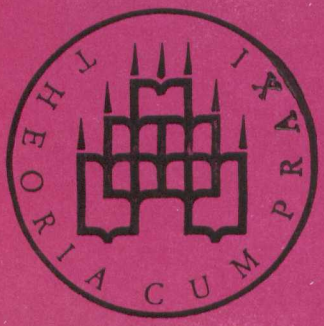
Rostocker Mathematisches Kolloquium

Heft 2



WILHELM-PIECK-UNIVERSITÄT
ROSTOCK

ROSTOCKER MATHEMATISCHES KOLLOQUIUM

Heft 2

1976

Wilhelm-Pieck-Universität Rostock
Sektion Mathematik

Redaktion: Abt. Wissenschaftspublizistik der Wilhelm-Pieck-Universität Rostock, DDR, 25 Rostock, Vogelsang 13/14
Fernruf 369 577

Verantwortlicher Redakteur: Dipl.-Ges.-Wiss. Bruno Schrage
Fachredakteur: Doz. Dr. rer. nat. Gerhard Maeß,
Sektion Mathematik

Herausgegeben von der Wilhelm-Pieck-Universität Rostock unter Genehmigungs-Nr. C 951/76
Druck: Ostsee-Druck Rostock, Werk II

Inhalt

Zum Geleit

Vom 23. - 27. 2. 1976 führte die Forschungsgruppe "Programmiersprachen" an der Sektion Mathematik und dem Rechenzentrum der Wilhelm-Pieck-Universität die vierte Arbeitssitzung der ALGOL-Gruppe DDR durch.

Diese Gruppe arbeitet innerhalb der Hauptforschungsrichtung "Mathematische Grundlagen der Informationsverarbeitung" und stellt gleichzeitig ein Verbindungsglied zur internationalen IFIP dar. Es waren Wissenschaftler aus mehreren Hochschulen und Universitäten der DDR anwesend. Auch die Praxis war durch den VEB Kombinat Robotron und die Bauakademie der DDR vertreten. Es wurden Vorträge zu den Komplexen Compilerbau, Anwendung und allgemeine Problemstellungen der Programmiersprachen gehalten. Diskussionsveranstaltungen zu wissenschaftsorganisatorischen Fragen der "Informationsverarbeitung" sowie deren Inhalt und Zielstellung fanden statt.

Die Tagungsteilnehmer betrachteten ihre Arbeit und die Ergebnisse als Beitrag zur Vorbereitung des IX. Parteitages der SED und begrüßten die dazu veröffentlichten Dokumente insbesondere die Abschnitte über Grundlagenforschung im Programmentwurf bzw. im Entwurf der Direktive für den Volkswirtschaftsplan 1976-1980.

Doz. Dr.sc. I.O. Kerner

Probleme und Konzeption der Implementierung einer Untersprache von ALGOL 68 auf dem Rechner R 4000

Die Arbeit beschäftigt sich mit einigen Aspekten der Implementierung einer Untersprache von ALGOL 68 auf dem Rechner R 4000. Insbesondere wird das Prinzip der Erweiterung des Standardvorspiels des Programmiersystems durch den Anwender dargelegt. Damit wird die implementierte Untersprache von ALGOL 68 mit ihren Möglichkeiten der Definition problem- und nutzerangepaßter Datenstrukturen und Operatoren, der standardmäßigen Bereitstellung derselben und der bequemen Hantierung vorgefertigter Prozeduren und Operatoren zu einer Bezugssprache für ein an beatimmte Problemklassen anpassungsfähiges Fachsprachensyatem. Weiterhin wird der grundsätzliche Aufbau des Übersetzerprogramms erläutert.

1. Bemerkungen zu höheren Programmiersprachen und zum Sprachkonzept der Untersprache von ALGOL 68

1.1. Vorzüge höherer Programmieraprachen gegenüber Assemblersprachen

Seit der erfolgreichen Entwicklung von FORTRAN in den Jahren zwischen 1954 und 1957 kommen höhere (maschinenunabhängige) Programmiersprachen zunehmend gegenüber den (maschinenabhängigen) Aasembleraprachen in Anwendung.

Als Vorteile der höheren Programmiersprachen gegenüber den Asaembleraprachen sind folgende Punkte zu nennen, die diese Entwicklungstendenz bestimmen:

- Die maschinenunabhängige Darstellung der Ausdrucksmittel der höheren Programmiersprachen erfüllt sowohl eine mnemotechnische als auch paradigmatische Funktion. Daraus resultieren Vorzüge, wie leichte Erlernbarkeit der Sprache, einfache (an Muster orientierte) Formulierbarkeit der Algorithmen und gute Lesbarkeit der Programme.

- Unmittelbar in diesem Zusammenhang ist wesentlich, daß in den höheren Programmiersprachen problem- und nutzerangepaßte Datenstrukturen a priori für den Programmierer zur Verfügung stehen, und Operatoren über diese Datenstrukturen problem- und nutzerangepaßt definiert sind (Datenstrukturen: Gestalten, Variable, gereihte und strukturierte Werte von definierbarem Modus).
- Spezielle syntaktische Konstruktionen für bestimmte algorithmische Elemente wie
  . Formeln
  . Alternativen
  . Fallauswahl
  . Zyklen
  ermöglichen eine übersichtliche Programmstruktur und eine einfache Programmierung.
- Die automatische Zuweisung von Speicherplatz für die symbolisch adressierten Datenstrukturen durch das Programmiersystem selbst schließt viele Fehlerquellen bei der Programmierung aus.
- Gegenüber der Assemblerprogrammierung ist die Unterprogrammtechnik durch die automatische Parameter- und Rücksprungorganisation weiterausgebaut.

Alle diese Vorzüge äußern sich schließlich im leichteren Programmieren der Algorithmen, in geringeren Fehlermöglichkeiten bei der Programmierung, im einfacheren Testen der Programme und in einfacheren Fehlerkorrektur, was insgesamt im geringeren Testaufwand für Programme der höheren Programmiersprachen gegenüber Assemblerprogrammen in Tabelle 1 zum Ausdruck kommt.

	Assembler	PL I	FORTRAN	COBOL	ALGOL 60
Codieraufwand (vom PAP ausgehend)	100 %	32 %	40 %	47 %	32 %
Testaufwand	100 %	16 %	30 %	41 %	16 %

Tabelle 1 Vergleich zwischen höheren Programmiersprachen und Assemblersprachen bezüglich Codier- und Testaufwand nach /7/

Immer wesentlicher für die breitere Anwendung höherer Programmiersprachen wird die Eigenschaft, daß die Programme mit relativ kleinen Änderungen über unterschiedliche Rechnerfamilien austauschbar sind. Vor allem behalten getätigte Programmierinvestitionen auch bei gerätetechnischen Neuentwicklungen weitgehend ihren Wert. Umgekehrt werden Geräteentwicklungen von einer Rücksichtnahme auf den bestehenden Programmfundus in höherem Maße befreit.

1.2. Entwicklungslinien höherer Programmiersprachen

Heute hat sich bei der Entwicklung höherer Programmiersprachen bereits ein Differenzierungsprozeß in 2 Richtungen abgezeichnet:

- Entwicklung von problemorientierten Programmiersprachen zu Spezialsprachen
- Entwicklung zu Universalsprachen.

Über die Dialektik und Objektivität beider Tendenzen soll hier nicht gesprochen werden. In diesem Zusammenhang wird auf /3/ und /4/ verwiesen.

Die Möglichkeiten der Erweiterbarkeit weisen den universellen Charakter moderner höherer Programmiersprachen aus. Insbesondere kann eine implementierte Universalsprache durch ihre Eigenschaften der Erweiterbarkeit in gewissem Maße als Bezugssprache für ein Fachsprachenprogrammiersystem eingesetzt werden. Die Erweiterbarkeit zwecks Anpassung an spezielle Anwendungsfälle im Rahmen des Sprachgerüstes, das aus allgemeingültigen, voneinander unabhängigen und zueinander paßfähigen Konzeptionen besteht, wird mehr zum dominierenden Merkmal der Universalsprachen.

Die Eigenschaft der Erweiterbarkeit der Sprache über das Prozedurkonzept hinaus gibt dem Programmierer die Möglichkeit der Definition problemangepaßter Ausdrucksmittel:

- Erweiterung des Spektrums der Wertarten durch Zurückführung auf bereits definierte zur Darstellung problemorientierter

Datenstrukturen (Modusdeklaration).
- Definitionsmöglichkeiten von Operatoren mit festzulegender Priorität zur problemangepaßten Formeldarstellung (Prioritäts- und Operatordeklaration).
- Einführung neuer syntaktischer Konstruktionen als algorithmische, problemangepaßte Sprachelemente durch Zurückführung auf die in der Sprache vorhandenen Elemente (dieses Prinzip ist bereits im System DEPOT /1/ verwirklicht, in ALGOL 68 noch nicht).

Die Forderung nach Erweiterbarkeit bedingt die "orthogonale" Gestaltung der in der Sprache standardmäßig vorhandenen Konzepte, wie sie beim Entwurf der Universalsprache ALGOL 68 beachtet wurde.

1.3. Das Konzept der Untersprache von ALGOL 68 für R 4000

Die Implementierung der Sprache ALGOL 68 auf dem Rechner R 4000 legt nach den Auffassungen der Implementatoren bei Berücksichtigung der konkreten Bedingungen, der Möglichkeiten der Rechenanlage und der bereits für die Rechenanlage vorhandenen Programmsysteme (insbesondere des Betriebssystems) die zu realisierende Sprachversion fest. Die vorgesehene Sprachversion ist weitgehend eine Untersprache von ALGOL 68 mit dem Haupteinsatzfeld wissenschaftlich-technischer und ökonomischer Berechnungen.
Gegenüber der vollen Sprache sind folgende Einschränkungen gemacht worden:
- Es ist keine Parallelverarbeitung und damit keine Semaphortechnik im Sinne des ALGOL 68-Berichts möglich; d.h., es ist keine parallele Klausel definiert.
- Da keine kollaterale Programmabarbeitung möglich ist, erscheint der Begriff der kollateralen Klausel nicht explizit in der Syntax, sondern nur indirekt als Display zur wertmäßigen Belegung von Reihungen und Strukturen.
- Unter Beachtung der zur Verfügung stehenden Speicherkapazität und zur Erzielung höherer Laufzeiteffektivität ist keine

dynamische Modusprüfung erlaubt. Somit entfallen der Vereinigungsmodus und die Konformitätsklausel.
- Vorerst ist keine heap-Technik vorgesehen. Damit gibt es keine flexiblen Reihungen. Eine Zeichenkettenverarbeitung ist deshalb nur im Rahmen von Reihungen des Modus char möglich, wofür die entsprechenden Operatoren zur Verfügung stehen.
- Eingeschlossene und serielle Klauseln sind nur in syntaktisch starker Position zulässig.
- Die Ein- und Ausgabemöglichkeiten sind reduziert und dem Betriebssystem angepaßt.
- Überladene Operatoren müssen mit der gleichen Priorität deklariert werden.

Zielstellung beim Sprachentwurf war es, die Möglichkeiten der Erweiterbarkeit, wie sie die volle Sprache ALGOL 68 vorsieht, bei der Implementierung der Untersprache weitgehend zu berücksichtigen. Daraus resultieren folgende Vorgaben:

- keine wesentlichen Einschränkungen für Deklarationen und Einzelklausel, insbesondere Realisierung der Modus- und Operatordeklaration
- selbständige Compilierung von Prozeduren und Operatoren und Verwendung von Codeprozeduren und Codeoperatoren im Anwenderprogramm. Für die Realisierung einer effektiven Parameterübergabe der selbständig kompilierten Prozeduren und Operatoren kann ein globaler Speicher für statisch definierte Objekte (Variable, Reihungen, Strukturen) in Verbindung mit einem neu eingeführten Globalgenerator genutzt werden.
- Erweiterbarkeit der Menge der standardmäßig definierten Modi, Prozeduren und Operatoren.

2. Realisierung des Prinzips der Erweiterung des Standardvorspiels im Programmiersystem ALGOL 68-R 4000 und der selbständigen Compilierbarkeit

Wie aufgezeigt wurde, sind die Erweiterungsmöglichkeiten des Standardvorspiels sehr entscheidend für den Einsatz des Programmiersystems ALGOL 68-R 4000 als Fachsprachenprogrammier-

system, das an bestimmte Problemklassen anpassungsfähig ist.

Die wahlweise, durch den Nutzer des Systems zu bestimmende Erweiterung des Standardvorspiels wird durch den besonderen Aufbau des Anwenderprogramms und durch spezielle Kommandos zur Modifikation der Arbeitsweise des Übersetzerprogramms erzielt.

Ein Anwenderprogramm besteht wahlweise aus dem sogenannten Anwendervorspiel und der den eigentlichen Algorithmus darstellenden markierten abgeschlossenen Klausel.

Anwenderprogramm ::= [pr <Anwendervorspiel> pr]$^{0:1}$
[<Marke> : <abgeschlossene Klausel>]$^{0:1}$
HALT

Für die Erweiterung des Standardvorspiels ist das Anwendervorspiel entscheidend, wobei die Tabelle 2 die im Anwendervorspiel möglichen Deklarationen aufführt.

Deklarationen im Anwendervorspiel	Reaktion des Übersetzerprogramms
Modusdeklaration	Erweiterung der Basissymboltabelle Darstellung des Modusbaumes
Prioritätsdeklaration	Tabelleneintragung der Priorität
Operator- und Prozedurdeklaration	Tabelleneintragung des Indikators bzw. Identifikators sowie des Operator- bzw. formalen Prozedurdeklarierers Ausgabe des Bibliotheksobjektprogramms (falls kein Codeoperator bzw. keine Codeprozedur)
Globaldeklaration	Tabelleneintragung Generierung der für das Ladeprogramm bzw. den Programmverbinder notwendigen Instruktionen bei Verwendung der globalen Objekte in den externen Prozeduren, Operatoren bzw. abgeschlossenen Klauseln.

Tabelle 2 Zu den Deklarationen im Anwendervorspiel

Operator- und Prozedurdeklarationen im Anwendervorspiel definieren Operatoren und Prozeduren, die extern bezüglich der abgeschlossenen Klausel des Anwenderprogramms sind.

Mit Hilfe des Anwendervorspiels werden ALGOL 68-Prozeduren und -Operatoren selbständig compiliert. Sie bilden zusammen mit den Assembler- und FORTRAN-Unterprogrammen die externen Prozeduren und Operatoren. Ihre Verwendung als externe Prozeduren und Operatoren im Anwenderprogramm setzt ihre Definition als Codeprozeduren bzw. Codeoperatoren

⟨Codeprozedur⟩ ::= ⟨formaler Prozedurdeklarierer⟩
⟨Identifikator⟩ = code

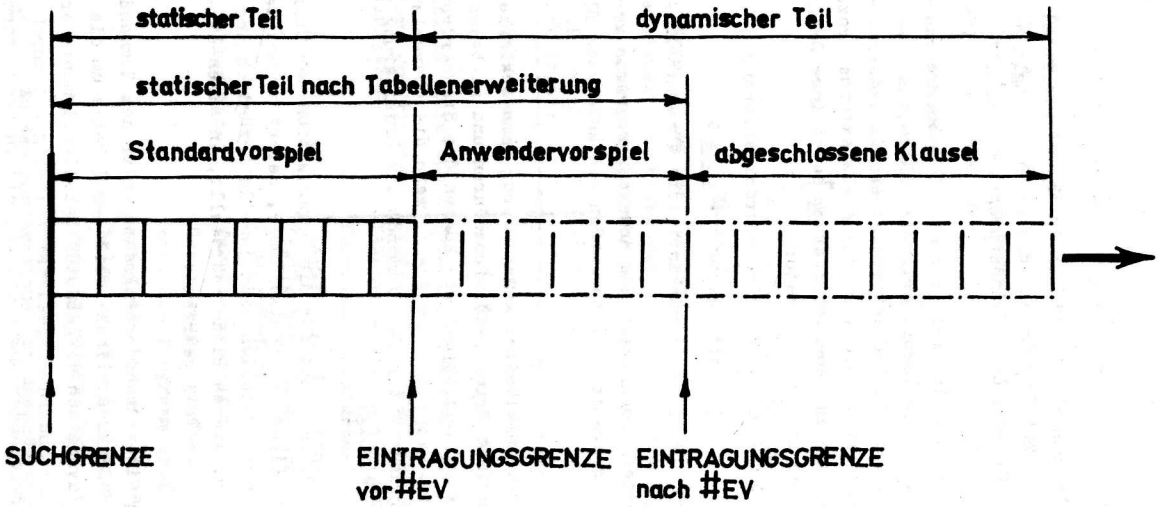
⟨Codeoperator⟩ ::= ⟨Operatordeklarierer⟩ ⟨Operator⟩ = code

in einem beliebigen Blockniveau des Anwenderprogramms voraus, wobei das Anwendervorspiel gewissermaßen den äußersten Block darstellt.

Alle im Anwendervorspiel eines Anwenderprogramms deklarierten Modi, Operatoren (mit Priorität), Prozeduren und globalen Variablen, Reihungen und Strukturen können das Standardvorspiel des Programmiersystems erweitern, wenn die Übersetzung des Anwenderprogramms unter dem Kommando zur Erweiterung des Vorspiels "#EV" erfolgt.

Das Übersetzerprogramm führt für die Organisation der Übersetzung die Identifikator-, Basissymbol-, Selektor- und Gestaltentabelle. Diese Tabellen sind jeweils bezüglich der laufenden Übersetzung, wie es Bild 1 darstellt, in einen statischen und dynamischen Teil getrennt.

