit schnellen Rechenanlagen, die eine Vielzahl von Operationen in Sekundenbruchteilen durchführen können, ist man in der Lage, diesen Zeitraum auf ein erträgliches Maß zu komprimieren (vgl. [8]).

9.2.Ein Modell MOD3 für Evolution selbstreproduzierender Programme

Wir gehen von der Vorstellung aus, daß während der Reproduktion eines Programms π mit einer gewissen Wahr-

scheinlichkeit p_1 (Modellparameter) Fehler unterlaufen, so daß die Kopie $\bar{\pi}$ von π verschieden ist und in diesem Sinne eine Mutation des Programms π darstellt. I.a. werden die Fehler minimal und die Unterschiede von π und $\bar{\pi}$ gering sein. Da in MOD3 Programme durch ihre Reproduktionszeit und ihre Stellung in der Vorrangmatrix beschrieben werden, müssen sich Mutationen in einer Änderung dieser Werte nach außen bemerkbar machen, vorausgesetzt, $\bar{\pi}$ ist noch ein selbstreproduzierendes Programm. Führt eine Mutation nicht zu einem selbstreproduzierenden Programm, so liegt eine Letalmutation vor. Da Mutationen immer sprunghaft und ungerichtet verlaufen, ist das Auftreten einer Letalmutation jederzeit möglich. In MOD3 gibt die Wahrscheinlichkeit p_2 (Modellparameter) ein Maß für die Häufigkeit der letal ausgehenden Mutationen an. Jede nicht letal verlaufende Mutation bringt im Grunde ein erstes Exemplar eines neuen Programmtyps hervor. Entsprechend werden Mutanten in MOD3 registriert, ohne daß jedoch die „Abstammung" der Mutante verloren geht. Jedes Auftreten einer nicht letalen Mutation bewirkt also in MOD3 immer eine Vergrößerung der Vielfalt an vorhandenen Programmtypen. Da durch den Konkurrenzkampf der Programme untereinander ein Selektionsdruck in Richtung kürzere Reproduktionszeit (höhere Vermehrungsrate) bzw. günstige Stellung in der Vorrangmatrix besteht, werden diejenigen Mutanten gegenüber ihren Originalprogrammen im Vorteil sein, die bzgl. dieser Werte Verbesserungen aufweisen (Erhöhung der Fitneß). Solche Mutanten werden - unter gewissen Nebenbedingungen - in der Lage sein, die Programmtypen, denen die Ursprungsprogramme angehören, zu verdrängen (Selektion). Selbstverständlich stellen durch Mutation entstehende Programmtypen keine endgültigen Formen dar, sondern können selbst wieder Mutationen hervorbringen. Da die Reproduktion von Programmen wie eine „ungeschlechtliche" Vermehrung verläuft, kann jede Mutante als Ausgangspunkt einer sich potentiell aufzeigenden „Linie" auseinander hervorgehender Programme(Typen) (Klons siehe [24] Seite 313) verstanden werden.

9.2.1.Informelle Beschreibung von MOD3

(i) Programme: In MOD3 werden Programme durch ihre Reproduktionszeit und ihren Namen repräsentiert. Stellt ein Programm eine Mutation dar, so gibt der Name Aufschluß über die „Abstammung" des Programms.

(ii) Speicher: Wie in MOD2.

(iii) Zeitverhalten: Wie in MOD2. Allerdings ist ein Programm, das zum Zeitpunkt t reproduktionsfähig ist, also so viele Zeittakte aktiv war, wie seine Reproduktionszeit angibt, in der Lage, eine Mutation hervorzubringen. Die Wahrscheinlichkeit für eine Mutation beträgt p_1. Mit der Wahrscheinlichkeit p_2 verläuft die Mutation letal. Verläuft die Mutation nicht letal, so unterscheiden sich Mutante und Originalprogramm mit der Wahrscheinlichkeit p_3 in der Komponente DELY. Mit der Wahrscheinlichkeit $1-p_3$ liegt der Unterschied im Konfliktverhalten (Vorrangmatrix) gegenüber anderen Programmen. Das Auftreten einer nicht letalen Mutation bewirkt die Erhöhung der Anzahl M der momentanen Programmtypen in MOD3 um 1. Stellt die Kopie eines Programms eine Mutation dar, so wird mit der Mutante weiter verfahren, als handele es sich um eine korrekte Kopie.

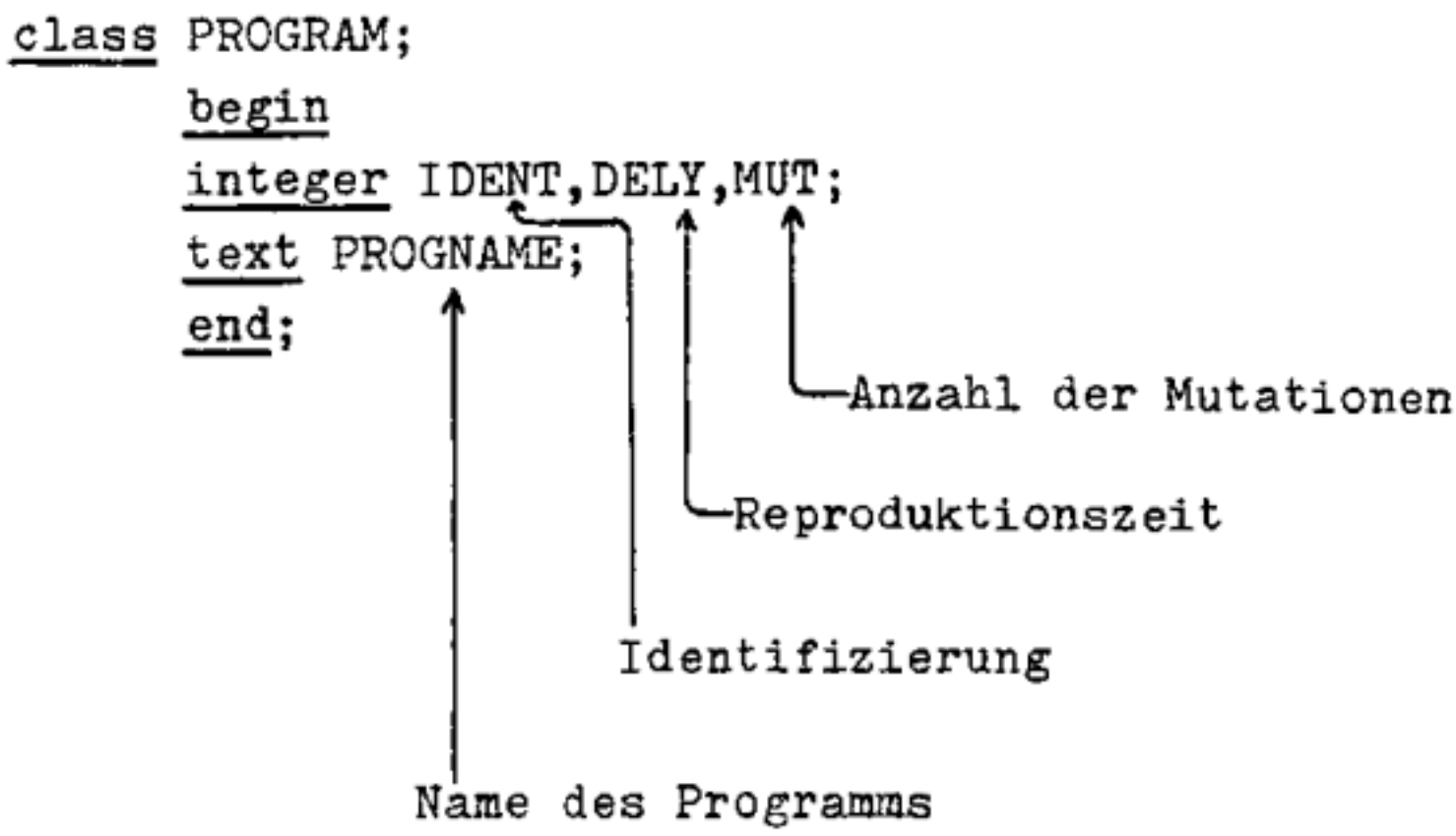
(iv) Räumliches Verhalten: Wie in MOD2. Mutanten und korrekte Kopien werden gleich behandelt.

(v) Verhalten der Programme untereinander: Wie in MOD2 wird das Verhalten der Programme untereinander durch eine Vorrangmatrix gesteuert. Beim Auftreten einer nicht letalen Mutation muß die Vorrangmatrix um eine Spalte und eine Zeile erweitert werden, um die Konfliktfälle zwischen Programmen des neuen Typs mit den Programmen der alten Typen zu regeln.

9.2.2.MOD3 als SIMULA-Programm

(i) Programme:

Ein Programm(-typ) wird durch die SIMULA-Struktur



dargestellt. Die Komponenten DELY und PROGNAME ergeben sich aus der Beschreibung in 9.2.1.(i). Die Komponente PROGNAME würde zur Identifizierung der einzelnen Programmtypen ausreichen. Trotzdem kann auf die Größe IDENT aus programmtechnischen Gründen (array-Zugriffe) nicht verzichtet werden. Die Komponente MUT gibt zu jedem Zeitpunkt die Anzahl der Mutanten an, die aus dem dargestellten Programm hervorgegangen sind.

(ii) Speicher:

Die Darstellung des Speichers erfolgt wie im SIMULA-Programm für MOD2. Auch der Mechanismus der Speichererweiterung sowie dessen Steuerung über die integer-Größen MORE und PERCENT werden übernommen. (Prozeduren: NEW_STORAGE,OVERFLOW)

(iii) Zeitverhalten:

Zunächst ist zu bemerken, daß während der Simulation die Anzahl M der vorhandenen Programmtypen i.a. nicht konstant bleibt. Somit sind alle Felder, die in MOD2 M Komponenten aufweisen, in MOD3 von variabler Länge. Entsprechendes gilt für die Vorrangmatrix. Es müssen also in MOD3 einige Felder dynamisch angelegt werden:

```
class PROG(P);integer P;
begin
ref (PROGRAM) array VECTOR[1:P];
end;

ref (PROG) PROGPOINTER;
```

anstelle von
ref (PROGRAM)
array P[1:M]

```
class ST(P);integer P;
begin
integer array S[1:P];
end;

ref (ST) STPOINTER;
```

anstelle von
integer array
ST[1:M]

```
class CONFLICT(P);integer P;
begin
integer array MAT[1:P,1:P];
end;

ref (CONFLICT) CONPOINTER;
```

anstelle von
integer array
CONFLICT[1:M,1:M]

Startsituation: Die Beschreibung der Startsituation überträgt sich aus 8.3.2.(iii) unter Berücksichtigung der gerade beschriebenen organisatorischen Änderungen der Datenstrukturen. Es bleibt nur zu vermerken, wie die beiden zusätzlichen Komponenten MUT und PROGNAME initialisiert werden. Sei $\{\pi_1, \ldots, \pi_M\}$ die Menge der anfangs vorkommenden Programmtypen, dann wird bei der Initialisierung

- die Komponente MUT für jedes π_j auf Ø gesetzt
- der Textkomponenten PROGNAME der Wert „P1" für den Programmtyp π_1, „P2" für den Programmtyp π_2, u.s.w., zugewiesen.

Simulation: Die Beschreibung der Simulation kann im wesentlichen aus 8.3.2.(iii) übernommen werden.

Da MOD3 eine echte Erweiterung von MOD2 darstellt, bedarf es jedoch einiger Ergänzungen, die die Erzeugung und Behandlung von Mutationen betreffen. Diese Ergänzungen äußern sich in einem Satz von Prozeduren, die sämtlich von der Prozedur MATCH aufgerufen werden. Bevor wir die so erweiterte Prozedur MATCH angeben können, müssen diese zusätzlichen Prozeduren erläutert werden.

Die modellwirksamen Eigenschaften eines Programms sind die Komponente DELY und die Stellung des Programms in der Vorrangmatrix. Nur Mutationen in einer dieser beiden Eigenschaften haben in MOD3 einen Selektionswert. In der SIMULA-Version von MOD3 werden Mutationen von Programmen mittels der Funktionsprozedur

ref (PROGRAM) procedure MUTANT(X); ref (PROGRAM) X;

erzeugt. MUTANT liefert als Ergebnis einen Zeiger auf ein Objekt vom Typ PROGRAM. Dieses Objekt unterscheidet sich geringfügig in einer der oben genannten modellwirksamen Eigenschaften von dem in Form des Zeigers X an die Prozedur übergebenen Originalprogramm und stellt in diesem Sinne eine Mutante dar. Das Auftreten einer Mutation bewirkt immer das Erscheinen eines neuen Programmtyps und macht die Erhöhung der Variablen M, die zu jedem Zeitpunkt die Anzahl der im Modell vorhandenen Programmtypen angibt, um 1 erforderlich. Diese Erhöhung wird bereits vor dem Aufruf der Prozedur MUTANT vorgenommen.

Funktionsweise von MUTANT:

I. Zunächst legt MUTANT ein neues Objekt vom Typ PROGRAM an und initialisiert die nicht unmittelbar modellrelevanten Komponenten. Dieses über den Zeiger HELP adressierte Objekt wird als Mutante des Programms X aufgebaut.

```
ref (PROGRAM) HELP;
HELP:-new PROGRAM;
HELP.MUT:=Ø;
HELP.IDENT:=M;
HELP.PROGNAME:-CREATE_NAME;
comment Beschreibung von CREATE_NAME s.u.;
X.MUT:=X.MUT+1;
comment *** Die Komponente MUT des Original-
            programms X wird erhöht, um die
            Mutation dieses Programms zu re-
            gistrieren. ***;
```

II. Da die Mutante einen neuen Programmtyp darstellt, muß die Vorrangmatrix um eine zusätzliche Zeile und Spalte erweitert werden. Zum Zeitpunkt des Aufrufs von MUTANT steht noch nicht fest, ob sich die Mutante etablieren kann oder im Anschluß an ihre Generierung wieder eleminiert wird (vgl. unten Funktionsweise der Prozedur MATCH). Daher führt MUTANT die Erweiterung der Vorrangmatrix noch nicht aus, sondern erzeugt als Seiteneffekt nur die Zeile und die Spalte, um die die Vorrangmatrix bei Etablierung erweitert werden muß. Die Zeile und die Spalte werden in Form des Datentyps

```
class FIELD(P); integer P;
   begin
   integer array V[1:2,1:P];
   comment *** V[1,...]  entspricht der Spalte
               V[2,...]  entspricht der Zeile ***;
   end;
```

zusammengefaßt. MUTANT erzeugt ein Objekt vom Typ FIELD und weist es der globalen Variablen CHANGE_CONFLICT zu:

```
CHANGE_CONFLICT:-new FIELD(M);
```

Die Komponenten der X.IDENT-ten Zeile bzw. Spalte der Vorrangmatrix regeln die Konflikte zwischen Exemplaren des Programms X und den

Exemplaren der anderen Programmtypen. Bzgl. der Konflikte weist die Mutante ein dem Programm X ähnliches Verhalten auf. Daher wird die X.IDENT-te Spalte in die erste Zeile von CHANGE CONFLICT.V und die X.IDENT-te Zeile in die zweite Zeile von CHANGE_CONFLICT.V kopiert. Siehe Abb. 9.2.2.A.