Eine Übersetzung eines Anwenderprogramms unter dem Kommando "# EV" bewirkt, daß der statische Teil der Tabelle um die im Anwendervorspiel dynamisch aufgebauten Tabellenelemente erweitert wird. Das Standardvorspiel ist beliebig oft durch die Übersetzung entsprechender Anwendervorspiele zu vergrößern.


statischer Teil
dynamischer Teil
statischer Teil nach Tabellenerweiterung
Standardvorspiel
Anwendervorspiel
abgeschlossene Klausel
SUCHGRENZE
EINTRAGUNGSGRENZE vor #EV
EINTRAGUNGSGRENZE nach #EV

Eine Rücksetzung des Standardvorspiels auf den durch die Implementation festgelegten Teil ist durch das Kommando zum Verkürzen des Vorspiels "#VV" möglich.

Zu bemerken ist, daß für die Identifikatoren externer Operatoren und Prozeduren, sowie für globale Variable, Reihungen und Strukturen nur 4 Zeichen signifikant sind, wie es der Programmverbinder des Rechners R 4000 vorschreibt. Die Identifikatoren, die die Indikatoren externer Operatoren bilden, und der externen Prozeduren, sowie die Identifikatoren der globalen Variablen, Reihungen und Strukturen müssen sich unterscheiden.

3. Grundkonzeption des Übersetzerprogramms

Die Gerätetechnik und die vorhandene Programmtechnik ist stets von wesentlichem Einfluß auf die Implementierung einer höheren Programmiersprache auf einem Rechnersystem. Eine Übertragung des Compilers auf ein anderes Rechnersystem ist damit nicht ohne weiteres möglich. Damit steht der Allgemeinwert des Compilers im Widerspruch zum gesamten Implementierungsaufwand.

Die vorgesehene Implementation unternimmt den Versuch, diesen aufgezeigten Widerspruch abzubauen durch

- den Einsatz einer maschinenunabhängigen, compilerbeschreibenden Sprache (CDL)
- die Festlegung einer maschinenunabhängigen Zielsprache des Compilers, da das Zielsprachniveau hauptsächlich die Phase der Codegenerierung, nicht aber die Phasen der lexikalen und syntaktischen Analyse des Compilers berührt.

Die grobe Struktur des Übersetzerprogramms ist im Bild 2 dargestellt. Es bedeuten:

L_{A68} - Quellsprache, die im wesentlichen eine Untersprache von ALGOL 68 ist

L_S - Syntaxsprache, die durch den Vorübersetzer bestimmt ist

L_Z - rechnerunabhängige Zielsprache (Zwischensprache)

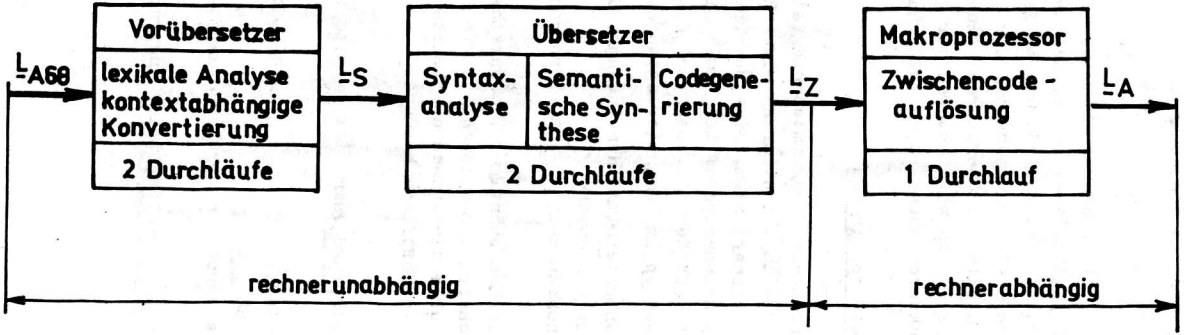


Abb. 2 Grobstruktur des Übersetzerprogramms

L_A - Assemblersprache SYPS 4000.

3.1. Vorübersetzer

Im wesentlichen wird in /2/ die Arbeitsweise des Vorübersetzers erläutert. Er führt in der Hauptsache folgende Aufgaben aus:

- zeichenweises Durchmustern des Quellprogramms und Ersetzen der aus mehreren Zeichen bestehenden Basissymbole durch ein Internzeichen
- kontextabhängige Umbenennung homonym und synonym verwendeter Basissymbole in entsprechende Internzeichen (diese Aufgabe begründet die Arbeit des Vorübersetzers in 2 Durchläufen)
- Herauslösen von Identifikatoren, Zeichenkettengestalten, Operatoren und Modusindikatoren und ihre Ersetzung durch Internzeichen
- Erkennung der Blockstruktur und Aufbau sowie Verwaltung der Spezifikationstabellen (Identifikator-, Gestalten-, Basissymbol- und Selektortabelle), Behandlung der Deklarationen im 1. Durchlauf, der Verwendungen im 2. Durchlauf /6/

Ein Programm der Syntaxsprache, das der Vorübersetzer liefert, widerspiegelt die Zeilenstruktur des Quellprogramms. Jedes Zeichen des Programms der Syntaxsprache besteht aus mindestens 2 Wörtern des R 4000. Die im Bild 3 verwendeten Abkürzungen bedeuten:

POS	- Position des Zeichens entsprechend Quellprogramm
L	- Länge der rechnerinternen Darstellung eines Zeichens
MOD	- Modusangabe bei der Darstellung von Gestalten (mit Ausnahme von Zeichenketten bleiben die Gestalten im Zwischenprogramm und werden nur ihrem Modus entsprechend auf eine normierte Darstellung gebracht)
CODE	- Codenummer des Zeichens
ADR/ KONST	- Verweisadresse auf Tabellenelement (bei Identifikatoren, Indikatoren oder Zeichenkettengestalt) bzw. Zeichenfolge, die numerische Gestalten dar-

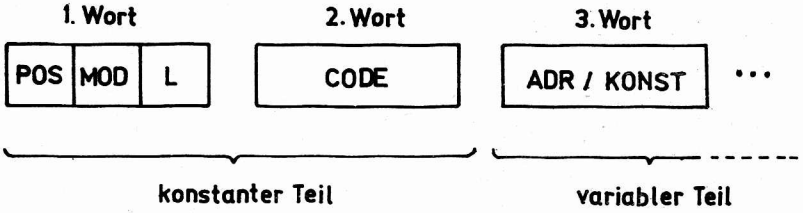


Abb. 3 Aufbau der Internzeichen der Syntaxsprache

stellt.

Die Widerspiegelung der Zeilenstruktur des Quellprogramms im Programm der Syntaxsprache und die Aufnahme der Positionsangabe in die Darstellung der Internzeichen ermöglichen weitgehend eine genaue Positionierung der Fehleranzeige bei lexikalen, syntaktischen aber auch semantischen Fehlern.

Die Darstellung der numerischen Gestalten sowohl im Syntaxprogramm als auch im Zielsprachprogramm erlaubt eine günstige Optimierung bei Operationen mit diesen.

3.2. Syntaktische Analyse, semantische Synthese und Codegenerierung

Im Übersetzer wird ein präzedenzgesteuertes bottom-up-Verfahren ohne Rücklauf verwendet, das mit begrenztem Kontext arbeitet /5/. Die Gesichtspunkte für die Auswahl eines solchen syntaktisch geleiteten Verfahrens waren:

- Die parametrisierte Arbeit syntaktisch geleiteter Analyseverfahren gestattet, Modifikationen der Syntax der Sprache ohne großen Änderungsaufwand vorzunehmen.
- Die Präzedenzrelationen ermöglichen ein schnelles Auffinden der reduzierbaren Zeichenreihe bei der Syntaxanalyse und eine sackgassenfreie Reduktion durch Zuhilfenahme begrenzten Kontextes. Damit kommt ein sehr zeiteffektives Verfahren zur Anwendung.
- Präzedenzgesteuerte Verfahren bieten günstige Möglichkeiten für die (frühzeitige) Fehlererkennung und Fehlerbehandlung.

Eine in /5/ begründete Speicherplatzabschätzung ergibt für Syntaxregeln, Präzedenzmatrix und Algorithmus einen Speicherplatzbedarf von ca. 6 K Wörten für die konzipierte Untersprache von ALGOL 68 bei komprimierter Abspeicherung der Präzedenzmatrix.

Im Übersetzer ist das Prinzip verwirklicht, daß eine feste Zuordnung zwischen syntaktischer Regel und semantischer Compileraktion besteht. Die Compileraktion umfaßt sowohl die semantische Synthese als auch die Codegenerierung. Es ist vorgesehen, für die Realisierung der Compileraktionen aufeinander abgestimmte elementare Funktionen im Übersetzer bereitzustellen.

Während die Syntaxanalyse bereits vollständig programmiert und ausgetestet wurde, konnte für die Compileraktionen das Prinzip bisher nur für die Modusdeklaration mit guten Ergebnissen überprüft werden.

3.3. Zielsprache und Makroprozessor

Der Definition der Zielsprache, die rechnerunabhängig ist, liegt eine interpretierende Modellmaschine zugrunde, deren Struktur und Organisation an reale Rechner anpaßbar ist und die verallgemeinerte Eigenschaften höherer Programmiersprachen widerspiegelt /8/.
Makroanweisungen bilden die Zielsprache,

- deren Wirkung durch Angabe der Abläufe in der interpretierenden Modellmaschine zurückgeführt wird (Semantik I)
- deren Semantik implementierungsabhängig auf die Makrodefinitionen führt, die den Makroprozessor (Bild 2) auf die reale Assemblersprache einstellen (Semantik II).

Somit erzeugt der Makroprozessor durch die Abarbeitung der in der Semantik II gegebenen Makrodefinitionen (Metaprogramm) den Ersetzungstext, dessen Abarbeitung den Aktionen der Modellmaschine (Semantik I) entspricht.

Die Zielsprache berücksichtigt folgende Grundanforderungen:

- Die Zielsprache stellt Makroanweisungen aus der Verallgemeinerung grundsätzlicher Eigenschaften höherer Programmiersprachen zur Verfügung, d.h. gleichartige Sprachelemente verschiedener höherer Programmiersprachen können durch diese Makroanweisungen dargestellt werden.

- Für die Operanden der Makroanweisungen erfolgt in der Zielsprache eine symbolische Programmobjektidentifikation, da die Speicherplatzzuordnung typisch rechnerabhängig ist.
- Quellprogrammobjekte (Gestalten), die ihren Wert selbst darstellen, werden ohne wesentliche syntaktische Veränderung im Zielsprachniveau widergespiegelt.
- Die Blockstruktur wird weitgehend in das Zielsprachniveau übernommen, da die notwendigen Aktionen von der konkreten Speicherverwaltung abhängen.
- In der Zielsprache werden wertartunabhängige Verknüpfungsoperatoren verwendet. Ihre Parametrisierung durch die Wertart der in die Operation eingehenden Operanden bewirkt eine Invarianz bezüglich der durch die MOS und des Befehlsvorrates der Rechner gegebenen Möglichkeiten der Wertartvermischung und -konvertierung.
- In der Zielsprache stehen Makroanweisungen für die Prozedurorganisation, Linearisierungsoperatoren zur Auflösung von verschachtelten Anweisungen und zur Deklaration von Programmobjekten zur Verfügung. Insbesondere ist die Deklaration von Objekten der rechnerorientierten Wertart Adresse möglich.
- Im Zielsprachniveau werden logische Speichertypen unterschieden, die aus den Wertarten und anderen Eigenschaften der Programmobjekte der zu implementierenden Programmiersprache abgeleitet werden, die unterschiedlich organisiert sind und in die die betreffenden Objektwerte abgebildet werden können (z.B. Blockstrukturspeicher, Akkumulationsspeicher, Externspeicher, Konstantenspeicher).

Die Vorteile des als Makrosprache festgelegten Zielsprachniveaus liegen in der Parametetrisierbarkeit der Makroanweisungen, in der Erweiterbarkeit der Makrosprache, in der Effektivität und Optimierung sowie in der Interpretierbarkeit, die insgesamt eine Implementierungsfreundlichkeit ergeben.

Der Grundaufbau der Makroanweisungen p_i entspricht der Form

$$p_i = op_i\ (opd_{i_1}, opd_{i_2}, \ldots, opd_{i_n}),$$

wobei

op_i die Makrobezeichnung

opd_j die die Makroanweisungen parametrisierenden aktuellen Makrooperanden sind.

Jedem Makrooperator op_i ist im Makroprozessor eine Makrodefinition $opdef_i$ zugeordnet

$$opdef_i = op_i\ (fopd_{i_1},\ fop_{i_2},\ \ldots,\ fopd_{i_n}),$$

wobei

$fopd_j$ die formalen Makrooperanden darstellen.

Diese Makrodefinition wird durch eine Folge von Modellanweisungen dargestellt. Im Makroprozessor werden unterschieden

- die Modellanweisung 1. Art
  Das sind die "formalen Assembleranweisungen", die über einen Substitutionsmechanismus aktueller für formale Makrooperanden als Bestandteil in den Ersetzungstext für eine Modellanweisung eingehen.
- die Modellanweisung 2. Art
  Die Erzeugung des Ersetzungstextes (Makroexpansion) wird in Abhängigkeit bestimmter Makro- (steuer-) operanden gesteuert ("Steuermodellanweisungen").
- die Modellanweisung 3. Art
  Die "Makroprozessoranweisungen" operieren mit den internen Datenstrukturen des Makroprozessors.

Die Arbeitsweise des Makroprozessors läßt sich durch folgende Schritte erklären:

1. **Klassifizierung** der laufenden Makroanweisungen p_i durch die **Makrobezeichnung** op_i über die Makrodefinitionstabelle

$$DEFTAB = \left\{ (op_i,\ opdef_i) \;\middle|\; op_i \text{ identifiziert } opdef_i,\ i = 1(1)i_{max} \right\}$$

2. Substitution der formalen durch die aktuellen Makrooperanden

$fopd_{i_j} := opd_{i_j}$ für $j = 1(1)n$

und Abarbeitung der Makrodefinition.

3. Ausgabe des aktualisierten Ersetzungstextes

4. Bestimmung der nächsten Makroanweisung

4. Zusammenfassung

ALGOL 68 ist als Bezugssprache für die Entwicklung problembezogener Programmiersysteme ein guter Ausgangspunkt. Das in der Arbeit aufgezeigte Konzept der Implementierung einer Untersprache von ALGOL 68 berücksichtigt insbesondere die Forderungen der einfachen Erweiterbarkeit der Menge der standardmäßig definierten Modi, Prozeduren und Operatoren. Die Implementierung der noch für den Anwender ansprechenden Untersprache von ALGOL 68 ist auf der kleinen bis mittleren Rechenanlage R 4000 durch einen Fünfpaßcompiler vorgesehen. Das Programmiersystem ALGOL 68-R 4000 soll vor allem als Experimentierbasis zur Herausbildung problembezogener (insbesondere der Prozeßrechentechnik angepaßter) Sprachelemente dienen.

5. Literaturverzeichnis

/1/ Bormann, J.
Lötzsch, J.
Definition und Realisierung von Fachsprachen mit DEPOT
TU Dresden Kollektivdissertation 1974

/2/ Jähnlich, H.
Loeper, H.
Otter, W.
Führung von Tabellen und Behandlung von Kontextabhängigkeiten im lexikalen Analysator eines Übersetzerprogramms für eine Untersprache von ALGOL 68
MKÖ IV Dresden 1975

/3/ Lehmann, N.J.
Probleme und Bedeutung der Entwicklung von Programmierungssprachen
EIK 11 (1975) H. 4 - 6

/4/ Lehmann, N.J. Einige methodische Aspekte der Entwicklung und Nutzung höherer Programmiersprachen Rechentechnik/Datenverarbeitung 3. Beiheft 1976

/5/ Loeper, H.
Horn, M.
Brankowa, E. Präzedenzgesteuerte Syntaxanalyse einer Untersprache von ALGOL 68 MKÖ IV Dresden 1975

/6/ Otter, W.
Loeper, H.
Grabner, H. Zur Strukturierung der Spezifikationstabellen von Übersetzerprogrammen für blockorientierte Programmiersprachen Wissenschaftliche Zeitschrift TU Dresden 24. Jhg. (1975) H. 5

/7/ Schmitz, P. Wirksamkeit von Programmiersprachen Betriebswirtschaftlicher Verlag Dr.Ph. Gabler, Wiesbaden 1972

/8/ Schreckenbach, R.
Göstl, H. Motivierung einer makroähnlichen Zielsprache für Compiler Vortrag Problemseminar "Übersetzerprogrammtechnik" Sektion Informationsverarbeitung der TU Dresden, Weißig 1975

/9/ Stiller, G. ALGOL 68 - Begriffe und Ausdrucksmittel BSB B.G. Teubner Verlagsgesellschaft Leipzig 1974

/10/ van Wijngaarden, A.
u.a. Proposed Revised Report on the Algorithmic Language ALGOL 68 Acta Informatica vol. 5 (1975) Fasc. 1 - 3

eingegangen: 23. 4. 1976

Anschrift des Verfassers

Doz. Dr.rer.nat. Hans L o e p e r
Technische Universität Dresden, Sektion Informationsverarbeitung
DDR 8027 Dresden

Günter Riedewald

Compilermodell auf der Grundlage einer Grammatik syntaktischer Funktionen

In /Rie 751/ wurde gezeigt, wie man die Realisierung der statischen Semantik (auch semantische Analyse genannt) mit Hilfe einer Grammatik syntaktischer Funktionen in der Phase der syntaktischen Analyse durchführen kann. In /Rie761/ wurde diese Methode erweitert durch die Einbeziehung der Objektcodegenerierung. Damit wurde nachgewiesen, daß es möglich ist, auf der Grundlage von Grammatiken syntaktischer Funktionen Compiler zu konstruieren.

In den weiteren Abschnitten werden Teile eines Compilers kurz beschrieben, der nach den in /Rie 761/ dargestellten Prinzipien erstellt wurde. Dieser Compiler übersetzt ASPLE-Programme in FORTRAN 63-Programme.