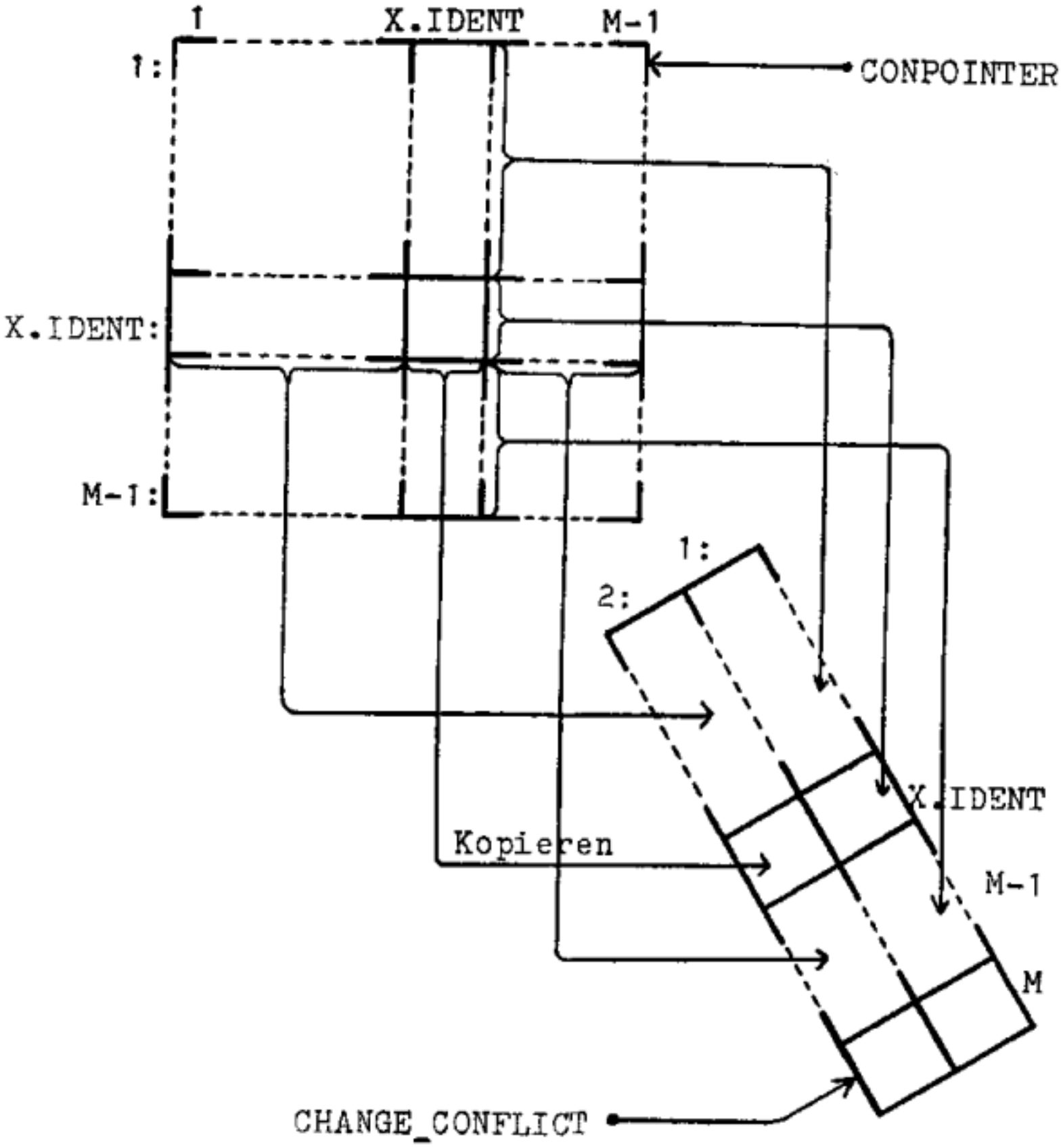


Abb. 9.2.2.A

CHANGE_CONFLICT.V(1,M),
CHANGE_CONFLICT.V(1,X.IDENT) und
CHANGE_CONFLICT.V(2,X.IDENT) stellen die Komponenten dar, die in der erweiterten Vorrangmatrix die innerartlichen Konflikte zwischen Exemplaren der Mutante HELP bzw. zwischen Exemplaren der Mutante und des Originalprogramms X regeln sollen und werden zu diesem Zweck neu generiert. Dies geschieht mit Hilfe der Zufallsfunktion RANDINT. Wegen der engen Verwandschaft zwischen Mutante und Originalprogramm weichen die Werte dieser Komponenten um höchstens 100% von der Komponente in der alten Vorrangmatrix ab, die die innerartlichen Konflikte zwischen Exemplaren des Programms X regelt. Bis auf diese 3 notwendigerweise neuen Werte weist die Mutante also bisher das gleiche Konfliktverhalten auf wie das Originalprogramm.

III. In welcher der beiden modellrelevanten Eigenschaften sich das Programm X von seiner Mutante unterscheidet, wird bestimmt mittels der Größe PROB_DELY (PROB_DELY stellt den Modellparameter p_3 auf Programmebene dar). Mit der Wahrscheinlichkeit PROB_DELY*10^{-3} mutiert (über Zufallsfunktion RANDINT) die DELY-Komponente des Programms X. Mit der Wahrscheinlichkeit 1-PROB_DELY*10^{-3} erhält die Mutante ein in Bezug auf einen der anderen in MOD3 vorhandenen Programmtypen (also nicht X) verändertes Konfliktverhalten als das Originalprogramm X.

- Unterscheiden sich die Mutante und das Originalprogramm in ihren DELY-Komponenten, so liegt der Unterschied bei höchstens 100%:

  ```
  HELP.DELY:=RANDINT(1,2*X.DELY,U_DELY2)
  while HELP.DELY=X.DELY do
  HELP.DELY:=RANDINT(1,2*X.DELY,U_DELY2)
  ```

- Erhält die Mutante ein anderes Konfliktverhalten als das Originalprogramm X, so wird

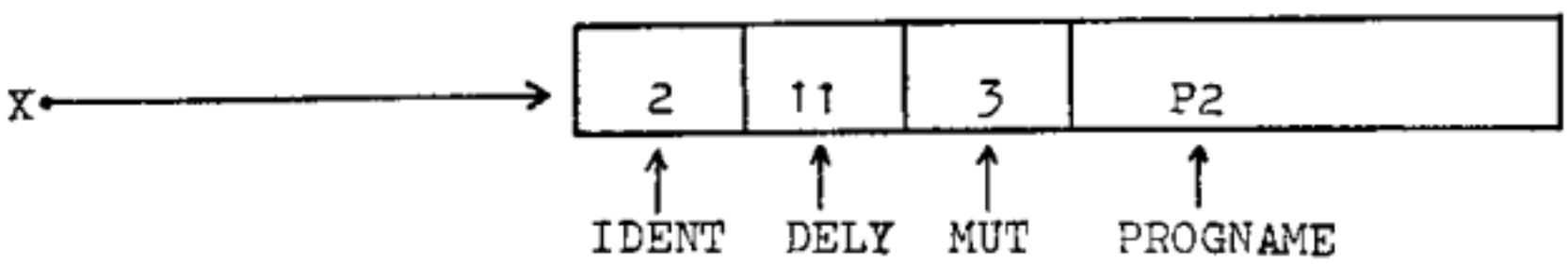
genau eine Komponente des Feldes CHANGE_CONFLICT um höchstens 100% geändert. Es muß sich dabei um eine Komponente handeln, die aus der alten Vorrangmatrix übernommen worden ist. Die Komponente darf also nicht die Indizes [1,M], [2,M], [1,X.IDENT] und [2,X.IDENT] aufweisen und wird mittels der Zufallsfunktion RANDINT ausgewählt:

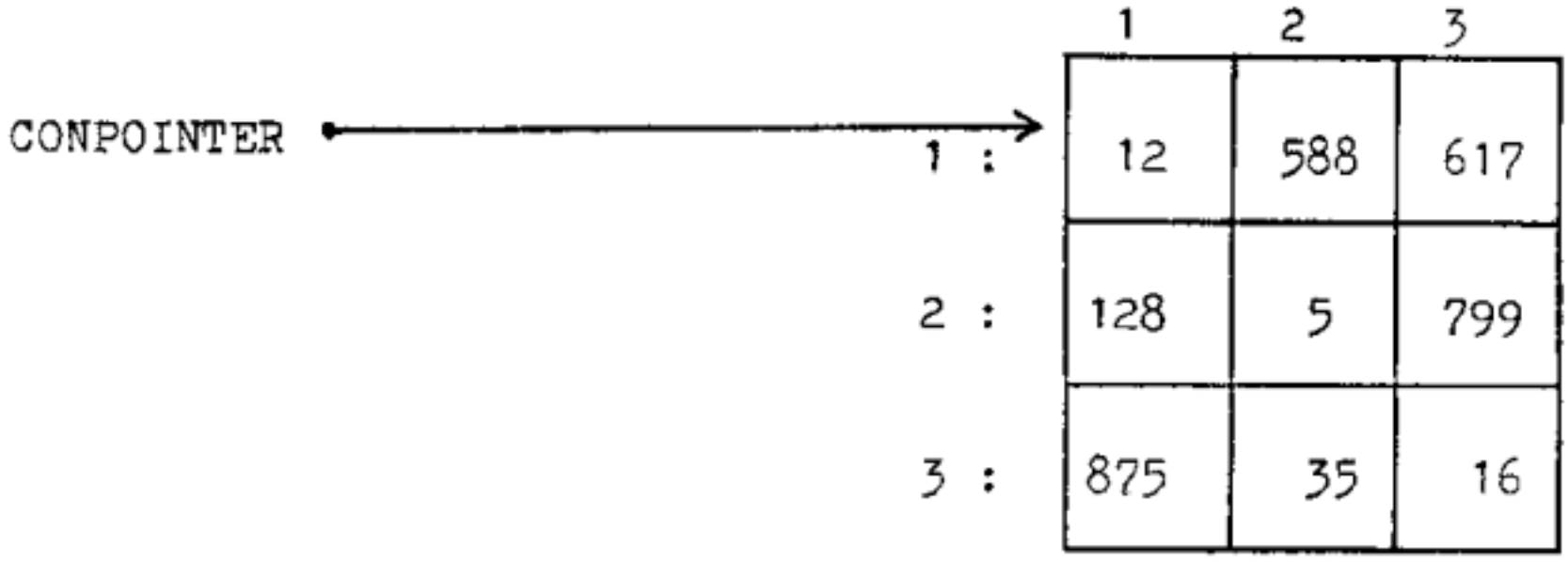
```
integer I,J,K,U_CONFLICT;
    :
J:=X.IDENT;
while J=X.IDENT do
begin
J:=RANDINT(1,2*(M-1),U_CONFLICT);
if J ≤ M-1
then I:=1
else
begin
I:=2;
J:=J-(M-1)
end
end;
K:=RANDINT(1,2*CHANGE_CONFLICT.V[I,J],U_CONFLICT);
while K=CHANGE_CONFLICT.V[I,J] do
K:=RANDINT(1,2*CHANGE_CONFLICT.V[I,J],U_CONFLICT);
CHANGE_CONFLICT.V[I,J]:=K;
```

IV. Nach Abschluß von III. liegen die fertige Mutante sowie die Änderungszeile und -spalte der Vorrangmatrix in Form von HELP bzw. CHANGE_CONFLICT vor. Da MUTANT eine Funktionsprozedur ist, wird MUTANT mit der Anweisung MUTANT:=HELP beendet.