Auaführliche Informationen findet der Leser in der angeführten Literatur. Eine umfassende Darstellung des Compilers kann /Rie 763/ entnommen werden.

1. Grammatik syntaktischer Funktionen für ASPLE

Die Programmiersprache ASPLE wurde 1973 entwickelt. Sie wurde nicht für die Praxis geschaffen und enthält deshalb nur eine geringe Anzahl verschiedener syntaktischer Konstruktionen. Hierzu gehören u.a. Laufanweisungen mit while-Element, bedingte Anweisungen, arithmetische und logische Ausdrücke, Zuweisungen, eine sehr einfache Ein- und Ausgabe. ASPLE wurde unter dem Einfluß von E. Dijkstras strukturierter Programmierung entwickelt. Verschiedene Formen der Beschreibung von ASPLE-u.a. mit Hilfe einer zweistufigen von Wijngaarden-Grammatik - findet der Leser in /U 73/.

Die allgemeine Form einer Grammatik syntaktischer Funktionen ist eine modifizierte Form der in /Rie 752/ angeführten. Hier

soll nur die Beschreibung der Regeln von bedingten Anweisungen in dieser Form erfolgen. Eine vollständige Beschreibung kann /Rie 763/ entnommen werden.

1 conditional(c): if symbol, value(bool;V), then symbol, statement train (ST), elseend (EE), COND (C;V; ST; EE)

2 elseend(0): fi symbol

3 elseend(ST): else symbol, statement train (ST), fi symbol

Aus den Regeln folgen als allgemeine Form der bedingten Anweisung

(1) if <Bedingung> then <Anweisungsfolge> else <Anweisungsfolge> fi

(2) if <Bedingung> then <Anweisungsfolge> fi

Die semantische Funktion COND dient zur Objektcodegenerierung, d.h. sie erzeugt in der Objektsprache eine Anweisungsfolge, die die gleiche semantische Bedeutung hat wie die zu übersetzende Anweisung. Der generierte Objektcode wird dem ersten Parameter von COND zugeordnet. Über den zweiten Parameter wird vorausgesetzt, daß ihm der Objektcode der Bedingung als Wert zugeordnet ist. Dem dritten Parameter ist immer der Objektcode der Anweisungsfolge nach dem then zugeordnet. Der Wert des vierten Parameters ist 0 (im Falle der bedingten Anweisung (2)) bzw. gleich dem Objektcode der Anweisungsfolge nach dem else (im Falle der bedingten Anweisung (1)).

Auf ähnliche Weise werden durch die Grammatik syntaktischer Funktionen für ASPLE-Sprachkonstruktionen beschrieben.

2. Compileraufbau

Zur Beschreibung der Funktionsweise des Compilers werden die bedingten Anweisungen und ihre Beschreibung in den Regeln 1, 2, 3 benutzt.

Der Compiler hat den üblichen logischen Aufbau, d.h. er besteht aus dem lexikalischen Analysator, dem Syntaxanalysator und der Codeerzeugung.

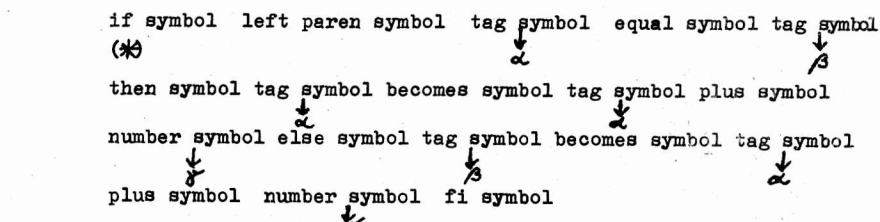
2.1. Lexikalische Analyse

Ein ASPLE-Programm wird durch die lexikalische Analyse auf übliche Art und Weise in eine interne Darstellung überführt. Zahlen und Identifikatoren werden in eine Tabelle übernommen und im internen Programm wird eine interne Kodierung, die für alle Zahlen bzw. alle Identifikatoren gleich ist, zusammen mit einem Zeiger auf die Eintragung der Zahl bzw. des Identifikators in der Tabelle abgesetzt.

Beispiel: ASPLE-Anweisung

if (x=y) then x := x+1 else y := y+1 fi

Interne Darstellung



2.2. Syntaktische Analyse

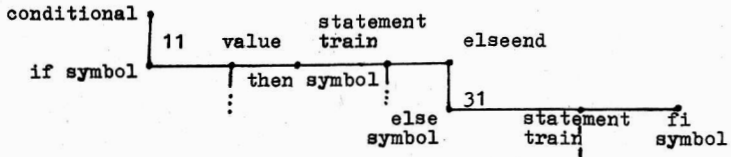
Das durch die lexikalische Analyse erzeugte Internprogramm wird bezogen auf eine Basisgrammatik syntaktisch analysiert. Die Methode ist entsprechend der Basisgrammatik frei wählbar.

Die Regeln der Basisgrammatik, die aus 1, 2, 3 durch Weglassen aller Parameterlisten und der semantischen Funktion COND entstehen, sind

11 **conditional:** if symbol, value, then symbol, statement train, elseend
21 **elseend:** fi symbol

31 elseend: else symbol, statement train, fi symbol

Das obige Beispiel syntaktisch analysiert ergibt dann als Syntaxbaum



Die Numerierung an den Kanten gibt die Nummer der angewendeten Regel der Basisgrammatik an. Durch die Punkte werden Teilbäume bezeichnet, die hier nicht von Interesse sind.

2.3. Codeerzeugung

Für die Codeerzeugung sind nötig eine Korrespondenzmenge von Regeln, die Informationen über die Arbeit mit den Parametern geben und die aus der Grammatik syntaktischer Funktionen durch Weglassen der syntaktischen Funktionen entsteht, ein Parameterwertvermittlungssystem, Unterprogramme zur Realisierung der syntaktischen Hilfsfunktionen und der semantischen Funktionen und eine Information darüber, welche Regeln der Korrespondenzmenge angewendet werden sollen.

Die Korrespondenzregeln für die bedingten Anweisungen sind

111 (C): , (bool; V), , (ST), (EE), COND (C; V; ST; EE)
211 (O):
311 (ST): , (ST),

Als Parameterwertvermittlungssystem wird das in /Rie 761/ angegebene verwendet. Die Korrespondenzregeln lassen sich durch dieses System folgendermaßen interpretieren:

111 a) Wenn der Wert des 1. Parameters der syntaktischen Funktion value bool ist, dann wird in der Parameterwertvermittlung fortgesetzt, sonst gibt es eine Fehlermeldung und die Parameterwertvermittlung wird ab-

gebrochen.

b) Der Wert des 2. Parameters von value wird dem 2. Parameter von COND zugewiesen.

c) Der Wert des Parameters von statement train wird dem 3. Parameter von COND zugewiesen.

d) Der Wert des Parameters von elseend wird dem 4. Parameter von COND zugewiesen.

e) COND wird aufgerufen.

f) Der Wert des 1. Parameters von COND wird dem Parameter von conditional zugewiesen.

211 Dem Parameter von elseend wird 0 zugewiesen

311 Der Wert des Parameters von statement train wird dem Parameter von elseend zugewiesen.

Die Information, welche Regeln der Korrespondenzmenge angewendet werden sollen, werden durch die syntaktische Analyse geliefert. Werden bei der syntaktischen Analyse die Regeln der Basisgrammatik i_1, ..., i_n angewandt, so müssen zur Codeerzeugung die Korrespondenzregeln j_1, ..., j_n in dieser Reihenfolge angewendet werden. Unter der Voraussetzung der obigen Numerierung besteht zwischen i_1 und j_1 folgende Beziehung: j_1 entsteht aus i_1 durch Anfügen der Ziffer 1 von rechts.

Beispiel: Im obigen Beispiel wurden die Regeln ..., 31, 11 bei der Syntaxanalyse angewendet. Zur Codeerzeugung müssen die Regeln ..., 311, 111 angewendet werden.

Die Unterprogramme zur Realisierung der syntaktischen Hilfsfunktionen dienen zur semantischen Analyse (Realisierung der statischen Semantik), d.h. mit ihrer Hilfe werden z.B. Kontextbedingungen überprüft. Die Unterprogramme zur Realisierung der semantischen Funktionen dienen der eigentlichen Codeerzeugung, wie z.B. COND.

Die mögliche Arbeitsweise eines solchen Unterprogrammes soll am Beispiel des Unterprogrammes zur Realisierung von COND erklärt

werden.
Seine Aufgabe und die Rolle seiner Parameter wurden schon unter 1. erklärt. Zum besseren Verständnis ist es nötig, sich klarzumachen, wie der Objektcode für eine bedingte Anweisung in ASPLE aussehen soll. In der Einleitung wurde davon gesprochen, daß der Objektcode in FORTRAN 63 dargestellt sein soll. Diese Tatsache wurde bisher noch nicht verwendet, aber spätestens hier muß sie beachtet werden.

Es sind folgende Zuordnungen möglich:

ASPLE	FORTRAN 63	
if <Bedingung> then <Anwei.-F.> fi	IH = <C-Bedingung> IF(IH) A1, A2 A1<C-Anwei.-F.> A2 CONTINUE	/1/
if <Bedingung> then <Anwei.-F1> else <Anwei.-F2> fi	IH = <C-Bedingung> IF(IH) A1, A2 A1 <C-Anwei.-F1> GOTO A3 A2 <C-Anwei.-F2> A3 CONTINUE	/2/

Dabei ist IH eine logische Hilfsvariable, <C-Bedingung> der Objektcode der <Bedingung>, <C-Anwei.-F.> der Objektcode von <Anwei.-F.>, <C-Anwei.-F1> der Objektcode von <Anwei.-F1>, <C-Anwei.-F2> der Objektcode von <Anwei.-F2>, A1, A2 und A3 durch einen Generator erzeugte Befehlsnummern.

Da die Codeerzeugung entsprechend der Aufstellung des Syntaxbaumes geschieht, würde eine einfache Aneinanderreihung der einzelnen erzeugten Objektcodefolgen zu falschen Objektprogrammen führen. Um das zu vermeiden, werden die erzeugten Anweisungen in Objektcode in einer Liste PRO und die erzeugten Ausdrücke in Objektcode in einer Liste PROH abgespeichert. Eine Liste FOL bzw. FOH dient zur Buchführung über die abgespeicherten Anweisungen bzw. Ausdrücke und über Verkettungen von Anweisungen zu größeren Programmteilen (FOL).

Ein Listenelement in FOL enthält 4 Angaben: Anfangs- und Endadresse und evtl. Fortsetzungsadresse eines Objektprogrammstückes in PRO und eine Hilfsinformation.
Ein Listenelement in FOH enthält neben der Anfangs- und Endadresse eines Objektcodeausdruckes in PROH noch 2 weitere Hilfsinformationen.

Die Parameter sind nun keine Programmteile in Objektcode, sondern Zeiger auf Eintragungen in FOL bzw. FOH.
Der 1. und 3. Parameter von COND sind damit Zeiger in die Liste FOL, der 2. Parameter zeigt in die Liste FOH und der 4. Parameter ist entweder 0 oder ein Zeiger in die Liste FOL.

Die Werte des 2., 3. bzw. 4. Parameters seien a , b bzw. c.
COND arbeitet dann auf folgende Weise:

Schritt 1: Aus IH= und dem Objektcode der Bedingung, der über a greifbar ist, wird eine FORTRAN 63-Zuweisung hergestellt und in PRO abgespeichert. Die Anfangsadresse wird in einem neuen Element in FOL abgespeichert. Ein Zeiger d auf dieses Element wird dem 1. Parameter von COND zugeordnet.

Schritt 2: Die Anweisung IF(IH) A1, A2 wird erzeugt und in PRO abgespeichert. Die Endadresse in PRO wird in FOL bei d abgespeichert.

Schritt 3: Über die Adresse b wird die erste Anweisung der übersetzten Anweisungsfolge nach then erreicht. Sie bekommt zusätzlich die Befehlsnummer A1. b wird als Fortsetzungsadresse in FOL bei d abgespeichert.
e sei identisch mit b.

Schritt 4: Wenn die Fortsetzungsadresse von e 0 ist, dann wird mit Schritt 5 fortgesetzt. Sonst wird e die Fortsetzungsadresse von e bezeichnen und Schritt 4 wird wiederholt.

Schritt 5: Wenn g 0 ist, wird A2 CONTINUE erzeugt und in PRO abgespeichert. Die Anfangsadresse und die Endadres-

se werden in FOL in einem neuen Element bei f abgespeichert. Die Fortsetzungsadresse wird 0 gesetzt. f wird als Fortsetzungsadresse bei e abgespeichert. COND hat seine Arbeit beendet.
Wenn g nicht 0 ist, wird mit Schritt 6 fortgesetzt.

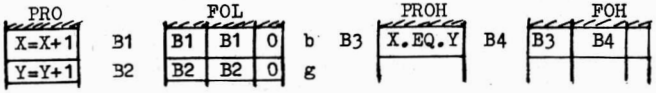
Schritt 6: GOTO A3 wird erzeugt und in PRO abgespeichert. Die Anfangs- und Endadresse werden in FOL in einem neuen Element bei f abgespeichert. Die Fortsetzungsadresse bei e wird auf f gesetzt. Die Fortsetzungsadresse von f wird auf g gesetzt. Über g wird die erste Anweisung der übersetzten Anweisungsfolge nach else erreicht. Sie bekommt zusätzlich die Befehlsnummer A2. g sei identisch e.

Schritt 7: Wenn die Fortsetzungsadresse von e 0 ist, dann wird mit Schritt 8 fortgesetzt. Sonst wird die Fortsetzungsadresse bei e mit e bezeichnet und Schritt 7 wird wiederholt.

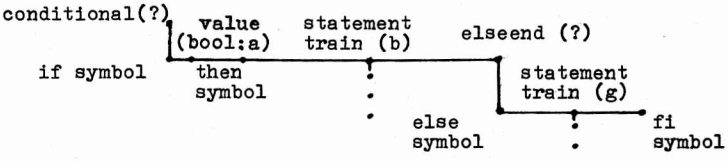
Schritt 8: A 3 CONTINUE wird erzeugt und in PRO abgespeichert. Die Anfangs- und Endadresse werden in FOL in einem neuen Element bei p abgespeichert. p wird als Fortsetzungsadresse bei e abgesetzt. Die Fortsetzungsadresse bei p wird auf 0 gesetzt.
COND hat seine Arbeit beendet.

Aus der Beschreibung der Arbeitsweise von COND ist zu ersehen, daß ein Großteil der Arbeit nur Organisationsarbeit ist, die auch bei der Generierung von Objektcode in einer anderen Sprache als FORTRAN 63 auftreten würde.

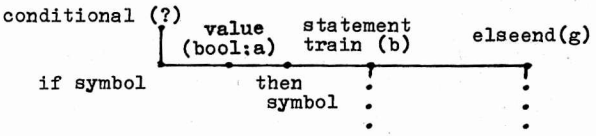
Die Codeerzeugung soll nun am Beispiel (*) demonstriert werden. Dazu wird vorausgesetzt, daß in PRO die Anweisungsfolgen nach then und else in Objektcode sind und das FOL die entsprechenden Informationen enthält. Die Bedingung in Objektcode wird in PROH vorausgesetzt. Dazu sind entsprechende Informationen in FOH.



Der Wert des 1. Parameters von value sei bool, der Wert des 2. Parameters a. Der Wert des Parameters des statement train nach then sei b und der Wert des Parameters von statement train nach else sei g.



Um den Wert des Parameters von elseend zu bestimmen, wird das Parameterwertvermittlungssystem und die Korrespondenzregel 311 benutzt. Damit bekommt man als nächsten Baum



Der Wert des Parameters von conditional wird durch die Korrespondenzregel 111 und das Parameterwertvermittlungssystem bestimmt.

111 a) Der Wert des 1. Parameters von value ist bool; die Parameterwertvermittlung wird fortgesetzt.
b) a wird dem 2. Parameter von COND zugewiesen.
c) b wird dem 3. Parameter von COND zugewiesen.
d) g wird dem 4. Parameter von COND zugewiesen.
e) Aufruf von COND (C; a; b; g):

Schritt 1:

PRO		FOL			
X=X+1	B1	B1	B1	0	b
Y=Y+1	B2	B2	B2	0	g
IH=X.EQ.Y	B3	B3			d

d wird dem 1. Parameter von COND zugewiesen.

Schritt 2:

PRO	
X=X+1	B1
Y=Y+1	B2
IH=X.EQ.Y	B3
IF(IH) A1 A2	B4

FOL			
B1	B1	0	b
B2	B2	0	g
B3	B4		d

Schritt 3:

PRO	
A1 X = X+1	B1
Y = Y+1	B2
IH =X.EQ.Y	B3
IF(IH)A1, A2	B4

FOL			
B1	B1	0	b
B2	B2	0	g
B3	B4	b	d

b wird zu e.

Schritt 4: Die Fortsetzungsadresse von e ist 0, Fortsetzung mit Schritt 5.

Schritt 5: g ist nicht 0, Fortsetzung mit Schritt 6.

Schritt 6:

PRO	
A1 X = X+1	B1
A2 Y = Y+1	B2
IH =X.EQ.Y	B3
IF(IH)A1,A2	B4
GOTO A3	B5

FOL			
B1	B1	f	b
B2	B2	0	g
B3	B4	b	d
B5	B5	g	f

g wird zu e.

Schritt 7: Die Fortsetzungsadresse von e ist 0, Fortsetzung mit Schritt 8.

Schritt 8:

PRO	
A1 X = X+1	B1
A2 Y = Y+1	B2
IH=X.EQ.Y	B3
IF(IH) A1, A2	B4
GOTO 43	B5
A3 CONTINUE	B6

FOL			
B1	B1	f	b
B2	B2	p	g
B3	B4	b	d
B5	B5	g	f
B6	B6	0	p

f) Der Wert des 1. Parameters von COND (D) wird dem Parameter von conditional zugewiesen.

Über d ist der gesamte FORTRAN 63-Programmteil, der der Objektcode zur ursprünglichen bedingten Anweisung in ASPLE ist, erreichbar. Die Reihenfolge der einzelnen Anweisungen ist durch die Verkettung der Elemente in FOL bestimmt.
(Verkettung in FOL: d b f g p; dem entspricht die Anweisungsfolge

```
      IH=X.EQ.Y
      IF(IH) A1, A2
   A1 X=X+1
      GOTO A3
   A2 Y=Y+1
   A3 CONTINUE       )
```

3. Schlußbemerkungen

Die Beschreibung des Compilers kann nur über einige Eigenschaften informieren. Die Leistungsfähigkeit des Compilers ist bei weitem größer, als aus dem angeführten Beispiel zu ersehen ist. Sie hängt neben der Wahl der Syntaxanalysemethode entscheidend von der Wahl des Parameterwertvermittlungssystems und seiner Realisierung ab. Das hier verwendete System läßt auch die Realisierung von automatischen Datentypanpassungen zu, die die Übermittlung von Parameterwerten in beiden Richtungen im Syntaxbaum benötigen.

Aus den Beispielen war zu ersehen, daß nur die Unterprogramme zur Realisierung der semantischen Funktion von der Objektsprache abhängig sind. Hat man also schon einen Compiler für die Übersetzung in eine Objektsprache, dann bekommt man daraus einen Compiler zur Übersetzung in eine andere Objektsprache durch Austausch der Unterprogramme zur Realisierung der semantischen Funktionen. Die Beschreibung der Quellsprache durch eine Grammatik syntaktischer Funktionen muß allerdings so angelegt sein, daß sie unabhängig von der Objektsprache ist, was sich auch leicht realisieren läßt.

Änderungen der Sprachdefinitionen sind ebenfalls möglich, ohne daß immer dadurch der gesamte Compiler geändert werden muß.

Nimmt man z.B. zusätzliche syntaktische Hilfsfunktionen in die Grammatikbeschreibung auf, so wird dadurch lediglich die Korrespondenzmenge beeinflußt. Bei einer Änderung der syntaktischen Struktur können die Änderungen natürlich umfangreicher sein.

Literaturverzeichnis

/Rie 751/ Riedewald, G. Grammatik syntaktischer Funktionen - eine praxisbezogene Grammatikform Rechentechnik und Datenverarbeitung, 2. Beiheft (1975)

/Rie 752/ Riedewald, G. Die Grammatik syntaktischer Funktionen - eine andere Form der van Wijngaarden Grammatik EIK 11 (1975) 7/8, 479 - 487

/Rie 761/ Riedewald, G. Zweistufenform der syntaktischen Analyse und Objektcodegenerierung erscheint in Rechentechnik und Datenverarbeitung

/Rie 762/ Riedewald, G. Gramatiky syntaktických funkcí a jejich používání v kompilátorech erscheint in acta polytechnica, ČVUT Praha

/Rie 763/ Riedewald, G. Compilermodell auf der Grundlage einer Grammatik syntaktischer Funktionen Bericht

/U 73/ Uzgalis, R. What every programmer should know about grammars Manuskript

eingegangen: 23. 4. 1976

Anschrift des Verfassers

Dr. rer.nat. Günter Riedewald, Wilhelm-Pieck-Universität Rostock
Sektion Mathematik, DDR 25 Rostock, Universitätsplatz 1

Hana Loeper, Wolfgang Ottèr, Harry Jähmlich

Führung der Tabellen und Behandlung von Kontextabhängigkeiten im Lexikalen Anaysator eines Übersetzerprogramms für eine Untersprache von ALGOL 68

geplante Veröffentlichung in:

- Tagungsmaterialien der MKÖ IV
  (Dresden 27. - 29. 10. 1975)
- Beiheft "Rechentechnik und Datenverarbeitung"

Anschrift der Verfasser

Dr.-Ing. Hans Loeper
HS-Ing. Wolfgang Otter
HS-Ing. Harry Jähmlich
Technische Universität Dresden
Sektion Informationsverarbeitung
DDR 8027 Dresden
Mommsenstr. 13

Hans-Joachim Bartsch, Peter Forbrig
Immo Kerner und Hartmut Radtke

Rechnerunterstützte zeichnerische Darstellung konvexer Polyeder

Zusammenfassung

Der Einsatz der digitalen Rechentechnik zur Bearbeitung geometrischer oder graphischer Problemstellungen, wie sie besonders im Bauwesen, aber auch im chemischen Anlagenbau (Rohrleitungen) und an vielen anderen Stellen auftreten, verlangt ein leicht verwendbares, überall greifbares Lösungssystem. Dies kann aus den weitverbreiteten R 300-EDV-Anlagen und der verfügbaren Programmiersprache ALGOL 60 als Grundlage und einer Zeichenmaschine als notwendige Ergänzung zusammengestellt werden.

Der hier beschriebene Anwendungstest beschränkt sich auf die Darstellung von Geraden, Ebenen und Körpern (Polyeder), die durch Ebenen begrenzt werden. Durch relativ einfach arbeitende Algorithmen konnte das Problem der verdeckten Kanten bei diesen Körpern ökonomisch gelöst werden. Die Grundlage der rechnerinternen Darstellung der geometrischen Gebilde ist ein Modell der mathematischen Auffassung endlicher, gerichteter Graphen als Menge der Knoten (Ecken) und Menge der Kanten.

Die vorhandenen ALGOL-Programme enthalten Prozeduren für einfache geometrische Bewegungen (Drehung, Verschiebung) im Raum und eine Zentralprojektion zum Erhalten eines ebenen Bildes.

Die Bereitstellung und Publikation des Verfahrens entspricht als Initiative zur Vorbereitung des 9. Parteitages voll und ganz dem Anliegen des 13. Plenums des ZK der SED, nämlich durch die EDV - gerechte Lösung von Routinearbeiten der technischen Vorbereitung die Arbeitsproduktivität zu erhöhen und somit auch die Voraussetzungen für die qualitative Verbesserung der Lösungen gewisser Aufgabenklassen aus dem Bauwesen zu schaffen.

1. Darstellung eines Graphen

Mathematisch besteht ein Graph aus einer Menge von Knoten und einer Menge von Kanten. Die Endpunkte der Kanten gehören zur Menge der Knoten, so daß Kanten in Knoten zusammenstoßen. Für die Bezeichnung Knoten sind die Bezeichnungen Punkt und Ecke auch üblich. Man kann sich einen Graphen als Drahtgitter vorstellen. Ein Graph aus den Kanten und Ecken eines Quaders, d.h. eines Körpers, der im Bauwesen eine besondere Rolle spielt, würde also wie folgt dargestellt werden können (Abb. 1):

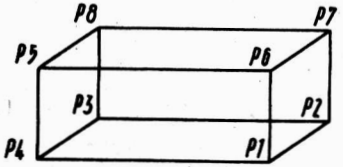


Abb. 1 Quader als Graph

Menge der 8 Knoten (P1, P2, P3, P4, P5, P6, P7, P8)
Menge der 12 Kanten (K1 (P1, P2), K2(P1,P4), K3 (P1,P6),
K4 (P2,P3), K5 (P2,P7), K6 (P4,P3),
K7 (P4,P5), K8 (P6,P5), K9 (P6,P7),
K10 (P3,P8), K11 (P5,P8), K12 (P7,P8))

An ein späteres ökonomisch ablaufendes Zeichnen des Graphen denkend kann man Kanten einen Richtungssinn beiordnen, in dem der Zeichenstift sie durchlaufen soll (Abb. 2):

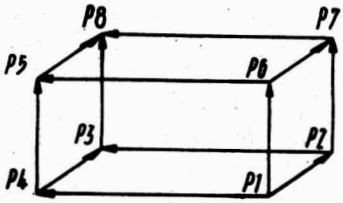


Abb. 2 Quader als gerichteter Graph

Wählt man die Darsellung der Abb. 2, so erhält man einen Anfangspunkt P1, von dem Kanten nur ausgehen, und einen Endpunkt P8, an dem Kanten nur eintreffen. In der oben angegebenen Kantenmenge wurde der Richtungssinn einer Kante bereits durch die Reihenfolge der Knoten in den Paaren (Pi, Pj) berücksichtigt. Den Knoten wird eine Information über die von ihnen ausgehenden Kanten mitgegeben. Dies erfolgt durch Angabe der festgelegten Kantennummern. Da diese Menge als geordnet angesetzt werden kann, genügen die Nummern des Anfangs und des Endes (i:j) "von i bis j" für "von Ki bis Kj". Die Knotenmenge für das obige Beispiel ist somit

(P1(1:3), P2(4:5), P3(10:10), P4(6:7), P5(11:11),
P6(8:9), P7(12:12), P8(0:-1)).

Lautet eine "von : bis" - Angabe (0:-1), so gehen von diesem Knoten keine Kanten aus.

Man kann viele andere Modelle erdenken, die unterschiedliche Vorteile haben. Bekannt ist das unmittelbar eine Zeichnung unterstützende Modell (1) der Operationen "verbinden" (-) und "anreihen" (·). Die erste Operation verlangt das Zeichnen einer Verbindungslinie und die zweite lediglich das Positionieren des Zeichenstiftes.

Der Quader aus dem Beispiel wäre danach anschaulich

P1 - P2 - P3 - P4 - P1 - P6 - P7 - P8 - P5 - P6 ·

P5 - P4 · P2 - P7 · P3 - P8.

(Es gibt auch andere Reiehnfolgen.) Das Positionieren bedeutet Scheinwege bzw. Scheinkanten.
Im vorliegenden Modell, das mit Hilfe von ALGOL 60 realisiert wurde, werden für einen Graphen aus n Knoten und m Kanten im dreidimensionalen Raum benötigt:

Eine Matrix P [1:3, 1:n] für die Koordinaten x, y, z;
eine Matrix E [1:2, 1:n] für die Knotenmenge (Ecken) mit den Angaben "von : bis";
eine Matrix K [1:2, 1:m] für die Kantenmenge mit den Nummern

der Anfangs- und Endknoten.
Die Matrizen E und K geben die Struktur des Graphen an. Geometrische Operationen (Bewegung, Verschiebung, Projektion) können auf P angewendet werden.
Die Struktur bleibt dabei unverändert.

2. Verbindung zum Zeichengerät

Die Ausgabe der ALGOL-Programme erfolgt mit Teilprogrammen, die an die (technischen) Anschlußbedingungen angepaßt sein müssen. Als Ausgabegeräte kommen Zeichenmaschinen oder Bildschirmgeräte in Frage; die quasigraphische Ausgabe mittels Schnelldrucker kann nur als Notbehelf betrachtet werden. - Den Verfassern stand eine Zeichenmaschine Typ ARISTOMAT zur Verfügung. Sie wird von einem 8-kanaligen Lochstreifen (gestanzt im ESSI-Code) gesteuert. Ein Wort pro Zeile bedeutet dabei "Hilfsfunktion", zwei Worte "Gerade", drei "Parabel", vier "Kreis".
Für den Testfall werden nur die beiden ersten Steuersatz-Typen benötigt. Das Stanzen der zugehörigen Lochstreifeninformationen erfolgt in für diese Anwendung hergestellten Codeprozeduren. Zu "Hilfsfunktion" gehören Steuerbefehle wie "Stift heben", "Stift senken", "Strichart wechseln" u.a.m., hier durch Prozeduren wie AUF, AB realisiert. "Gerade" veranlaßt die Zeichenmaschine zum Transport des Zeichenstiftes längs einer Geraden vom gegenwärtigen Standort zum Zielpunkt, dessen Koordinaten relativ zum alten Standort als Parameter anzugeben sind. Die relativen Koordinaten lassen sich als Abstände in mm-Angaben vorstellen; an der Zeichenmaschine selbst kann mit einem Umschalter der Maßstab noch beeinflußt werden. Beim Zeichnen einer geschlossenen Figur fährt der Stift "automatisch" wieder zum ersten Startpunkt zurück; bei anderen Figuren empfiehlt es sich, die Teilstrecken zu summieren, um dann in einem Zuge zum ersten Startpunkt zurückzufinden (die Rückkehrgenauigkeit der Zeichenmaschine war so hoch, daß in keiner der durchgeführten Zeichnungen Abweichungen festgestellt werden konnten).
Die Reihenfolge der Stiftbewegungen ist ökonomisch auszuwählen, damit Leerfahrten weitgehend vermieden werden.

Die Codeprozeduren-Benutzung wird am Beispiel "Rechteck mit Diagonalen" demonstriert.

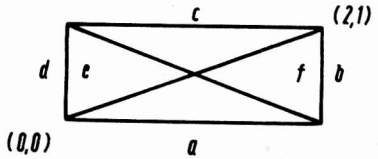


Abb.3 Rechteck mit Diagonalen

Mit der Annahme: Stift befindet sich am Startpunkt (linke untere Ecke) wird begonnen.

AB	Stift senkt sich aufs Papier
gerade (2,0)	Stift zeichnet a
gerade (0,1)	Stift zeichnet b
gerade (-2,0)	Stift zeichnet c
gerade (0,-1)	Stift zeichnet d
gerade (2,1)	Stift zeichnet e
AUF	Stift aufheben
gerade (0,-1)	Leerfahrt längs b
AB	Stift absenken
gerade (-2,1)	Stift zeichnet f
AUF	Stift aufheben
AUS	Maschine ausschalten

Mit einer zusätzlichen Leerfahrt könnte zum Ausgangspunkt zurückgekehrt werden (hier wären Leerfahrten nicht unbedingt nötig, wenn man Mehrfachzeichnung zuläßt).

3. Geometrische Bewegungen

Die Programme für die geometrischen räumlichen Bewegungen und auch für die Projektion beinhalten lediglich Elementargeometrie und bieten auch programmierungstechnisch kaum Probleme. Es sind ausgearbeitet worden

Verschiebung (Translation)
Drehung (Rotation) und
Zentralprojektion.

Da die Drehung Kugelkoordinaten verwendet, muß dabei auf die Reihenfolge der Bewegungen in "Länge" und "Breite" geachtet werden, wenn man einen gewünschten Effekt erzielen will.

Bei der Zentralprojektion erfordert die Angabe des Zentrums und der Projektionsebene einige Erfahrung, um gefällige Bilder zu erhalten. Die Projektion liefert ein ebenes Bild und kann mit der Zeichenprozedur ZEICHNUNG durch Erzeugung eines Steuerstreifens für die Zeichenmaschine im off-line Betrieb gezeichnet werden.

4. Graph-Modell und Zeichenprozedur

Obwohl das Graph-Modell nicht spezifisch zum Zeichnen ausgelegt wurde, kommt es doch diesem recht entgegen, so daß der Stift relativ wenig Leerfahrten durchführt.

Die Prozedur arbeitet die beiden Strukturmatrizen durch, wobei E die Steuerfunktion innehat. Die dort angegebenen "untergeordneten" Kanten werden der Reihe nach gezeichnet. Die Koordinaten werden der Matrix der Projektion PRO entnommen.
Zeichenprozedur:

```
procedure  ZEICHNUNG  (PRO, E, K, n, SX, SY);
   array PRO, E; real SX, SY;
   integer array K; integer n;
   begin integer i, j, DX, DY, x, y, PX, PY;
      DX := DY := 0;
   comment DX, DY sammeln die Teilwege des Zeichenstiftes;
      PX := SX; PY := SY;
   comment PX, PY sind die momentanen Stiftkoordinaten;
   AUF;
   comment Prozedur zum Heben des Zeichenstiftes;
   for i := 1 step 1 until n do
   begin integer E1, E2, KJ;
```

```
        x := PRO [1, i] - PX ;
        y := PRO [2, i] - PY ;
       DX := DX + x ;
       DY := DY + y ;.
       GERADE (x, y);
       comment Stift - Bewegung zum i-ten Knoten;
       PX := PRO [1, i] ; PY := PRO [2, i];
       E1 :=   E [1, i] ; E2 :=   E [2, i];
       for j := E1 step 1 until E2 do
           begin      KJ := K [1, j];
                       x := PRO [1, KJ] - PX;
                       y := PRO [2, KJ] - PY;
                  AB; GERADE  ( x,  y);
                  comment Zeichnen der Kante von Knoten
                                K i zu Knoten KJ;
                  AUF; GERADE (-x, -y);
           end  Alle Kanten des i-ten Knotens gezeichnet
       end Alle Kanten gezeichnet;
       SX := SX + DX; SY := SY + DY;
   end Prozedur;
```

Die angegebene Zeichenprozedur wurde im Rahmen eines größeren ALGOL-Programmes getestet. Sie beinhaltet keine globalen Grössen. Alle notwendigen Angaben werden in Form von Parametern vermittelt. Von diesen Parametern sind die Matrizen E und K bereits aus Abschnitt 1 bekannt.

Die Matrix PRO enthält die durch Zentralprojektion auf eine Ebene aus den Raumkoordinaten der in der Matrix P enthaltenen Punkte gewonnenen Ebenenkoordinaten. Durch die Variablen SX und SY werden die Koordinaten des Zeichenstiftes der Zeichenmaschine bei Aufruf der Prozedur ZEICHNUNG übermittelt. Die Anzahl der Knoten des Graphen ist durch den Parameter n festgehalten.