Beispiel:

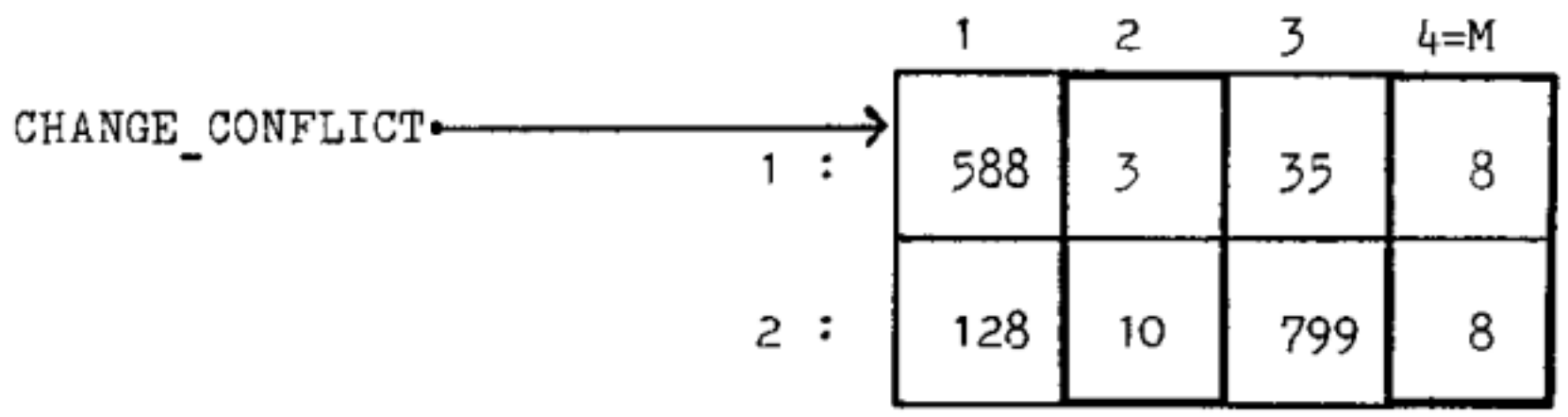
M = 3



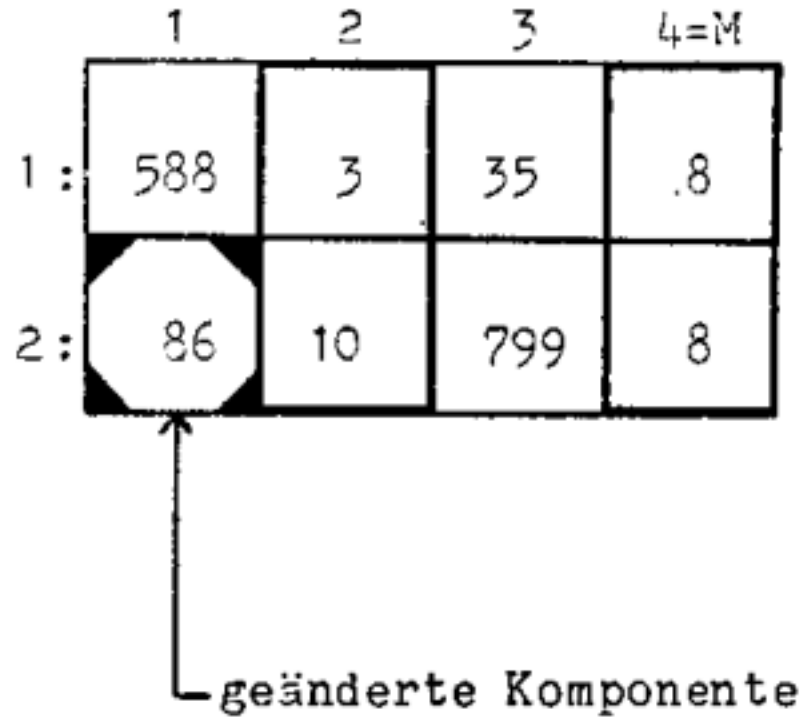


Eine Mutation tritt auf:

M := M+1



Mutation DELY-Komponente	Mutation Konfliktverhalten
MUTANT → 4 | 19 | ø | P2.1	MUTANT → 4 | 11 | ø | P2.1
CHANGE_CONFLICT wie oben	CHANGE_CONFLICT



In der Prozedur MUTANT wird die Komponente PROGNAME der generierten Mutante durch Aufruf der Funktions-prozedur

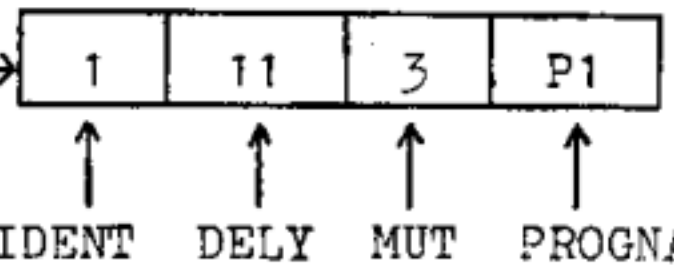
```
text procedure CREATE_NAME(X); ref (PROGRAM) X;
```

gesetzt.

Funktionsweise von CREATE_NAME:

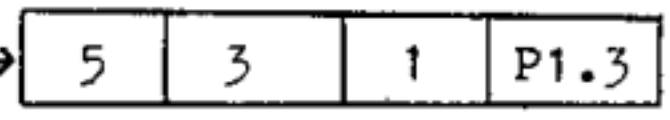
CREATE_NAME besitzt als formalen Eingabeparameter X einen Zeiger auf ein Objekt vom Typ PROGRAM und liefert als Ergebnis einen Text. Dieser Text stellt den Namen einer Mutante des Programms X dar und wird aus den Komponenten X.MUT und X.PROGNAME erzeugt, indem an den Text X.PROGNAME das Zeichen „." gefolgt von der Zahl X.MUT (als Text interpretiert) gehängt wird.

Beispiel: a)



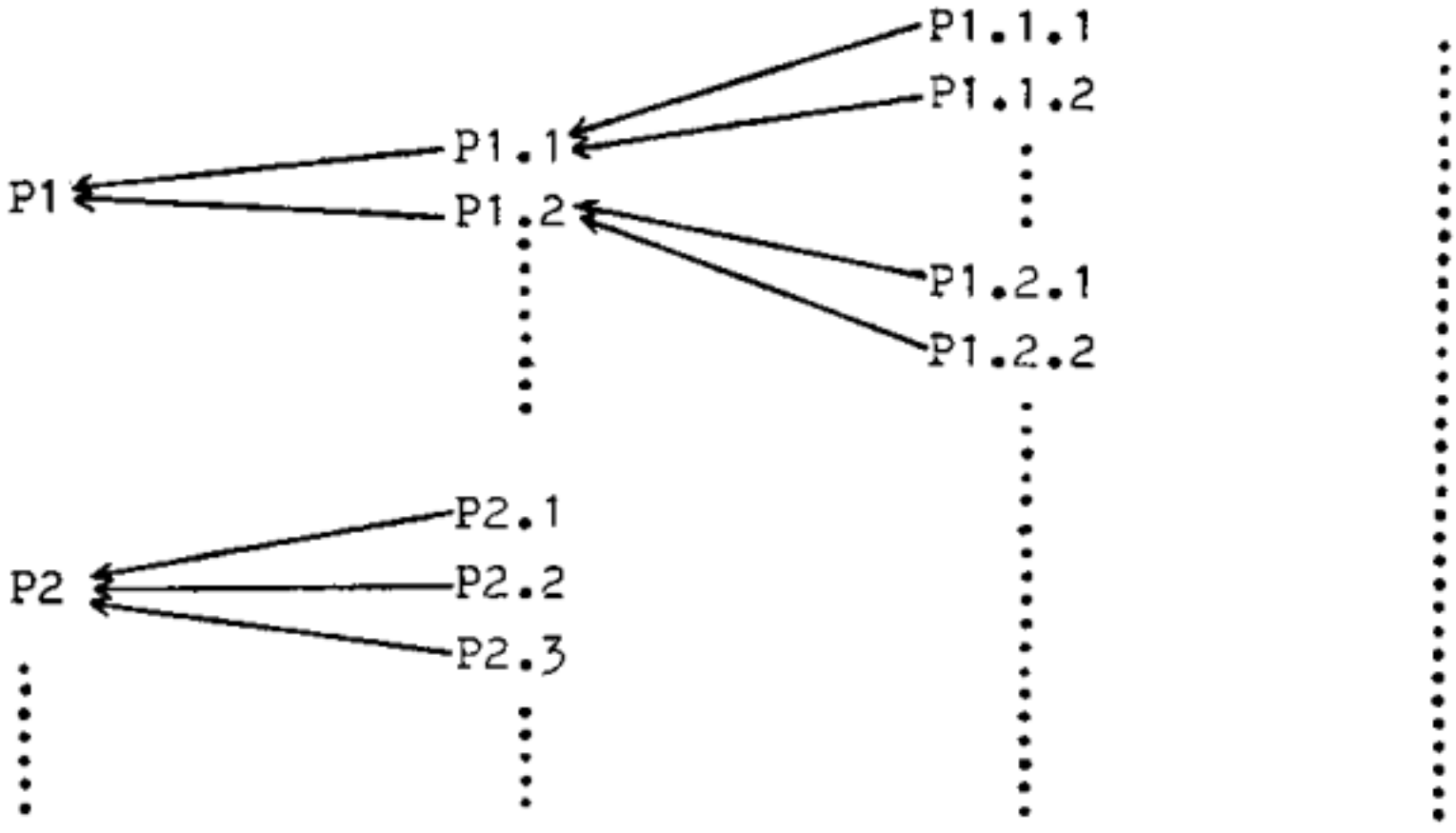
CREATE_NAME = P1.3

b)



CREATE_NAME = P1.3.1

Da beim Auftreten einer Mutation eines Programms dessen Komponente MUT um 1 erhöht wird, liefert dieser Mechanismus für aufeinanderfolgende Mutationen desselben Programms verschiedene Namen. Da der aktuelle Parameter für X selbst eine Mutante sein kann (s. Beispiel b)), läßt sich der Komponente PROGNAME eines Programms stets dessen „stammesgeschichtliche Entwicklung" zurückverfolgen (siehe Abb. 9.2.2.B). Dies wäre an Hand der integer-Komponenten IDENT, die in MOD2 zur Identifizierung der Programme ausreichte, nicht möglich.