Jede Bewegung des Zeichenstiftes wird durch Abspeicherung seiner Koordinaten festgehalten. Zur einfacheren Beschreibung sei die Verbindungslinie zwischen zwei Knoten kurz als "Teilweg" bezeichnet. Diese Teilwege werden durch die Variablen x und y festgehalten. Die Summe der Teilwege als Differenz des wirklichen Standes des Zeichenstiftes von den in SX und SY gespeicherten Werten wird durch die Variablen DX und Y gemerkt. Diese werden am Anfang der Abarbeitung auf Null gesetzt. Danach werden die Stiftkoordinaten in PX und PY umgespeichert. Nach diesen Befehlen beginnt der eigentliche Zeichenzyklus. Der Zeichenstift wird vom Papier angehoben und zum 1. Knoten geführt. Die x- und y-Komponente dieses Weges werden zu den Variablen DX und DY addiert. Aus der Matrix E wird entnommen, zu welchen Knoten eine Verbindungslinie zu zeichnen ist.

Im Zyklus wird dann der Zeichenstift abgesetzt, die jeweilige Kante gezeichnet, der Zeichenstift angehoben und zum Ausgangsknoten zurückgeführt.
Dieses Verfahren wird dann auf alle weiteren Knoten angewendet. Ist dies geschehen und alle Linien sind gezeichnet, so wird die Summe der Teilwege zu den anfänglichen Stiftkoordinaten SX und SY addiert. Damit enthalten diese Variablen nach Abarbeitung der Prozedur wieder die momentanten Stiftkoordinaten und es könnte zur Zeichnung eines weiteren Graphen übergegangen werden.

Zum Test wurde eine "Wendeltreppe" gezeichnet. Die Zeichenmaschine benötigte dazu 18 Minuten reine Zeichenzeit. Das ALGOL-Programm lief in einem Stapel mit anderen Programmen, wodurch eine exakte Rechenzeitangabe nicht möglich ist; der Operator hat 0,2 Std. eingetragen.

Die Abb. 4 zeigt die Wendeltreppe, bei deren Stufen alle Kanten sichtbar sind.

Beim Betrachten ist die richtige Wahl der Augenhöhe zu beachten.

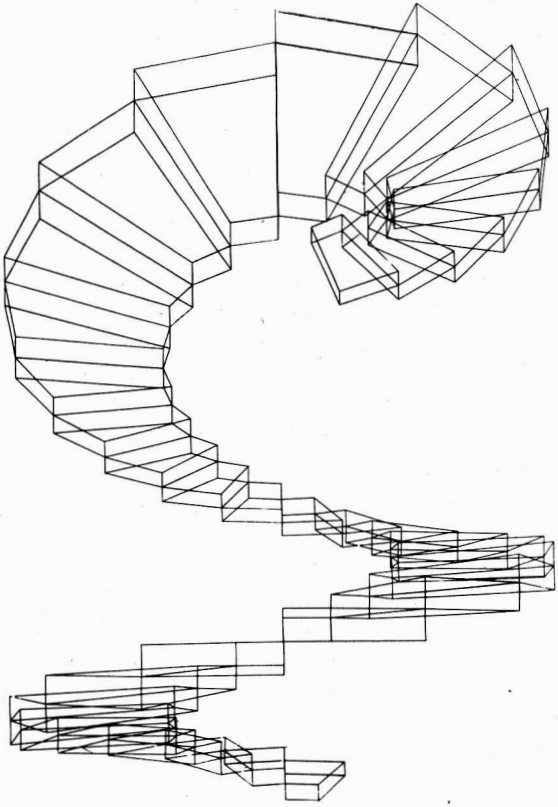


Abb. 4 Wendeltreppe, alle Stufenkanten sichtbar

5. Verdeckungsproblem

Zeichnungen aus architektonischen Anwendungen (Baukörper, Projektierung ganzer Stadtteile) gewinnen natürlich an unmittelbarer Anschaulichkeit, wenn verdeckte Linien nicht mit gezeichnet werden. Das ist in voller Allgemeinheit ein durchaus kompliziertes Problem und Programme, die das allgemeine Problem

lösen, arbeiten dann auch entsprechend aufwendig (2).

Für einfachere Aufgabenstellungen kann man auch ökonomischere Lösungen angeben. So kann man sich zunächst auf "Selbstverdeckung" beschränken, d.h. auf Aussonderung der "hinten" liegenden Kanten, und die "Fremdverdeckung" durch andere Körper ausklammern. Dann können beispielsweise isoliert stehende Häuser schon korrekt dargestellt werden (Haus = Einzelquader). Eine andere Einschränkung betrifft die Körperauswahl. Der Quader ermöglicht sehr einfache Lösungsansätze und gerade er spielt aus bautechnologischen Gründen eine große Rolle im Bauwesen. Bei näherer Untersuchung zeigt sich die Eigenschaft "konvex" des Quaders als die Ursache der einfachen Lösungen. Das Verfahren konnte als Folge davon auf beliebige "konvexe durch ebene Flächenstücke begrenzte" Körper (Polyeder) ausgedehnt werden, ohne den Aufwand wesentlich zu steigern.

Bei den angestellten Versuchen war es interessant zu sehen, daß angesetzte Näherungen in den meisten Fällen (90 % und mehr) das Problem voll lösen. Beispielsweise ist für einen Quader bei Zentralprojektion stets der leicht austestbare hinterste Punkt (d.h. der vom Zentrum entfernteste) unsichtbar und die von ihm ausgehenden Kanten sind es auch (siehe P3 in Abb. 2). Meist sind genau diese drei Kanten unsichtbar. Es gibt einige spezielle Lagen, wo dies aber nicht gilt (Abb. 5).

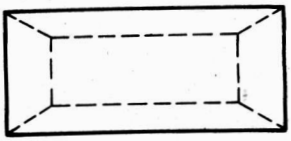


Abb. 5 Quader mit acht verdeckten Kanten

Obwohl die Wendeltreppenstufen keine Quader sind, wurde auch versucht, die genannte einfache Fragestellung als Näherung anzuwenden. Nur bei zwei Stufen zeigten sich fehlerhafte Ergebnisse der Selbstverdeckung. Das Beseitigen dieser an sich er-

warteten Fehler brachte die Erweiterung vom Quader auf beliebige konvexe Polyeder.

Die Grundidee liegt darin, daß auch die Projektion eines konvexen Polyeders konvex ist. Die Kontur, ein konvexes Polygon, erhält man aus der Frage: Liegen alle projizierten Punkte auf derselben Seite einer Konturstrecke? Die Selbstverdeckung wird dann durch Anwendung der Erkenntnis gelöst, daß alles sichtbar ist, was vor der Kontur liegt. Die für den Quader und auch andere spezielle konvexe Körper richtige Aussage - Der hinterste Punkt ist unsichtbar! - ist nicht richtig für beliebige konvexe Polyeder.

Die Abb. 6 zeigt das Testbeispiel "Wendeltreppe", wobei alle selbstverdeckten Kanten einer Stufe ausgelassen sind. Es wird deutlich, daß diese Vereinfachung des Verdeckungsproblems gegenüber der Abb. 4 bereits viel mehr Anschaulichkeit in die Zeichnung bringt, und daß bereits größere Teile ganz und gar korrekt sind.

Die Zeichenzeit für Bild 6 sank auf 12 Minuten, da der Zeichenalgorithmus unsichtbare Kanten einfach ausläßt. Es wurde für die Rechnung ein Sichtbarkeitsvektor Boolean array S [1:m] für die Kanten eingeführt, dessen logische Komponenten die Sichtbarkeitsinformation liefern. Der vergrößerte Rechenaufwand zum Feststellen der Sichtbarkeitsverhältnisse geht offenbar in der Stanzzeit für den Lochstreifen unter, denn es wurden wieder 0,2 Std. ausgewiesen.

Das erweiterte oder eigentliche Verdeckungsproblem zweier oder mehrerer Körper untereinander wurde auf einen Algorithmus beschränkt, der nur zwei Körper berücksichtigt. Es wird bei seiner Anwendung vom Programmierer verlangt, das Vorhandensein mehrerer Körper im Programm zu beachten. Kompliziertere Verhältnisse bedingen somit ein komplizierteres Anwenderprogramm - aber einfache Probleme werden auch einfach, d.h. ökonomisch und ohne für diese Fälle überladenen Apparat, bearbeitet.

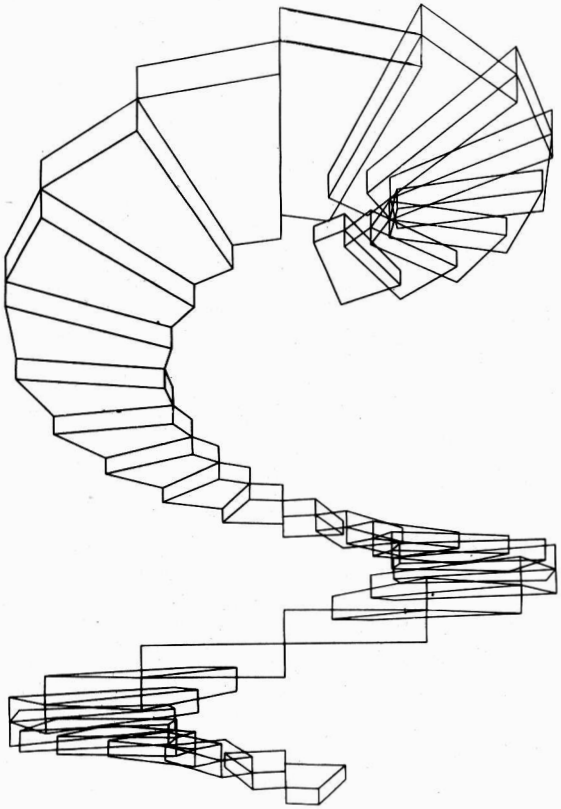


Abb. 6 **Wendeltreppe, selbstverdeckte Kanten der Stufen nicht gezeichnet**

Die Grundidee des Verdeckungsalgorithmus bei Fremdverdeckung läuft auf die Beantwortung folgender Fragen oder Teste hinaus:

1. Liegt Verdeckung vor?
   Dies ist der Fall, wenn die Projektionskonturen sich überschneiden oder sich etwa umfassen und dies erkennt man, wenn die projizierten Knoten des einen Graphen teils inner-

halb der Kontur des anderen Graphen liegen. Es kann hierbei eine Prozedur verwendet werden, die bereits beim Problem der Selbstverdeckung auftrat.

2. Falls teilweise Verdeckung vorliegt, muß bekannt sein, wer was verdeckt. Dies wird durch den Test Vorn - Hinten, d.h. Entfernung zum Projektionszentrum, für die Original-Knotenkoordinaten entschieden.

3. Die Kanten des verdeckten Körpers können gar nicht, teilweise oder völlig sichtbar sein. Es müssen die Schnittpunkte mit der Kontur des verdeckenden Körpers bestimmt werden. Bei konvexen Körpern kann es pro Kante maximal zwei Schnittpunkte geben. Diese Punkte erzeugen höchstens drei Abschnitte auf den Kanten. Der bei der Selbstverdeckung benötigte Sichtbarkeitvektor S muß durch eine Sichtbarkeitsmatrix Boolean array S [1:3, 1:m] und eine Schnittpunktmatrix array SCH [1:4, 1:m] ergänzt werden. In SCH bedeuten die vier Zahlenwerte x- und y-Koordinaten der beiden möglichen projizierten Schnittpunkte.

4. Die Zeichenprozedur hat die nun vorliegenden komplizierteren Verhältnisse zu beachten.

Das Problem der Fremdverdeckung wurde, wie deutlich ist, wieder so angelegt, daß einfache Körper (wenig Kanten) weniger Speicherbedarf und weniger Rechenzeit benötigen und daß kompliziertere Körper einen größeren Aufwand bedingen. Das schlägt sich dann auch in den Rechen- und Zeichenkosten nieder. Eine obere Schranke für die Zeichenkosten ist dabei meist das Zeichnen der Figur mit allen Kanten (entspricht Abb. 4), denn die Befehle für zeitweiliges Abheben des Zeichenstiftes beim Durchfahren einer Kante werden durch die Einsparung ganzer Kanten bei Vollverdeckung kompensiert.

Im Anwendungstest "Wendeltreppe" wurden stets nur zwei benachbarte Stufen dem Verdeckungsprogramm als Parameter gegeben. Wie bereits in Abb. 6 deutlich wird, gibt es auch für nicht

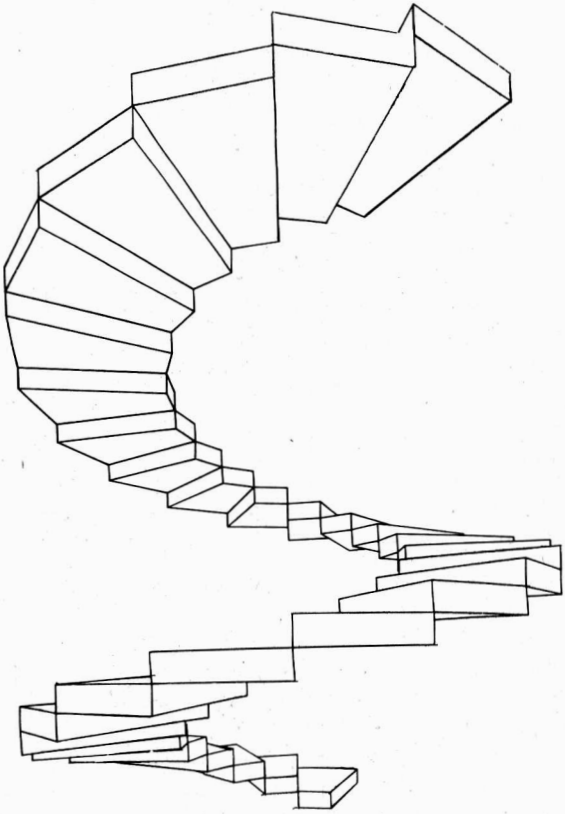


Abb. 7 Wendeltreppe unter voller Beachtung der Sichtbarkeitsverhältnisse

benachbarte Stufen Verdeckungsfälle. Diese wurden durch den Test nicht erfaßt und deshalb wurde das Testbild um vier Stufen verkürzt. Auf eine Verkürzung der Bilder 4 und 6 wurde im Interesse einer korrekten Darstellung des Testes verzichtet. Der erhöhte Rechenaufwand wurde durch den Operator auf 0,4 Std. geschätzt.

Das Verfahren wurde auch auf einen Polyeder angewendet, der eine Kugel mit Längen- und Breitenkreisen approximiert. Die Abb. 8 zeigt diese Kugel mit und ohne verdeckte Kanten in gedrehter Lage.

Das Auslassen der verdeckten Kanten erhöht im Bild stark die Anschaulichkeit.

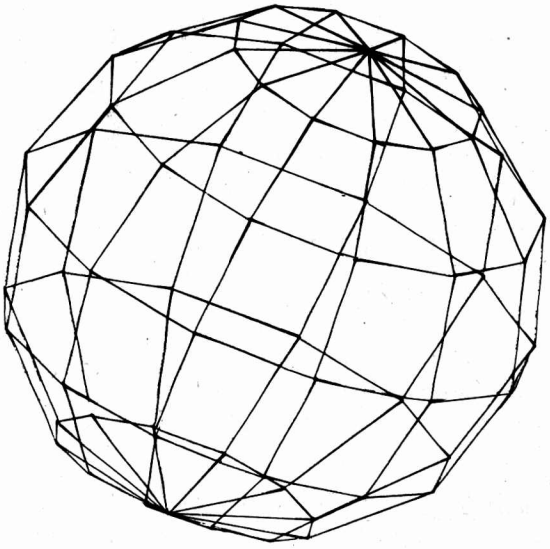


Abb. 8a Kugelapproximation
a) gedreht mit allen Kanten

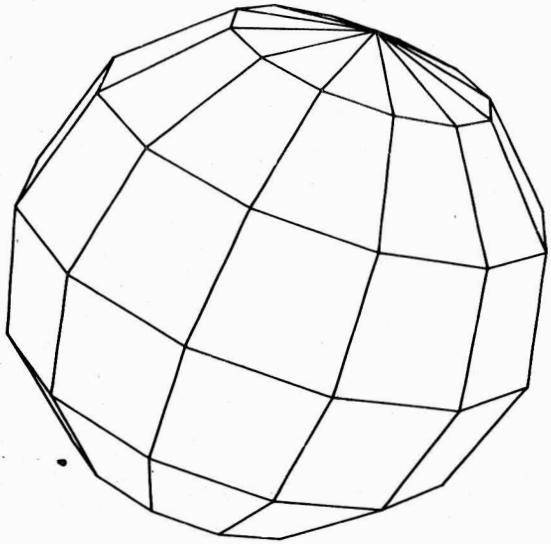


Abb. 8b Kugelapproximation
b) gedreht ohne verdeckte Kanten

6. Schlußbemerkungen

Das verwendete Graph - Modell für Körper bedingt beim Verdeckungsproblem Schwierigkeiten für den Algorithmus, denn ein Graph ist nur ein "Drahtgerüst". Der Körper entsteht lediglich durch (irgendeine) Interpretation, die im vorliegenden Fall - und auch sonst natürlich - in das Programm bzw. in den Algorithmus verlegt werden mußte. Grundlage der Interpretation ist hier die Eigenschaft "konvex".

Abgesehen von dieser Problematik, bildeten für die Überlegungen nur elementargeometrische Kenntnisse die Grundlage.

Alle Programme wurden denn auch nach einer vorgelegten ersten Fassung von Studenten des zweiten Studienjahres fehlerfrei gemacht und getestet.