Name der Originalprogramme	Namen der Mutanten erster Stufe	Namen der Mutanten zweiter Stufe	....

Abb. 9.2.2.B

Wie schon wiederholt erwähnt, stellt das Auftreten einer Mutation i.a. eine Vergrößerung der Anzahl der aktuellen Parametertypen dar. Das macht sich schon in der (zunächst vorläufigen) Erhöhung der Variablen M bemerkbar. Kann sich die Mutante etablieren (s.u. Beschreibung der Prozedur MATCH), so müssen die dynamischen arrays erweitert werden, die die Mutante speichern, registrieren bzw. verwalten. Diese Erweiterungen werden durch Aufrufe der Prozeduren

```
procedure NEW_PROG(P); ref (PROGRAM) P;
procedure NEW_ST(T); ref (PROGRAM) T;    und
procedure NEW_CONFLICT(A); integer array A;
```

gewährleistet.

Funktionsweise von NEW_PROG:

Der Zeiger PROGPOINTER verweist auf dasjenige Feld,

mit dessen Hilfe die eigentliche Abspeicherung der Programme (Typen) erfolgt. Beim Auftreten einer Mutation - die Mutante wird in Form des Zeigers P als Parameter an NEW_PROG übergeben - muß dieses Feld um eine Komponente erweitert werden. Realisiert wird dieses durch Generierung eines neuen Feldes (Objekt vom Typ PROG), einen einfachen Kopierprozeß und anschließendes Umsetzen des Zeigers PROGPOINTER. Siehe Abb. 9.2.2.C.

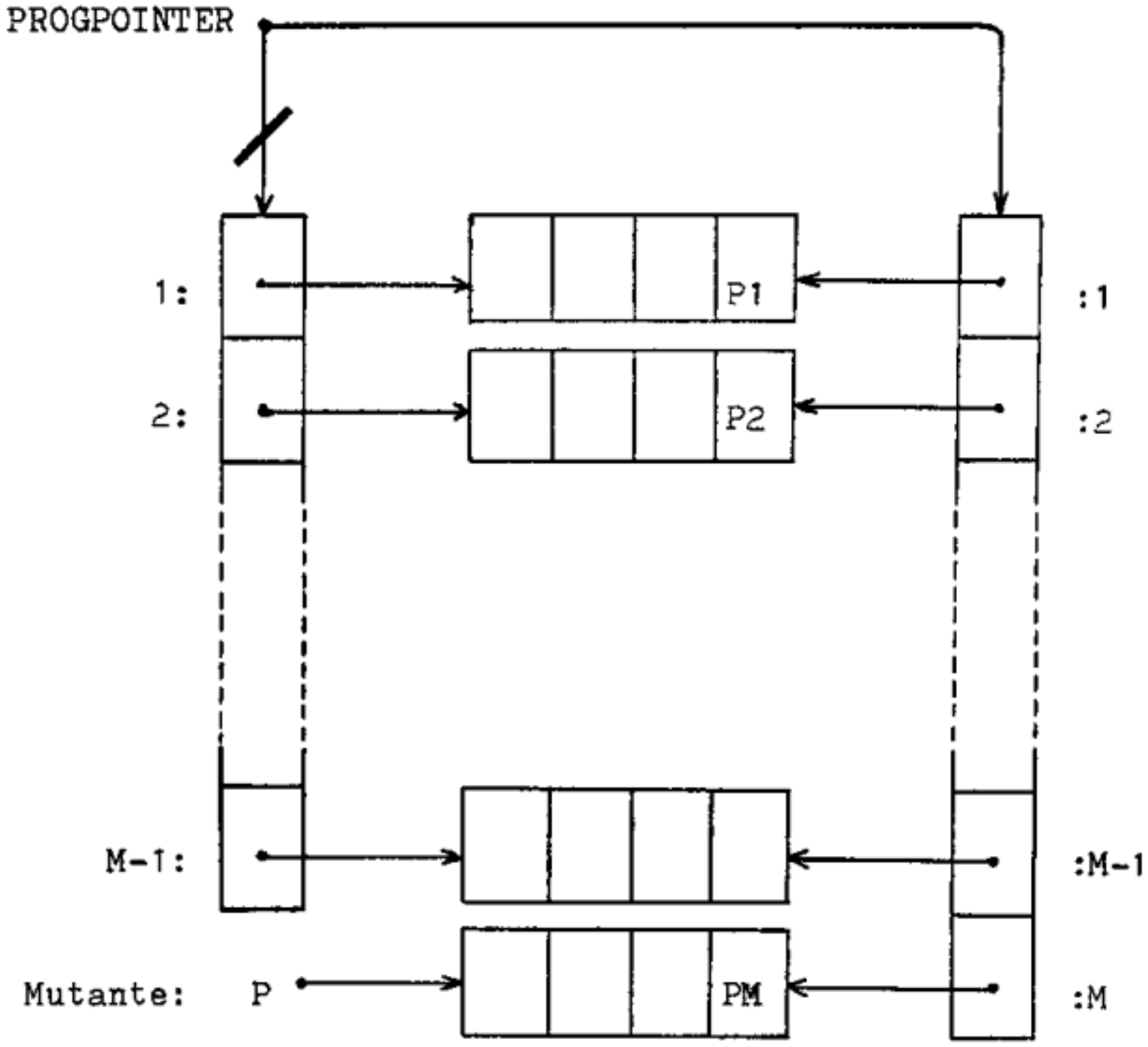


Abb. 9.2.2.C

Funktionsweise von NEW_ST:

Das Feld, auf das der Zeiger STPOINTER verweist, enthält zu jedem Zeitpunkt der Simulation in der i-ten Komponente die momentane Anzahl der Exemplare des i-ten Programmtyps. Das Feld wird beim Auftreten einer Mutation um eine Komponente zur Registrierung der Exemplare der Mutante erweitert. Die zusätzliche Komponente wird mit 1 initialisiert. Ansonsten analog zu NEW_PROG. Siehe Abb. 9.2.2.D.

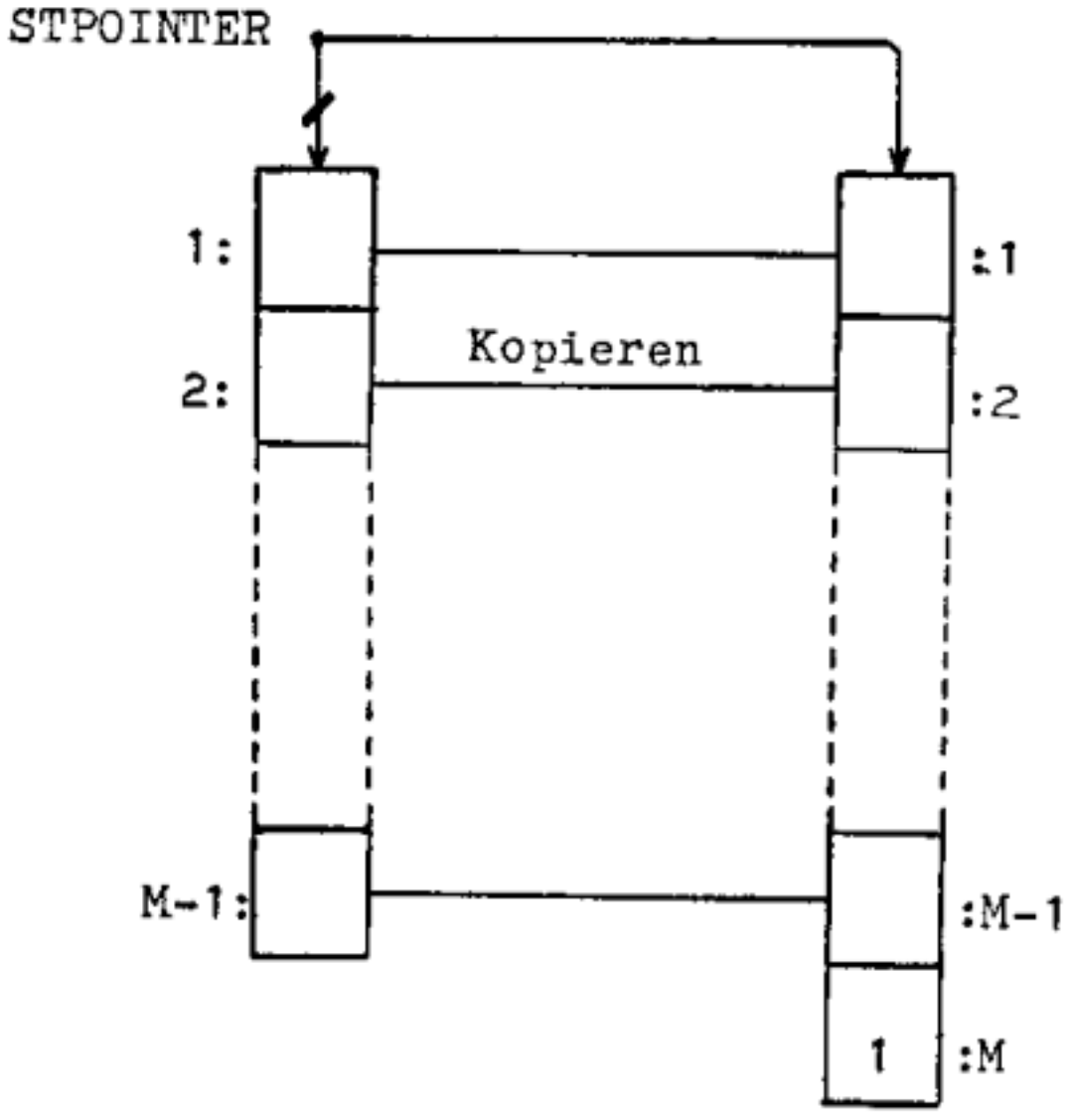


Abb. 9.2.2.D

Funktionsweise von NEW_CONFLICT:

Der Zeiger CONPOINTER verweist auf das Feld, das die Vorrangmatrix speichert. Beim Auftreten einer Mutation muß dieses Feld um eine Spalte und eine Zeile erweitert werden. Diese Zeile und Spalte entsprechen dem Konfliktverhalten der Mutante. Zeile und

Spalte werden in Form des formalen Parameters integer array A an NEW_CONFLICT übergeben. Der Aufruf von NEW_CONFLICT erfolgt mit dem durch CHANGE_CONFLICT adressierten Feld als aktueller Parameter. Dieses Feld wird vor Aufruf von NEW_CONFLICT von der Prozedur MUTANT generiert (s.o. Funktionsweise der Prozedur MUTANT). Im übrigen erfolgt der Ablauf von NEW_CONFLICT, wie Abb. 9.2.2.E zeigt, analog zu NEW_PROG und NEW_ST.
Abb. 9.2.2.E greift das Beispiel aus der Beschreibung der Prozedur MUTANT auf.