Literaturverzeichnis

/1/ Anwender - Information 18/71
Programmsystem DIGRA
Institut für Schiffbau, Rechenzentrum, Rostock, Juni 1971

/2/ Marin, H., Thielke, H.
Digitalgrafische Algorithmen zur Darstellung von räumlichen Gebilden unter Berücksichtigung der Sichtbarkeitsverhältnisse
Dissertation Universität Rostock, Sektion Mathematik, 1972

eingegangen: 23. 4. 1976

Anschrift der Verfasser

Dr.-Ing. H.-J. Bartsch
Wilhelm-Pieck-Universität
Rechenzentrum
DDR 25 Rostock
Albert-Einstein-Straße 21

P. Forbrig, Doz. Dr.sc.nat. I.O. Kerner, H. Radtke
Wilhelm-Pieck-Universität
Sektion Mathematik
DDR 25 Rostock
Universitätsplatz 1

Hans-Peter Lorenzen

Programmbeispiel ALGOL 68 - Lösung einer aussagenlogischen Aufgabe mit Hilfe der Programmiersprache ALGOL 68

Die universelle Programmiersprache ALGOL 68 (s. /1/ u. /2/) läßt sich zur Lösung der vielfältigsten Aufgaben aus der numerischen wie auch aus der nicht numerischen Datenverarbeitung einsetzen. Im folgenden wird eine Aufgabe aus der Aussagenlogik und ihre Lösung mit Hilfe von ALGOL 68 vorgestellt.

Es seien fünfzehn Aussagen gegeben:

Auf dem Campingplatz stehen fünf Zelte.
Der Leipziger wohnt im roten Zelt.
Der Berliner fährt einen Wartburg.
Bier wird im grünen Zelt getrunken.
Der Dresdner trinkt Tee.
Das grüne Zelt steht unmittelbar rechts neben dem grauen Zelt.
Der Campingfreund, der "Jubilar" raucht, fährt Motorrad.
"Carré" werden im gelben Zelt geraucht.
Milch wird im mittleren Zelt getrunken.
Der Erfurter wohnt im ersten Zelt.
Der Mann, der "Orient" raucht, zeltet neben dem Zelt mit dem Mann mit dem Fahrrad.
Vom Bewohner des Zeltes neben dem Zelt mit dem Mopedbesitzer werden "Carré" geraucht.
Der Mann, der Fruchtsaft trinkt, raucht "Diplom".
Der Magdeburger raucht "Dubec".
Das Zelt des Erfurters steht neben dem blauen Zelt.

Mit Hilfe dieser Aussagen ist herauszufinden, wer Kaffee trinkt und wer zu Fuß geht.
Wie man sofort sieht, existieren in diesen Aussagen sechs Merkmalklassen:

1. Zelte
2. Zeltbewohner
3. Zigaretten

4. Getränke
5. Farben
6. Fahrzeuge

Zu jeder Merkmalsklasse gehören fünf Merkmale und, da zwischen den Merkmalen durch die Aussagen Beziehungen bestehen, ein bis zwei Relationen (s. unten Relationsmatrix). Setzt man die Zeltnummer für jedes Merkmal als gesucht voraus, so lassen sich die Merkmale in einer fünf-mal-fünf-Matrix anordnen. Jede Zeile dieser Matrix entspricht hierbei einer Merkmalsklasse. Um die Relationen untereinander ausdrücken zu können, wird jedem Merkmal eine Variable (bzw. in drei Fällen die Zeltnummer selbst) zugeordnet. Diese Variable liefert im Falle einer Lösung der Aufgabe die Zeltnummer des entsprechenden Merkmals. Innerhalb einer Zeile werden nun die Merkmale einer Klasse willkürlich angeordnet und entsprechend den bestehenden Relationen mit Variablen versehen.
Hieraus ergibt sich z.B. folgende Relationsmatrix:

Erfurter	Leipziger	Berliner	Dresdner	Magdeburger
1	V1	V2	V3	V4
Jubilar	Carré	Orient	Diplom	Dubec
V5	V6	V7	V8	V4
Bier	Tee	Milch	Fruchtsaft	Kaffee
V9	V3	3	V8	V10
rot	blau	grün	grau	gelb
V1	2	V9	V9-1	V6
Wartburg	Motorrad	Fahrrad	Moped	zu Fuß
V2	V5	V7±1	V6±1	V11

Die elf Variablen V1 bis V11 müssen jetzt für die Werte 1 bis 5 so variiert werden, daß die Zeltnummern in jeder Zeile paarweise und auch von den fest vorgegebenen Zeltnummern verschieden sind. Kann diese Bedingung für keine Belegung der Variablen eingehalten werden, existiert keine Lösung der Aufgabe. Aus Effektivitätsgründen werden die Variablen V 1 bis V 4 und

V5 bis V8 permutiert. Dadurch wird für die ersten beiden Zeilen immer die geforderte Bedingung eingehalten. Die Variable V9 kann nur die Werte 4 oder 5 annehmen, daraus ergibt sich aus der Zeile 3 sofort ein Wert für V 10. Die Variable V 11 wird für die Werte 1 bis 5 variiert.
Die Relationsmatrix läßt sich mit Hilfe von ALGOL 68 folgendermaßen aufbauen:

```
mode Merkmal = struct (string Name,
                       flex [1 : 0] proc void Relation,
                       int Zeltnummer);
[1 : 5, 1 : 5] Merkmal Matrix;
```

Jedes Merkmal wird durch eine Struktur dargestellt, in der der Name des Merkmals als Zeichenkette enthalten ist, außerdem ein Relationsvektor, dessen Elemente Prozeduren sind, und ein Feld, das die Zeltnummer des Merkmals angibt. Diese Zeltnummer wird geliefert, wenn ein Element des Relationsvektors abgearbeitet wird, d.h. eine Prozedur aufgerufen wird, die eine entsprechende Relation für dieses Merkmal realisiert. Da die Anzahl der Relationen für jedes Merkmal unterschiedlich ist, sind die Grenzen des Relationsvektors variabel gehalten. In ALGOL 68 werden auch Prozeduren als Werte behandelt. Sie können daher z.B. als Komponenten in Strukturen oder als Elemente von mehrfachen Werten (arrays) auftreten oder aber auch als Parameter in eine Prozedur übermittelt oder als Wert eines Prozeduraufrufes geliefert werden. In vielen anderen Sprachen, z.B. in PL1, existiert diese Möglichkeit nicht.

Das Permutationsprogramm, das in Form einer Prozedur definiert wurde (s.unten), permutiert n Elemente nach einer lexiographischen Ordnung. Sind $a_1\ a_2 \dots a_n$ die Permutationselemente mit $a_1 < a_2 < \dots < a_n$, so sind die Permutationen

$a_1\ a_2 \dots a_n$

$a_1\ a_2 \dots a_n\ a_{n-1}$

$a_1\ a_2 \dots a_{n-1} a_{n-2}\ a_n$

$a_1\ a_2 \cdots a_{n-1}\ a_n\ a_{n-2}$

$a_1\ a_2 \cdots a_n\ a_{n-2}\ a_{n-1}$

$a_1\ a_2 \cdots a_n\ a_{n-1}\ a_{n-2}$

usw. bis

$a_n\ a_{n-1} \cdots a_2\ a_1$

Die Permutationselemente werden der Prozedur als Vektor übermittelt. Er enthält nach der Abarbeitung die nächste Permutation. Außerdem liefert die Prozedur den Wert true, falls noch eine Permutation erfolgte, anderenfalls den Wert false. Hiermit ergibt sich das folgende ALGOL 68-Programm zur Lösung der gestellten Aufgabe:

```
begin co Ausgabe der Aufgabenstellung co
      print(ßp10x"Aufgabenstellung:"lß);
      print(ß10x"Auf␣dem␣Campingplatz␣stehen␣5␣Zelte."lß);
      print(ß10x"Der␣Leipziger␣wohnt␣im␣roten␣Zelt."lß);
      print(ß10x"Der␣Berliner␣fährt␣einen␣Wartburg."lß);
      print(ß10x"Bier␣wird␣im␣grünen␣Zelt␣getrunken."lß);
      print(ß10x"Der␣Dresdner␣trinkt␣Tee."lß);
      print(ß10x"Das␣grüne␣Zelt␣steht␣unmittelbar␣rechts␣
           neben␣dem␣grauen␣Zelt."lß);
      print(ß10x"Der␣Campingfreund,␣der␣""Jubilar""␣raucht,␣
           fährt␣Motorrad."lß);
      print(ß10x"""Carre""␣werden␣im␣gelben␣Zelt␣geraucht."lß);
      print(ß10x"Milch␣wird␣im␣mittleren␣Zelt␣getrunken."lß);
      print(ß10x"Der␣Erfurter␣wohnt␣im␣ersten␣Zelt."lß);
      print(ß10x"Der␣Mann,␣der␣""Orient""␣raucht,␣zeltet␣
           neben␣dem␣Zelt␣mit␣dem␣Mann␣mit␣dem␣Fahrrad."lß);
      print(ß10x"Vom␣Bewohner␣des␣Zeltes␣neben␣dem␣Zelt␣mit␣
           dem␣Mopedbesitzer␣werden␣""Carre""␣geraucht."lß);
      print(ß10x"Der␣Mann,␣der␣Fruchtsaft␣trinkt,␣raucht␣
           ""Diplom""."lß);
      print(ß10x"Der␣Magdeburger␣raucht␣""Dubec""."lß);
      print(ß10x"Das␣Zelt␣des␣Erfurters␣steht␣neben␣dem␣
           blauen␣Zelt."lß);
```

```
print(ø10x"Wer.geht.zu.Fuß?"1ø);
print(ø10x"Wer.trinkt.Kaffee?"21ø);

co Deklaration der Relationsmatrix co
mode Merkmal = struct(string Name,
                      flex /1:0/ proc void Relation,
[1:5, 1:5] Merkmal Matrix;          int Zeltnummer);
 [1:11] int V;
ref string s; int i, j;

co Deklaration der Permutationsprozedur co
proc Permutation = (ref [] int a)bool:

begin co Nächste Permutation der Elemente von a nach der
         lexiographischen Ordnung. Erfolgte noch eine
         Permutation, so wird true geliefert. co
      proc next = (ref int a, ref [] int b) int:

      begin co Für das Element a wird aus dem Vektor b
               ein Nachfolger bestimmt co
            int j:= lwb b; bool t:=false;
            for i from j to upb b
            do if b [i] > a and b [i] < b [j]
               then t:=true; j:=i fi
            od;
            if t then j else 0 fi
      end;
      proc ordne = (ref / / int a)void:

      begin co Der Vektor a wird nach der lexiographischen
               Ordnung geordnet. co
            int ug= lwb a, og= upb a;
            if ug=og then goto ende fi;
            ref int m,p,j; int k;
            for i from ug to og-1
            do j:=a [i]; m:=j;
               for n from i+1 to og
               do p:=a [n]; if p< j then j:=p fi od;
               k:=m; ref int(m):=j; ref int(j):=k
            od;
```

```
ende: skip
end;
int i,k,n; int ug= lwb a, og= upb a;
if ug=og then false else goto M fi exit
M : i : = 0;
    for j from ug to og-1 while i=0
    do k:=og+ug-1-j; i:=next(a[k], a[k+1:og]) od;
    if i=0 then false
    else n:=a[k]; a[k]:=a[i]; a[i]:=n;
         ordne(a[k+1:og]); true fi
end co Permutation co;
proc Zeilentest = (ref [ ] Merkmal Zeile, int i) bool:
co Es wird für jedes Element der angegebenen Zeile der
   Relationsmatrix eine Relation abgearbeitet.
   Anschließend wird überprüft, ob alle Zeltnummern die-
   ser Merkmale paarweise verschieden sind. Ist das der
   Fall, wird true geliefert, anderenfalls werden evtl.
   weitere Relationen abgearbeitet und der Test wiederholt.
   co
begin for k to upb Relation of Zeile [i]
      do (Relation of Zeile [i]) [k];
         if i≠lwb Zeile
         then if Zeilentest (Zeile, i-1) then goto MM fi
         else for j from lwb Zeile to upb Zeile-1
              do for n from j+1 to upb Zeile
                 do if Zeltnummer of Zeile [j]
                       Zeltnummer of Zeile [n] then goto M
                    fi
                 od
              od;
              goto MM;
           M:skip
         fi
      od;
      false exit
  MM:true
```

```
end co Zeilentest co;
co Herstellen der Relationsmatrix co
Matrix [1,1]:= ("Erfurter", void:skip, 1);
Matrix [1,2]:= ("Leipziger",
                void:Zeltnummer of Matrix[1,2]:= V[1],skip);
Matrix [1,3]:= ("Berliner",
                void:Zeltnummer of Matrix [1,3]:= V[2],skip);
Matrix [1,4]:= ("Dresdner",
                void:Zeltnummer of Matrix[1,4]:= V[3],skip);
Matrix [1,5]:= ("Magdeburger",
                void:Zeltnummer of Matrix [1,5]:= V[4],skip);
Matrix [2,1]:= ("Jubilar",
                void:Zeltnummer of Matrix[2,1]:= V[5],skip);
Matrix [2,2]:= ("Carre",
                void:Zeltnummer of Matrix[2,2]:= V[6],skip):
Matrix [2,3]:= ("Orient",
                void:Zeltnummer of Matrix[2,3]:= V[7],skip);
Matrix [2,4]:= ("Diplom",
                void:Zeltnummer of Matrix[2,4]:= V[8],skip);
Matrix [2,5]:= ("Dubec",
                void:Zeltnummer of Matrix[2,5]:= V[4],skip);
Matrix [3,1]:= ("Bier",
                void:Zeltnummer of Matrix[3,1]:= V[9],skip);
Matrix [3,2]:= ("Tee",
                void:Zeltnummer of Matrix[3,2]:= V[3],skip);
Matrix [3,3]:= ("Milch", void:skip, 3);
Matrix [3,4]:= ("Fruchtsaft",
                void:Zeltnummer of Matrix[3,4]:= V[8],skip);
Matrix [3,5]:= ("Kaffee",
                void:Zeltnummer of Matrix[3,5]:= V[10],skip);
Matrix [4,1]:= ("rot",
                void:Zeltnummer of Matrix[4,1]:= V[1],skip);
Matrix [4,2]:= ("blau", void:skip, 2)
Matrix [4,3]:= ("grün",
                void:Zeltnummer of Matrix[4,3]:= V[9],skip);
Matrix [4,4]:= ("grau",
                void:Zeltnummer of Matrix[4,4]:=V[9]-1,skip);
```

```
   Matrix [4,5]:=("gelb",
                  void:Zeltnummer of Matrix[4,5]:=V[6],skip);
   Matrix [5,1]:=("Wartburg",
                  void:Zeltnummer of Matrix[5,1]:=V[2],skip);
   Matrix [5,2]:=("Motorrad",
                  void:Zeltnummer of Matrix[5,2]:=V[5],skip);
   Matrix [5,3]:=("Fahrrad",
                  void:Zeltnummer of Matrix[5,3]:=V[7]+1,
                  void:Zeltnummer of Matrix[5,3]:=V[7]-1),skip);
   Matrix [5,4]:=("Moped",
                  void:Zeltnummer of Matrix[5,4]:=V[6]+1,
                  void:Zeltnummer of Matrix[5,4]:=V[6]-1),skip);
   Matrix [5,5]:= ("zu Fuß",
                  void:Zeltnummer of Matrix[5,5]:=V[11],skip);

   co Hier beginnt die eigentliche Rechnung co
   V [1:4]:= (2,3,4,5);
M1:i:=1; j:=5;
   while j ≤ 8
   do if V[4] ≠ i then V/j/:=i; j:=j+1 fi; i:=i+1 od;
M2:V[9]:=4;
   if V[9]=V[1] or V[9]=V[6] or V[9] -1=V[1] or V[9] -1=V[6]
   then V[9]:=5 fi;
   i:=1;
   while (i=3 or i=V[3] or i=V[8] or i=V[9]) and i < 5
   do i:=i+1 od;
   V[10]:=i; V[11]:=1;
M3:for i from 1 upb Matrix by -1 to 1 lwb Matrix
   do if not Zeilentest (Matrix [i,],2 upb Matrix)
      then goto Zyklus fi
   od;
   co Es wurde eine Lösung gefunden co
   print (ß10x"Die␣Lösung␣der␣Aufgabe␣lautet:"1ß);
   print (ß2110x"Zelt1"12x"Zelt2"12x"Zelt3"12x"Zelt4"12x
          "Zelt5"ß);
   for l from 1 lwb Matrix to 1 upb Matrix
```

```
   do printf(ø2110xø);
      for k from 2 lwb Matrix to 2 upb Matrix
      do i:=2 lwb Matrix;
         while j:=Zeltnummer of Matrix [1,i];
         if j=k
         then s:=Name of Matrix [1,i];
              print ((øn(upb s)aø,s,øn(17-upb s)xø));
              false
              else true fi
         do i:=i+1 od
      od
   od exit
Zyklus:V[11]:=V[11]+1;if V[11] ≤ 2 upb Matrix then goto M3 fi;
       if Permutation(V[5:8]) then goto M2 fi;
       if Permutation(V[1:4]) then goto M1 fi;
       print (ø10x"Es.gibt.keine.Lösung.der.Aufgabe."ø)
end co Programm co;
```

Literaturverzeichnis

/1/ van Wijngaarden, A.
Revised Report on the Algorithmic Language ALGOL 68
Acta Informatica
Springer-Verlag Berlin - Heidelberg - New York 1975

/2/ Kerner, I.O.
Revidierter Bericht über die algorithmische Sprache ALGOL 68
Akademie-Verlag Berlin 1977

eingegangen: 17. 5. 1976

Anschrift des Verfassers

Dipl.-Math. Hans-Peter Lorenzen
Wilhelm-Pieck-Universität Rostock
Rechenzentrum
DDR 25 Rostock
Albert-Einstein-Straße 21

Hans-Peter Lorenzen

Semantische Synthese für einen ALGOL 68-Compiler (Kurzfassung)

Um eine Programmiersprache auf einer Rechenanlage benutzen zu können, ist es notwendig, ein Übersetzerprogramm zu schreiben, das diese Programmiersprache in eine der Rechenanlage verständliche Sprache übersetzt. An einem solchen Projekt, einem Compiler (ESER, OS) für die universelle Programmiersprache ALGOL 68, arbeitet zur Zeit die Gruppe "Programmiersprachen" der Wilhelm-Pieck-Universität Rostock.