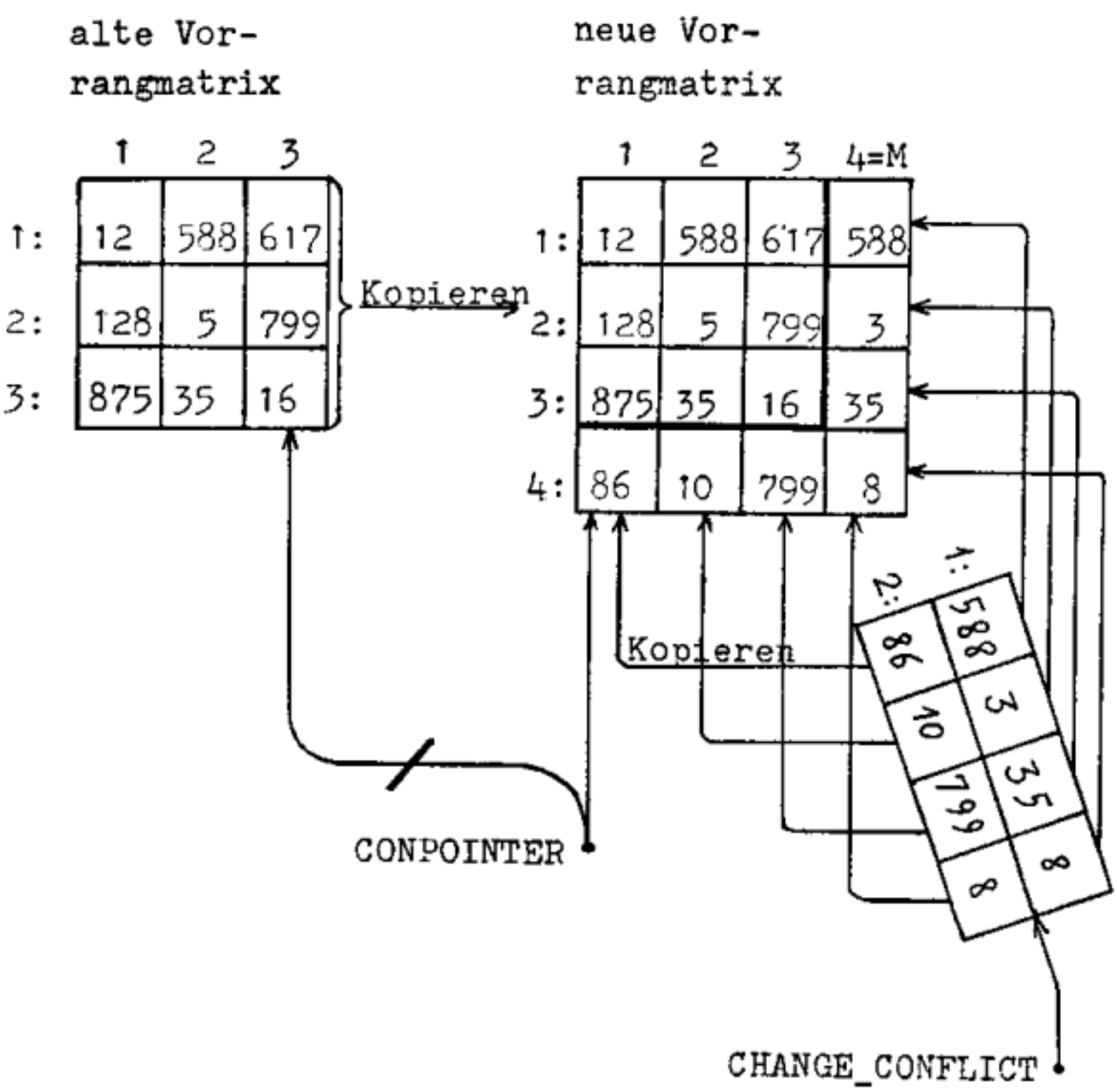


Abb. 9.2.2.E

Nach Erläuterung der der Erzeugung und Behandlung von Mutationen dienenden Prozeduren sind wir in der Lage, die Prozedur MATCH anzugeben. MATCH stellt, wie schon im SIMULA-Programm zu MOD2 (s. 8.3.2.), das Herzstück der Simulation dar. Das übergeordnete Simulationsschema, wie es in 8.3.2.(iii) Seite 182 angegeben ist, kann vollständig übernommen werden.

```
procedure MATCH(I); integer I;
begin
   :
   .
boolean IS_COPY;
IS_COPY:=false;
[Erhöhe TIMECOUNT-Komponente der I-ten Speicher-
zelle um 1.];
if [TIMECOUNT-Komponente der I-ten Speicherzelle
    gleich DELY-Komponente des in der I-ten Spei-
    cherzelle befindlichen Programms]
then
begin
  comment *** Reproduktion des in der I-ten Zelle
              befindlichen Programms *** ;
   [Setze TIMECOUNT-Komponente der I-ten Speicher-
   zelle auf Ø] ;
   [Wähle eine zufällige Zelle des Speichers aus.
   Sei die Wahl auf die W-te Speicherzelle gefal-
   len.] ;
   comment *** Siehe dazu oben 8.3.2.(iv) *** ;
   [Treffe Entscheidung, ob das in der I-ten Spei-
   cherzelle befindliche Programm mutiert.] ;
   comment *** Die Mutationswahrscheinlichkeit be-
               trägt PROB_MUT*10^-8 = p1 (Modell-
               parameter). Die integer-Größe PROB_
               MUT stellt den Modellparameter p1
               auf Programmebene dar. *** ;
```

```
if [Programm in der I-ten Speicherzelle mutiert.]
then
begin
   [Treffe Entscheidung, ob die Mutation letal verläuft.] ;
   comment *** Die Wahrscheinlichkeit für den letalen Ver-
               lauf einer Mutation beträgt PROB_LETAL*10^-6
               = p2 (Modellparameter). Die integer-Größe
               PROB_LETAL stellt den Modellparameter p2
               auf Programmebene dar. *** ;
   if [Letaler Verlauf der Mutation]
   then [Erhöhe die Komponente MUT des in der I-ten Zelle
         gespeicherten Programms um 1.                    ] ;
        comment *** Dies geschieht zur Registrierung der
                    Mutation. Im Falle einer nicht letalen
                    Mutation wird die Erhöhung der Kompo-
                    nente MUT von der Prozedur MUTANT vor-
                    genommen. ***
else
begin
   comment *** Existenzfähige Mutation *** ;
   M:=M+1;
   comment *** Rettung der alten Vorrangmatrix und des
               alten Programmspeichers *** ;
   OLD_CONPOINTER:-CONPOINTER;
   OLD_PROGPOINTER:-PROGPOINTER;
   [Erzeuge Mutante des in der I-ten Speicherzelle be-
   findlichen Programms (Aufruf MUTANT). Trage den von
   MUTANT erzeugten neuen Programmtyp in den Programm-
   speicher ein (Aufruf NEW_PROG). Erweitere die Vor-
   rangmatrix (Aufruf NEW_CONFLICT).                  ] ;
comment *** CONPOINTER und PROGPOINTER verweisen auf
            die neue Vorrangmatrix bzw. den neuen
            Programmspeicher *** ;
if [W-te Speicherzelle leer]
then
begin
   comment *** Ungehindertes Ablegen der erzeugten
               Mutante im Speicher *** ;
```

```
      [Schreibe die Mutante in die W-te Speicherzelle] ;
      NEW_ST;
      IS_COPY:=true
    end
    else
    begin
      comment *** W-te Speicherzelle ist bereits be-
                  setzt. *** ;
      [Treffe Entscheidung mittels der neuen Vorrangma-
      trix, ob die Mutante das in der W-ten Zelle be-
      findliche Programm überschreiben darf.            ];
      comment *** Siehe unten (v) *** ;
      if [Überschreiben nicht möglich]
      then
      begin
        comment *** Vernichtung der Mutante, Wiederher-
                    stellung der alten Tabellen *** ;
        M:=M-1;
        CONPOINTER:-OLD_CONPOINTER;
        PROGPOINTER:-OLD_PROGPOINTER
      end
      else
      begin
        comment *** Die Mutante kann das in der W-ten
                    Zelle befindliche Programm überschrei-
                    ben. *** ;
        [Schreibe Mutante in die W-te Speicherzelle] ;
        NEW_ST;
        IS_COPY:=true
      end
    end
  end
  else
  begin
    comment *** Das Programm in der I-ten Speicherzelle er-
                zeugt eine korrekte Kopie ohne Mutation *** ;
    if [W-te Speicherzelle leer]
    then
```

```
begin
  comment *** Ungehindertes Ablegen der Kopie *** ;
  [Schreibe Kopie in die W-te Speicherzelle] ;
  IS_COPY:=true
end
else
begin
  comment *** W-te Speicherzelle bereits besetzt *** ;
  [Treffe Entscheidung mittels der Vorrangmatrix, ob
  die Kopie des in der I-ten Zelle befindlichen
  Programms das Programm in der W-ten Zelle über-
  schreiben darf.                                    ];
  comment *** Siehe unten (v) *** ;
  if [Überschreiben nicht möglich]
  then comment *** Es geschieht nichts *** ;
  else
  begin
    [Schreibe Kopie in die W-te Speicherzelle] ;
    IS_COPY:=true
  end
 end
end;
comment *** Falls das Programm aus Speicherzelle I
            seine Mutante/Kopie im Speicher ablegen
            konnte, muß noch in Abhängigkeit von
            der Durchlaufrichtung die Komponente
            TIMECOUNT der W-ten Speicherzelle ge-
            setzt werden * **;
```

⊛ (s.S.184)

```
 end
end *** MATCH *** ;
```

(iv) Räumliches Verhalten:

Wie im SIMULA-Programm für MOD2, da die Möglichkeit der Mutation hier keine Änderung bedingt.

(v) Verhalten der Programme untereinander:

Wie im SIMULA-Programm für MOD2. Jedes Element v_{ij} der momentanen Vorrangmatrix wird jedoch als „v_{ij} Tausendstel" interpretiert, was eine Verfeinerung gegenüber dem Programm für MOD2 darstellt. Die Elemente der jeweiligen Vorrangmatrix sind somit Elemente der Menge [↑000].

Anhang C.3. zeigt die Realisierung von MOD3 als ausführlich kommentiertes SIMULA-Programm. Einen Überblick über die in diesem Programm benutzten Datenstrukturen gibt Abb. 9.2.2.F.

Eingabeparameter des SIMULA-Programms für MOD3:

- Die Anfangslänge des Speichers	N
- Die anfängliche Anzahl der unterschiedlichen Programmtypen	M
- Die M anfänglichen Programmtypen, in Form der Größe DELY und der Anfangshäufigkeit	DELY,STPOINTER.S[...]
- Die M×M Elemente der anfänglichen Vorrangmatrix. Jedes Element ist aus [↑000].	CONPOINTER.MAT
- Die Zahlen, die die Wahrscheinlichkeiten für Mutationen, Letalmutationen und Mutationsart festlegen	PROB_MUT $\in [\uparrow 0^8]$ PROB_LETAL $\in [\uparrow 0^6]$ PROB_DELY $\in [\uparrow 0^3]$
- Die Anzahl der vorgesehenen Speicherdurchläufe	TIME
- Die Speicherparameter	MORE,PERCENT

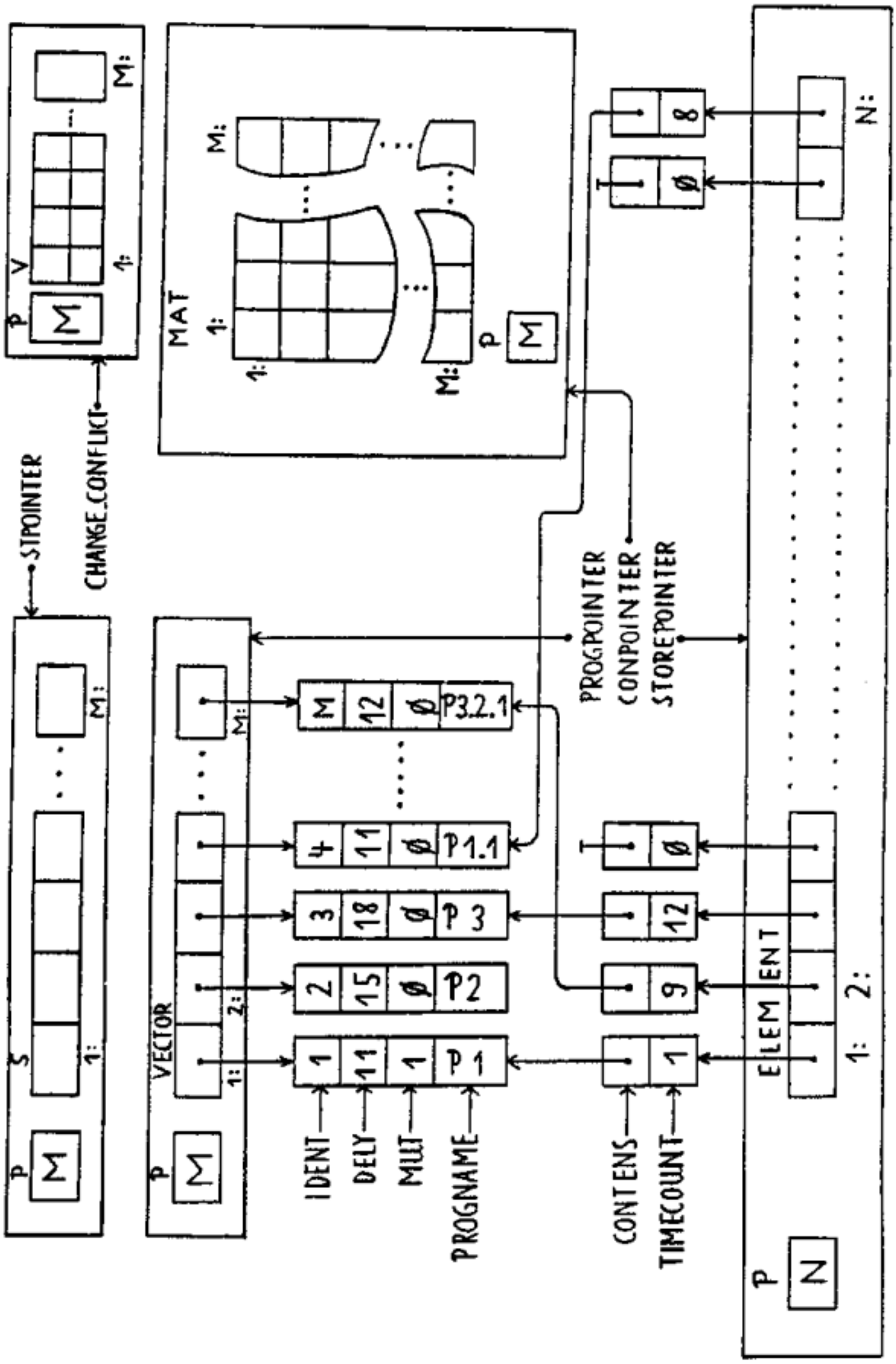

GLOBALE DATENSTUKTEN IN MOD3 (BEISPIEL)
P
M
V
1:
M:
MAT
1:
M:
P
M
S
1:
M:
SIPOINTER
CHANGE_CONFLICT
VECTOR
1:
2:
M:
P
M
IDENT
DELY
MULT
PROGNAME
1
11
1
P1
2
15
Ø
P2
3
18
Ø
P3
4
11
Ø
P1.1
M
12
Ø
P3.2.1
PROGPOINTER
CONPOINTER
STOREPOINTER
CONTENS
TIMECOUNT
1
1
9
12
Ø
Ø
ELEMENT
1:
2:
N:
8
Ø
P
N

Abb. 9.2.2.F

9.2.3.Einige Aspekte des SIMULA-Programms für MOD3

I. Das SIMULA-Programm für MOD3 gestattet Simulation sowohl mit endlichem als auch mit unendlichem Speicher (abhängig von PERCENT).