Der Compiler setzt sich aus mehreren Teilen zusammen, z.B. dem Vorprozessor, der Syntaxanalyse und der semantischen Synthese. Die Syntaxanalyse liefert einen Erzeugungsbaum in komprimierter Form, d.h. es sind in ihm nur noch Begriffe vertreten, für die eine Semantik (s. /1/ u. /2/) definiert ist. Die semantische Synthese, d.h. die semantischen Compilerprozeduren, arbeitet mit diesem Erzeugungsbaum. In ihm sind alle Informationen enthalten, die für die Erzeugung des Objektprogrammes (ESER-Assembler) notwendig sind.

Im Gegensatz zu anderen Implementationen dynamisch organisierter Sprachen, wurde die Arbeit mit den Blöcken erweitert. In ALGOL 68 kann ein Block auch einen Wert liefern (im Gegensatz zu ALGOL 60). Dadurch wird eine Organisation der Blockarbeit z.B. nach /3/ uneffektiv, da jedesmal nach Verlassen eines Blockes, der einen Wert liefert, der Ergebniswert an den Blockanfang gespeichert werden müßte und die Werte in ALGOL 68 recht kompliziert aufgebaut sein können. Nach der vorgestellten Methode werden die Blockzellen nicht beim Verlassen eines Blockes freigegeben, sondern erst in Abhängigkeit vom Kontext. Dadurch wird z.B. eine wesentliche Einsparung bei der Abarbeitung von rekursiven Prozeduren erreicht, bei denen der Ergebniswert unverändert durch alle Niveaus durchgereicht wird. Die evtl. Umspeicherung des Ergebniswertes, z.B. in Zuweisungen, ist damit

von der Verschachtelungstiefe der rekursiven Prozedur unabhängig.

Um auch sehr umfangreiche Programmsysteme mit Hilfe von ALGOL 68 realisieren zu können, ist eine Segmentierungsmöglichkeit unbedingt erforderlich. Es wird ein automatisches Segmentierungsverfahren vorgestellt, das das Prozedurkonzept ausnutzt und ein dynamisches Laden und auch Überlagerung erlaubt. Hierbei erscheinen aber keine Ladebefehle im ALGOL 68-Programm, sondern durch die Übersetzung eines Prozeduraufrufes werden die entsprechenden Befehle erzeugt. Es ist nur notwendig, die Prozeduren zu klassifizieren. Das geschieht durch die Deklaration einer Prozedur im Programm oder im Bibliotheksvorspiel. Außerdem muß im Programm angegeben werden, ob dynamisches Laden oder Überlagerung erfolgen soll. Diese Angabe (dynamisches Laden bzw. Überlagerung) gilt dann für alle ladefähigen Prozeduren in einem Lauf, sie kann aber in einem anderen Lauf geändert werden, ohne daß die Prozeduren geändert werden müssen. Mit dieser Methode ist eine einfache Handhabung umfangreicher Programmsysteme gewährleistet.

Literaturverzeichnis

/1/ van Wijngaarden, A.
Revised Report on the Algorithmic Language ALGOL 68
Acta Informatica
Springer-Verlag Berlin - Heidelberg - New York 1975

/2/ Kerner, I.O.
Revidierter Bericht über die algorithmische Sprache ALGOL 68
Akademie-Verlag Berlin 1977

/3/ Randell, B. and Russel, L.J.
ALGOL 60 Implementation
Academic Press 1964

Der vollständige Wortlaut des Vortrages erscheint in einem Sonderheft der Zeitschrift "Rechentechnik und Datenverarbeitung" Verlag Die Wirtschaft Berlin

eingegangen: 17. 5. 1976

Anschrift des Verfassers

Dipl.-Math. Hans-Peter Lorenzen
Wilhelm-Pieck-Universität Rostock
Rechenzentrum
25 Rostock
Albert-Einstein-Straße 21

Bernhard Essegern und Gerd Stiller

Konstruktionskriterien für Prozeßsteuersprachen

1. Einleitung

Eine Analyse aktueller und künftiger Einsatzfälle für Prozeßrechner zeigt folgende Anwendungsfälle /1/:

1. wissenschaftlich-technische und ökonomische Berechnungen
2. automatisierte Produktionssteuerung
3. Labor- und Prüffeldautomatisierung
4. intelligentes Terminal in Rechnersystemen und -netzen.

Für die Prozeßrechentechnik ist gegenwärtig charakteristisch, daß immer preisgünstigere Hardwarebausteine zur Verfügung stehen, die Kosten für die Softwareentwicklung jedoch permanent steigen und bereits über den Hardwarekosten liegen /2/. Hauptsächlich dürfte das darauf zurückzuführen sein, daß zur Zeit die Probleme in Assemblersprachen programmiert werden. Der Trend, dem "Prozeßrechnerprogrammierer" eine höhere Programmiersprache zur Verfügung zu stellen, ist jedoch unverkennbar. Dabei stützt man sich bei der Konstruktion von Prozeßrechnersprachen auf Sprachen wie FORTRAN, BASIC, PL/I und ALGOL 68. Diesbezügliche Literaturangaben sind in /3/, /4/ enthalten.

Die Programmiersprache PEARL /5/ ist die im Moment am weitesten entwickelte Prozeßrechnersprache, die eine Programmierung nahezu aller Aufgaben gemäß obiger Anwendungsfälle erlaubt.
Sie soll im folgenden kurz vorgestellt werden.

2. Die Programmiersprache PEARL

2.1. Grundlegende Sprachelemente (algorithmischer Sprachteil)

a) Es existieren Gestalten zur Darstellung von ganzen und reellen Zahlen mit unterschiedlicher Genauigkeit, von Bit- und Zeichenketten unterschiedlicher Länge, von Formaten sowie Zeitpunkten und -intervallen.

b) Als Deklarierer fungieren die

- value_attribute's, die sich unterteilen in (alle hier aufgeführten Syntaxregeln sind gegenüber dem PEARL-Bericht /5/ vereinfacht und zeigen nur das Wesentliche):

```
value_attribute ——→  simple_attribute |
                     STRUCT(...) |
                     type_indicant
simple_attribute ——→ { INT | REAL |
                       BIN | STRING |
                       FORMAT } {(int_denotation)} 1 |
                                                   0
                       CLOCK | DURATION |
                       DEVICE | INTERRUPT | SIGNAL
```

Die in der letzten Regel auftretende int_denotation enthält eine Genauigkeits- bzw. Längeninformation. Die Schlüsselwörter sprechen für sich. CLOCK und DURATION dienen der Festlegung von Zeitpunkten bzw. -intervallen.

- value_array_attribute's:
  value_array_attribute ——→ (bound_pair_list) value_attribute

c) Zur Deklaration werden bereitgestellt die

- Konstantendeklaration:

  DCL {idf | idf_list} VAL {value_attribute | value_array_attribute}

  = expression

- Variablendeklaration:

  DCL {idf | idf_list} {value_attribute | value_array-attribute}

  { = expression | := expression } 1 0

- Pointerdeklaration:

  DCL { idf | idf_list} REF {value_attribute | value_array_attribute}

  { = expression | := expression } 1 0

(Diese 3 Deklarationsformen werden zur Identitätsdeklaration zusammengefaßt. Damit sind Konstanten, Variablen und Pointer - Referenzstufe ≤ 2 - für einfache Werte, gekennzeichnet durch simple_attribute, Reihungen und Strukturen - Strukturen dürfen jedoch keine Reihungen enthalten - deklarierbar.)

- Markendeklaration (analog PL/I) zur Deklaration von Markenkonstanten und -variablen.

- Prozedurdeklaration zur Deklaration von Prozeduren mit oder ohne Parameter. Prozeduren sind keine Werte im Sinne von ALGOL 68. Die Beendigung bzw. Wertübergabe erfolgt analog PL/I mit RETURN.

- Typdeklaration zur Deklaration von type_indicant's (Einführung neuer Modi), wobei jedoch nur type_indicant's für Strukturen eingeführt werden können.

- Operatordeklaration zur Deklaration von monadischen und dyadischen Operatoren.

- Prioritätsdeklaration zur Festlegung der Priorität von im Programm definierten Operatoren.

(Die Semantik der letzten 4 Deklarationen ist ähnlich wie in ALGOL 68. Syntaktisch sind die Konstruktionen etwas modifiziert.)

- Synchronisiervariablen (-pointer)-Deklaration zur Deklaration von Semaphoren.

- Filedeklaration zur Deklaration von Dateien.

d) Zur Arbeit mit den deklarierten Objekten werden Ausdrücke und Anweisungen analog zu ALGOL 68 zur Verfügung gestellt, wobei in der Notation einige "Modifizierungen" erfolgen, z. B. bei der "Laufklausel" DO ... DONE statt do ... od usw. Das Klauselkonzept von ALGOL 68 (Vereinheitlichung von Ausdruck und Anweisung) wurde nicht beibehalten.

Bei Ausdrücken wird jeweils auf die Referenzstufe Bezug genommen, z. B. reference_one_assignment, reference_two_assignment, reference_one_slice usw.

Aufgrund der letzten beiden Konzepte wird der algorithmische Teil der Sprache unnötig unübersichtlich und kompliziert.

c) Die selbständig compilierbare Einheit ist in PEARL ein Modul:

```
program_module ——►
MODULE {LIBRARY idf}1 0 ;

{ SYSTEM; system_division
{ PROBLEM; problem_division}1 0 !
PROBLEM;   problem_division}
MODEND;
```

Eine Menge von program_module's, die nach ihrer Compilierung in einem Bindelauf mit den Funktionen des Betriebssystems und den erforderlichen Bibliotheksfunktionen verknüpft werden, bilden ein PEARL-Programmsystem. Ein Modul kann wahlweise in eine Bibliothek übernommen werden. Auf Modulebene (und nur in dieser) können globale Objekte - gekennzeichnet durch das Attribut GLOBAL -, die dem Datenaustausch zwischen den einzelnen Moduln dienen, deklariert werden.

2.2. Taskorganisation

Unter einer Task wird die dynamische Ausführung eines Algorithmus unter Steuerung eines Betriebssystems verstanden. Die Steuerung ist dadurch gekennzeichnet, daß einer Task geeignete Zustände (und u. U. weitere organisatorische Größen wie z. B. Priorität, Semaphor) zugeordnet werden, die jeweils durch eine Merkmalsmenge beschreibbar sind. Diese Zuordnung ist im PEARL-Bericht nur unscharf festgelegt, so daß semantische Unklarheiten auftreten. Es wird nicht exakt beschrieben, welche Operationen, die auch implizit (durch das Betriebssystem)

ausgeführt werden können, zu welchen Zuständen führen. Eine wesentliche Präzisierung des Zustandsmodells, das grundlegend für die Beschreibung der Semantik der Echtzeitsteuerung ist, erfolgt in /6/. Auf dieses Modell wird weiter unten eingegangen. In PEARL gibt es für das "Taskmanagement" folgende Anweisungen:

task_declaration ⟶ TASK idf [RESIDENT]$^{1}_{0}$: unlabeled_statement

task_operation ⟶

[schedule_list]$^{1}_{0}$ ACTIVATE idf [priority]$^{1}_{0}$ [sema]$^{1}_{0}$

[" "]$^{1}_{0}$ SUSPEND [idf]$^{1}_{0}$ [exception]$^{1}_{0}$

[" "]$^{1}_{0}$ CONTINUE [idf] [priority] $^{1}_{0}$ [exception]$^{1}_{0}$

[" "]$^{1}_{0}$ DELAY idf [UNTIL ce | DURING de]$^{1}_{0}$

[" "]$^{1}_{0}$ TERMINATE idf [exception]$^{1}_{0}$

[schedule_list]$^{1}_{0}$ PREVENT idf

priority ⟶ PRIORITY int_expression [REL | SYS]

sema ⟶ USING sema_idf

exception ⟶ EXCEPT [idf | idf_list]

ce ⟶ clock_expression

de ⟶ duration_expression

Durch eine (dynamische!) Prioritätsangabe kann jeder Task eine Priorität zugeordnet werden, die die Abarbeitungsreihenfolge bei gleicher Vorplanung und bei der Betriebsmittelvergabe steuert. Weiterhin kann jeder Task ein Semaphor zugeordnet werden, der automatisch bei Aktivierung der Task um 1 erniedrigt und bei natürlicher Beendigung derselben um 1 erhöht wird.

EXCEPT zeichnet eine Menge von Subtasks aus, auf die sich die vorangehende Taskoperation nicht beziehen soll.
Um eine übersichtlichere Darstellung zu erlauben, wird die folgende schedule-Regel "zweidimensional" notiert, wobei übereinanderstehende Konstruktionen Alternativen darstellen.

schedule ⟶ AT ce { EVERY ce { UNTIL ce } 1 } 1 |
                 { ALL de   { DURING de } 0 } 0 |

ON interrupt_idf { AFTER de } 1 { ALL de
                 { 0   }   { UNTIL ce } 1 } 1 |
                               { DURING de } 0 } 0 |

AFTER de { ALL de { UNTIL ce } 1 } 1 |
         {   { DURING de } 0 } 0 |

ALL de { UNTIL ce } 1
       { DURING de } 0

Unter Berücksichtigung von /6/ ergibt sich das in Abbildung 1 gezeigte Zustandsmodell. Die Pfeile geben die möglichen Übergänge zwischen den einzelnen Zuständen an. Die Ziffern an den Pfeilspitzen (Operationsnummern) legen fest, welche Operation den Übergang bewirkt.

Operationsnummer	Operationsname	Operation
1	Deklarieren	Deklaration im entsprechenden Block
2	Einplanen	ACTIVATE mit schedule
3	Aktivieren	ACTIVATE ohne schedule
4	Starten	für eine Task, die mittels Operation 2 eingeplant wurde, erfolgt Operation 3, falls die durch Operation 2 vorgeplante Bedingung eintritt
5	Unterbrechen	SUSPEND
6	Anhalten	Ausführung einer Synchronisationsfunktion, die zum Warten zwingt

Operationsnummer	Operationsname	Operation
7	Freigeben	Ausführung einer Synchronisationsfunktion, die Wartezustand aufhebt
8	Weiterführen	CONTINUE
9	Beenden	TERMINATE
10	Absetzen	PREVENT
11	Streichen	Austragen aus der Liste der zu startenden Tasks, wenn nach Operation Beenden kein nichterfülltes schedule mehr vorliegt
12	Tilgen	Verlassen des entsprechenden Blocks
13	Verzögern	DELAY
14	Enthemmen	Starten, wenn die durch DELAY festgelegte Zeitspanne verstrichen ist

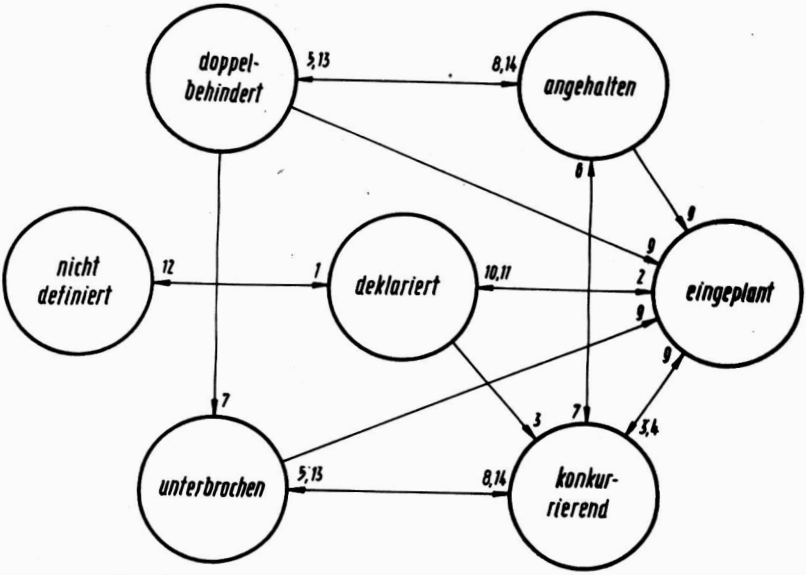


Abb. 1 Konstruktionskriterien für Prozeßsteuersprachen

2.3. Beschreibung der Prozeßperipherie (system_division)

Der system_division dient der Beschreibung der Prozeßkonfiguration. Es erfolgt eine Festlegung der
- Transgaberichtungen innerhalb der Prozeßperipherie
- Gerätebezeichnungen für die Geräte, die im problem_division ansprechbar sein sollen
- Interrupts und Signale.

Die Gestaltung des system_division soll an zwei einfachen Beispielen demonstriert werden. Zur Beschreibung der Prozeßperipherie wird als Bindeglied zwischen Rechner (hier das Prozeßrechnersystem PRS 4000) und EMSR-Einrichtungen die Prozeßeingabeausgabeeinheit PEA 4000 zugrundegelegt /7/.