II. Zur Unterstützung der Ausgabe wurden die Prozeduren DUMP, CONTROL und AVERAGE in das Programm eingefügt. Daher enthält das Programm die drei modellunabhängigen Parameter

    WHEN_DUM
    WHEN_CON und
    WHEN_AVE (vgl. 8.2.4. I. und II.)

III. Wird PROB_MUT auf Ø gesetzt, also das Auftreten von Mutationen unterdrückt, so ergibt sich die SIMULA-Version von MOD2.

IV. Mit Hilfe des Programms für MOD3 lassen sich gewisse Fragestellungen experimentell untersuchen. Aus III. folgt, daß sich alle im Zusammenhang mit dem SIMULA-Programm für MOD2 stellenden Fragen auch mit dem Programm für MOD3 bearbeiten lassen. Das Programm für MOD3 ermöglicht darüber hinaus, Fragen im Hinblick auf Evolution zu bearbeiten. Z.B.:

- Welche Mutationshäufigkeit ist im vorhandenen Modell optimal?
- Welche Mutationsrate darf keinesfalls überschritten werden, um die auftretenden Mutanten nicht instabil werden zu lassen?
- Durch entsprechendes Setzen von PROB_DELY sind differenzierte Betrachtungen im Hinblick auf die Selektionswirksamkeit der Reproduktionszeit (DELY-Komponente) und der Stellung in der Vorrangmatrix möglich.
- Wie können sich Mutanten einerseits gegenüber Exemplaren ihres eigenen Ursprungstyps und andererseits gegenüber Exemplaren anderer Programm-

typen durchsetzen? (Die Beantwortung wird unterstützt durch die Programmnamen der Mutanten, die die „stammesgeschichtliche" Entwicklung der Mutanten enthalten (vgl. CREATE_NAME).

- Obige Fragen mit unterschiedlicher „Populationsdichte" (Steuerung über MORE und PERCENT)
- viele weitere Fragen.

Das SIMULA-Programm für MOD3 bietet also ebenfalls ein weites Experimentierfeld. Leider kann die eine oder andere der obigen Fragestellungen im Rahmen dieser Arbeit nicht mehr näher untersucht werden.

V. Aufwand:

Ohne in Einzelheiten zu gehen ist klar, daß sowohl für den Speicherplatzaufwand, als auch für die Laufzeit, die für das Programm für MOD2 gemachten Aussagen gültig sind. Der Aufwand wächst also i.a. exponentiell mit der Anzahl der Speicherdurchläufe. Gehemmt wird dieses Wachstum durch einen Faktor, der um so einflußreicher ist, je höher die zulässige Belegungsdichte des simulierten Speichers ist (vgl. 8.3.3.IV.). Ein gewisser Mehraufwand wird durch die komplizierten Datenstrukturen und die mutationsgenerierenden- und verwaltenden Prozeduren bewirkt (siehe Abb. 9.2.3.A).

Prozedur	Aufwand für großes M
MUTANT	O(M)
CREATE_NAME	konstant
NEW_PROG	O(M)
NEW_ST	O(M)
NEW_CONFLICT	$O(M^2)$
	(M = Aktuelle Anzahl der vorhandenen Programmtypen)

Abb. 9.2.3.A

Dieser Mehraufwand fällt jedoch kaum ins Gewicht, zumal es realistisch ist, von kleinen Mutationsraten auszugehen, so daß M ebenfalls klein bleibt.

VI. Es gilt auch für das SIMULA-Programm für MOD3 bzgl. des Speichererweiterungsmechanismus die Bemerkung 8.3.3.IV..

VII. Im SIMULA-Programm für MOD3 unterscheiden sich Mutanten von ihren Originalprogrammen in genau einer modellrelevanten Größe um maximal 100% (vgl. Beschreibung der Prozedur MUTANT). Dieser Spielraum von 100% ist willkürlich gewählt und ließe sich sicher auch in Form eines variablen Parameters festlegen. Die kleinstmögliche Mutationsrate beträgt in der SIMULA-Version von MOD3 10^{-8}. Dieser Wert orientiert sich an der Biologie und ist im Zusammenhang mit Evolution bei Rechnerprogrammen zumindest fraglich. Es bietet sich daher an, auch diesen Wert durch einen variablen Eingabeparameter zu ersetzen.
Analog: kleinstmögliche Rate für Letalmutationen.

Literaturverzeichnis

[1] Beilner, H. : Betriebssysteme
Vorlesungsskript, UNI DO, WS 1976/77

[2] Bertalanffy L. von : General System Theory
George Braziller, New York, 1968

[3] Brainerd/Landweber : Theory of Computation
John Wiley & Sons, 1974

[4] Claus, V. : Rekursive Funktionen
Vorlesungsbegleitmaterial, UNI DO, WS 1974/75

[5] Ehrich, H.D. : Berechenbarkeit,
Vorlesungsskript, UNI DO, WS 1977/78

[6] Geschwind, H.W. : Design of Digital Computers
- an Introduction, Springer-Verlag New York
- Wien, 1967

[7] Hoffmann, G. : Programmiersprachen und ihre
Übersetzer, Vorlesungsbegleitmaterial, UNI DO,
SS 1977

[8] Holland, J., An Arbor, Michigan, USA :
Studies of the spontaneous emergence of
self-replicating systems using cellular automata
and formal grammars,
Aus: Lindemayer/Rosenberg, Hrsg. : Automata,
Languages, Development, North-Holland Publishing
Company - 1976

[9] Hopcroft, Ullman : Formal Languages and their Relation to Automata, Addison-Wesley, 1969

[10] Jensen, K. und Wirth, N. : pascal User Manual And Report, second edition, Springer-Verlag New York - Heidelberg - Berlin, 1978

[11] Kästner, H. : Architektur und Organisation digitaler Rechenanlagen, Teubner Stuttgart 1978

[12] Kästner, H. : Rechnerfeinstrukturen Vorlesungsskript, UNI DO, SS 1976

[13] Kaplan, R.W. : Der Ursprung des Lebens 2. Auflage, Georg Thieme Verlag Stuttgart 1978

[14] Lee, J.A.N. : Computer Semantics Van Nostrand Reinhold Company, 1972

[15] Linder, H. : Biologie J.B. Metzlersche Verlagsbuchhandlung Stuttgart

[16] Manna, Z. : Mathematical Theory of Computation McGraw-Hill 1974

[17] Reusch, B. : Grundlagen der theoretischen Informatik, Vorlesungsbegleitmaterial, UNI DO, SS 1977

[18] Rogers, H.Jr. : Theory of Recursive Functions and Effective Computability, McGraw-Hill Book Company, 1967

[19] Rohlfing, H. : SIMULA - Eine Einführung
B.I. Hochschultaschenbücher, Band 747

[20] Schnorr, C.P. : Rekursive Funktionen und ihre
Komplexität, Teubner Studienbücher, Band 24

[21] Schnupp, P. : Rechnernetze Entwurf und
Realisierung, De Gruyter - Berlin - New
York, 1978

[22] Siemens-System 7.000 Beschreibung und
Befehlsliste, Stand Januar 1976, Siemens
Aktiengesellschaft

[23] Siemens-System 7.000 Siemens-System 4004
Betriebssystem BS 1000 F-Assembler
Betriebssystem BS 2000 Assembler
Beschreibung, Stand Dezember 1977,
Siemens Aktiengesellschaft

[24] Siewing, R. Hrsg. : Evolution
Gustav Fischer Verlag, Stuttgart New York
1978

[25] SIMULA Programmer's Guide
Siemens-System BS 2000, Version 0.0 September 78

[26] Stone, H.S. Hrsg. : Introduction to Computer
Architecture, SCIENCE RESEARCH ASSOCIATES
INC., 1975

[27] Vogel/Angermann : DTV-Atlas zur Biologie, Band 2
Deutscher Taschenbuch Verlag, 1968

[28] Wirth, N. : Algorithmen und Datenstrukturen
Teubner Taschenbücher