Die vor dem Doppelpunkt stehenden Identifikatoren sind problemorientierte Identifikatoren (Meßstellennummern). Sie können im problem_division - nach vorheriger Spezifikation als DEVICEobjekte - in prozeßspezifischen Transgabebefehlen als aktuelle Parameter auftreten. Alle anderen Identifikatoren sind systemspezifisch und müssen dem Compiler mit ihrer immanenten Semantik bekannt sein.
Für alle Beispiele gilt, daß der Informationsaustausch vom Rechner zur Prozeßperipherie über den programmierten Kanal und die Anschlußsteuereinheit 3 erfolgt:

PRS4000 ⟷ PK;

PK ⟷ AS3;

1. Anschluß einer Analogeingabeeinheit mit Relaisdurchschaltung (AER) die 256 Eingänge hat, mit direktem Anschluß des Programmunterbrechungssignals (PU) an den ersten Zusatzunterbrechungskanal (ZUK):

   AS3 ⟵ AER;

   ZUK(1) ⟵ AER * PU;

```
/* Der Anschluß der Prozeßendstellen (Meßgeräte) erfolgt
   völlig willkürlich: */
OFEN1:TO001: ⟶ AER *1;/*Die Zahl nach dem Stern legt die
                        Eingangsnummer fest.*/
OFEN2:TO002: ⟶     *2;
   ⋮
DRUCK1:PO001: ⟶    *117;
   ⋮
DICHTE12:QO012: ⟶ *256;
```

2. Anschluß einer Digitaleingabeeinheit (statisch), die 16 Eingänge mit jeweils 16 Bit hat:

```
AS3 ⟵ DES;
SCHALTER1: ⟶ DES*1*1,1; /* 1. Zahl: Nummer des Eingangs,
                            2. Zahl: Bitposition,
                            3. Zahl: Bitanzahl */
   ⋮
SCHALTER16: ⟶*1*15,1;
   ⋮
ZAEHLER1: ⟶*2*1,8;
ZAEHLER2: ⟶*2*9,8;
   ⋮
```

2.4. Transgabe

In PEARL wird zwischen nichtorganisierter (auf Geräte wirkende) und organisierter (auf Dateien wirkende) Transgabe unterschieden. Beide Formen können formatiert oder unformatiert sein. Die Formate sind den FORTRAN- bzw. PL/I-Formaten ähnlich.

Die konventionelle Transgabe (zeichenorientiert: formatiert, unformatiert; binär) unterscheidet sich hinsichtlich ihrer Wirkungsweise nur wenig von der Transgabe allgemein bekannter Sprachen. Die prozeßspezifische Transgabe (Meßwerterfassung,

Steuerwertausgabe) wird durch Möglichkeiten der graphischen Transgabe ergänzt. Dazu werden geeignete graphische Formate bereitgestellt.

2.5. Einschätzung der Sprache

PEARL ist die derzeit umfassendste unter dem Aspekt der Prozeßsteuerung entwickelte Sprache. Diskussionen über Stand und Entwicklungen von Prozeßrechnersprachen können sehr gut von PEARL ausgehen. Der definierende Bericht /5/ enthält jedoch eine Reihe von Unklarheiten und ist an einigen Stellen fehlerhaft. Der algorithmische Kern der Sprache ist ein Hybrid aus einem abgerüsteten ALGOL 68, abgewandelt durch Elemente und Darstellungen von PL/I. Er enthält algorithmische Mittel, die im Hinblick auf den Prozeßrechnereinsatz benötigt werden. Aufwendige Laufzeitkonstruktionen und Speicherorganisationstechniken werden vermieden.

Die Sprachmischung ALGOL 68-PL/I für den algorithmischen Teil wurde zweckmäßig entworfen, unterliegt aber den grundsätzlichen methodischen Nachteilen eines solchen Vorgehens:

- das Produkt ist meist schlechter strukturiert als die Bezugssprachen. Wegen des deutlichen Qualitätsunterschiedes zwischen ALGOL 68 und PL/I ergibt sich hier der Sachverhalt, daß der PEARL-Sprachkern gegenüber ALGOL 68 einen Schritt zurück, gegenüber PL/I jedoch einen Fortschritt bedeutet.

- die semantischen Möglichkeiten bieten nichts Neues im Vergleich zu den Bezugssprachen. Dennoch wird wieder eine Vielzahl neuer Sprachelemente geschaffen und damit im wesentlichen ein Beitrag zur Erhöhung der allgemeinen Unübersichtlichkeit bzw. Konfusion bei Programmiersprachen geleistet.

Der system_division, die Transgabe und die Taskorganisation enthalten konzeptionell interessante Lösungen, wobei die Taskorganisation hervorzuheben ist. In diese Teile der Sprache sind offenbar umfangreiche prozeßspezifische Erfahrungen eingeflossen.

3. Prozeßrechnersprachen auf der Basis von ALGOL 68

3.1. Sprachversion für wissenschaftlich-technische Berechnungen (WTKR-Version)

Der wachsende Umfang moderner universeller Sprachen fordert zwingend eine klare Spracharchitektur ("Orthogonalität"):

Aufbau der Sprache aus wenigen, unikalen, kombinierbaren Konzepten /8/. Einer der Vorteile, die sich aus dem orthogonalen Aufbau einer Sprache ergeben, besteht in der übersichtlichen Ableitbarkeit von Untersprachen mit wohldefiniertem Regelwerk. Da Prozeßrechner stets in abgestuften Leistungsparametern zur Verfügung stehen, muß ein derartiges Stufenkonzept Bestandteil der Sprachdefinition sein: um einen Sprachkern sollen sich zweckmäßig strukturierte, der leistungsfähigkeit der betreffenden Anlage angepaßte Ausdrucksmittel gruppieren lassen.

ALGOL 68 /9/ erfüllt diese und noch andere für den Prozeßrechnereinsatz spezifische Forderungen gegenwärtig sehr gut.

An der Sektion Informationsverarbeitung der TU Dresden werden gegenwärtig zwei Sprachversionen für den Prozeßrechnereinsatz entwickelt. Entsprechend den Haupteinsatzfällen wird die eine Version für wissenschaftlich-technische Berechnungen die andere für Prozeßsteueraufgaben geeignet sein.
Bezüglich der Definition und der Implementierung der WTKR-Version wird auf /10, 11, 12, 13/ verwiesen.

3.1. Sprachversion für Prozeßsteuerung

Neben den üblichen Vorzügen einer höheren Sprache wie z. B. leichte Erlernbarkeit, hoher Dokumentationswert hat eine gut strukturierte Prozeßsteuersprache großen Einfluß auf die Prozeßanalyse, da wegen der allgemeineren Ausdrucksmittel auch allgemeinere Prozeßcharakteristika gefunden werden.

Dabei kommt es jedoch u. E. nicht darauf an, eine weitere neue

Sprache zu entwerfen, sondern eine bereits vorhandene leistungsstarke Sprache bezüglich ihrer Eignung zu analysieren und gegebenenfalls zu modifizieren.

Die in /11/ vorgestellte Sprachversion ist von ihrer Struktur her auch als Basis für eine Prozeßsteuersprache geeignet. Es sind jedoch aus Effektivitätsgründen einige Einschränkungen wie z. B.

- Verbieten der rekursiven Definition von Prozeduren und Operatoren
- Beschränkung der Dimension von Reihungen (z. B. auf 3)
- Verbot bzw. Einschränkung des verschachtelten Auftretens von Reihungen und Strukturen
- Verbot aller Anpassungen
- Einschränkung der laufzeitorganisierten Speicherorganisation (z. B. keine dynamischen Indexgrenzen)
- keine dynamischen Wiederholer im Formattext

vorzunehmen.

Zusätzlich sind jedoch in die WTKR-Version prozeßspezifische Ausdrucksmittel aufzunehmen.

Die beiden grundlegenden Aufgaben sind dabei die exakte Definition der Taskorganisation sowie die Beschreibung der Prozeßperipherie inklusive Prozeßkommunikation.

Alle weiteren notwendigen Ausdrucksmittel zu denen u. a.

- Kennzeichnung reentranter oder residenter Objekte
- Definition von Bolt- und Ereignisvariablen
- Interruptsimulation und -unterdrückung

gehören, sind relativ einfach zu definieren. Bei der Beschreibung wird dabei der volle Sprachumfang von ALGOL 68 ausgenutzt, das ist möglich, da die benötigten Modi und Operatoren durch ein geeignetes Vorspiel vorgegeben werden.

Explizite Gedanken über die Definition einer Prozeßsteuersprache auf der Basis von ALGOL 68 werden in /14/ vorgestellt.

Literaturverzeichnis

/1/ "Rechentechnik/Datenverarbeitung" 3. Beiheft 1975

/2/ Fachtagung Prozeßrechner, Karlsruhe 1974 enthalten in: Goos, G.; Hartmanis, J. (Ed.): Lecture Notes in Computer Science 12 Berlin, Heidelberg, New York; Springer Verlag 1974

/3/ Stiller, G.; Loeper, H.: Einschätzung des internationalen Standes und Entwicklungstrends auf dem Gebiet der problemorientierten Sprachen für Prozeßrechner Referat, gehalten auf der Konferenz "Entwicklung und Anwendung der elektronischen Rechentechnik in der DDR" Dresden, 26. - 28. 11. 1975; zur Veröffentlichung in "Rechentechnik/Datenverarbeitung" vorgesehen

/4/ Born, W.: Entwurf von prozeßsteuerungsorientierten Programmiersprachen Dissertation A, TU Dresden 1974

/5/ Timmesfeld, K.H. et al.: PEARL. Gesellschaft für Kernforschung mbH, Karlsruhe, Report KFK-PDV 1, April 1973

/6/ Rieder, P.: Prozeßzustände bei Echtzeitprogrammiersprachen enthalten in /2/

/7/ Autorenkollektiv: Steuerprogrammsystem ESKO 4000. Systemunterlagendokumentation VEB Kombinat Robotron, 1974

/8/ Lehmann, N.J.; Stiller, G.: Einige methodische Aspekte der Entwicklung und Nutzung höherer Programmiersprachen zur Veröffentlichung in "Rechentechnik/Datenverarbeitung" vorgesehen

/9/ Wijngaarden, A. van et al.: Revised Report on the Algorithmic Language Algol 68 Acta Informatica Vol. 5, Fasc. 1- 3, 1975

/10/ Stiller, G.; Loeper, H.; Essegern, B.: Eine Untersprache von ALGOL 68 für den prozeßgekoppelten Betrieb Schriftenreihe der IHS Dresden, 5 (1975) Heft 3

/11/ Essegern, B; Stiller, G.; Loeper, H.: Definition von Programmiersprachen auf der Basis von ALGOL 68 für das Prozeßrechnersystem PRS 4000
MKÖ IV, Dresden, 1975;
zur Veröffentlichung in "Wissenschaftliche Zeitschrift der TU Dresden" vorgesehen

/12/ Loeper, H.; Horn, M.; Brankowa, E.: Präzedenzgesteuerte Syntaxanalyse einer Untersprache von ALGOL 68
MKÖ IV, Dresden, 1975

/13/ Otter, W.; Jähmlich, H.; Loeper, H.: Führung der Tabellen und Behandlung von Kontextabhängigkeiten im lexikalen Analysator eines Übersetzungsprogrammes für eine Untersprache von ALGOL 68
MKÖ IV, Dresden, 1975

/14/ Essegern, B.; Stiller, G.: Entwurf eines "Systemvorspiels" als Bestandteil einer ALGOL68-Implementierung für Prozeßrechnereinsatz
8. Kolloquium über Rechentechnik und Datenverarbeitung, Magdeburg, Juni 1976

eingegangen: 17. 5. 1976

Anschrift des Verfassers

Dipl.-Phys. Bernhard Essegern;

Prof. Dr.rer.nat.habil. Gerd Stiller

TU Dresden, Sektion Informationsverarbeitung,

8027 Dresden, Mommsenstraße 13

Immo O. Kerner

Operatoridentifizierung in Programmiersprachen

Zusammenfassung

Die Problemstellung der Operatoridentifizierung zur Compilierzeit wird für Programmiersprachen erläutert und formuliert. Speziell für die Sprache Revidiertes ALGOL 68 wird die Behandlung des Problems in der Rostocker Implementierung beschrieben. Die Darstellung ist frei von Besonderheiten des zugrunde liegenden Computers der ESER-Reihe.

Die Problematik der automatischen Anpassung der Wertarten (Modi) für Operanden ist enthalten, auch für den Sonderfall der Balance mehrerer zunächst angebotener Modi für einen Operanden.

1. Allgemeines und Einführung

Für problemorientierte Programmiersprachen schafft man üblicherweise - dem Nutzer und Anwender entgegenkommend - die Möglichkeit der "Formel"-Notierung, da eine Prozedurnotierung schwerfällig und für die menschliche durch langjährigen und historisch bedingten Erziehungs- und Ausbildungsprozeß (Gewöhnung) geformte Auffassungsgabe fremd erscheint.
Man vergleiche

```
a * (b + c)
```

mit

```
Mult(a,Add(b,c)).
```

Für Spezialsprachen oder in erweiterbaren Sprachen für spezielle abgeleitete Fachsprachen wird man fachspezifische Operatoren neu deklarierbar zulassen, ihre Priorität oder Vorrangeinstufung frei festlegbar halten und sogar weitere zusätzliche Bedeutungen bereits eingeführter Operatoren definierbar machen müssen. Auch damit folgt man einem üblichen anwendungsbezogenen und schon lange praktizierten Verfahren. Bereits das Pluszeichen hat in Mathematik und Technik je nach den dabei stehen-

den Operanden unterschiedlichste Bedeutung:

i + j,	ganzzahlige Operanden - Festkommaaddition
x + y,	reelle Operanden - Gleitkommaaddition
u + v,	komplexe Operanden - komplexe (zweimalige) Gleitkommaaddition
p + q,	logische Operanden - logische Oderoperation
s + t,	Zeichenkettenoperanden - Verkettungsoperation
a + b,	Vektoren oder Matrizen - Vektor- oder Matrixaddition.

Diese bekannte Erscheinung nennt man "Überladen" von Operatoren (Überladen mit Bedeutung). Eine DVA oder ein Computer, der Formeln einer Programmiersprache mit überladenen Operatoren abzuarbeiten hat, muß die gerade oder aktuell vorliegende Bedeutung erkennen. Die entsprechende Definition des Operators muß "identifiziert" werden.

Die Problematik wird komplizierter bei Sprachen mit Blockstruktur, d.h., mit unterschiedlichen Definitions- oder Gültigkeitsbereichen. Aber gerade solche Sprachen entsprechen dem Zug der Entwicklung, der in Richtung der "strukturierten Programmierung" geht, d.h., vom groben oder globalen Lösungsweg eines Problems über immer feinere Strukturen für Teilprobleme bis zur endgültigen Form in Befehlen oder Anweisungen niedergeschrieben.

In vielen Implementierungen von Programmiersprachen mit überladenen Operatoren wurde diese Identifizierung der aktuellen Operatordefinition auf die Laufzeit der Programme verlagert. Das ging oft auch nicht anders, da z.B. bei Prozedurdeklarationen in ALGOL 60 oder SUBROUTINE-Definitionen in FORTRAN einfach noch keine vollständigen Informationen über den Datentyp der Parameter vorliegen. Tritt also im Körper einer Prozedur oder einer Subroutine beispielsweise das "+" mit formalen Parametern als Operanden auf, so kann erst bei einem Aufruf (CALL) der Prozedur oder der Subroutine je nach den aktuellen Parametern die Identifizierung vorgenommen werden. Jedoch in

zahlreichen anders gelagerten Fällen ist die fortwährende Typprüfung von Operanden eine überaus große und dazu unnötige Belastung für die Laufzeit. Wohlüberlegte und fachmännisch konstruierte moderne Programmiersprachen vermeiden dies und verwenden konsequent solche Konzepte für sprachliche Formulierungen, daß die Operatoridentifizierung bereits zur Compilerzeit ermöglicht wird. Die Sprache ALGOL 68 gehört zu dieser Gruppe. Die nutzbringende Auswirkung zeigt eine Information über eine ALGOL 68-Implementierung (Bra 74). Es wurden gleiche Problemstellungen sowohl mit ALGOL 60 als auch mit ALGOL 68 auf derselben Anlage bearbeitet. Die Laufzeiten für die ALGOL 68-Programme sanken bis auf den dritten Teil der entsprechenden von ALGOL 60-Programmen ab. Das ist natürlich nicht nur allein auf die bessere Operatoridentifizierung zurückzuführen. ALGOL 68 enthält viele Konzepte, welche sich auf eine Laufzeitverbesserung der Programme auswirken (Wij 75).

Programmiersprachen, welche von kapitalistischen Computerproduzenten angeboten werden, sollten seitens der Anwender mit allergrößter Vorsicht betrachtet werden. Das Hauptziel dieser Firmen ist der möglichst große Vertrieb von Datenverarbeitungsanlagen. Das geht natürlich nur, wenn den Käufern besondere Vorteile gerade dieser Anlagen suggeriert werden. Tatsächlich aber beweisen die Praktiken, daß die in diesem Zusammenhang mit angebotene Software, wozu auch Programmiersprachen, Compiler und Betriebssysteme gehören, meist speicheraufwendig und wenig effektiv arbeitet. Als Folge davon meinen die Nutzer recht bald, das nächst höhere Glied einer Rechnerfamilie oder einen schnelleren und mehr Speicher besitzenden Computer benötigen zu müssen. Das muß dann natürlich eine Anlage derselben Firma sein, denn Peripherie und Programme können wegen der "Aufwärtskompatibilität" - ein weiterer Verkaufstrick - beibehalten werden (Wa 69). Da höhere Systemfamilienglieder nicht einfach durch höhere Rechengeschwindigkeit und mehr Speicherplatzangebot ausgezeichnet sind, sondern auch qualitativ in der Struktur mehr bieten müssen, können bei Weiterverwendung der Programme niederer Systemstufen gerade diese besseren

Qualifikationen nicht angesprochen werden. Somit erweist sich die anscheinend so nutzerfreundliche Aufwärtskompatibilität als ein trojanisches Pferd, denn gerade dadurch wird die bessere Anlage wiederum besonders uneffektiv belastet und abermals wünscht sich der Kunde ... siehe oben.

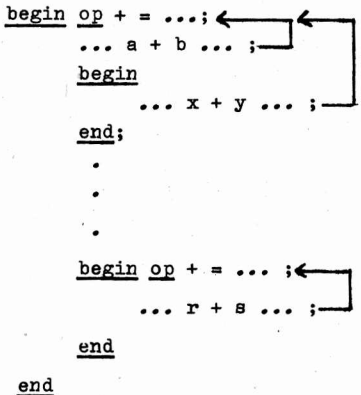
2. Operatordefinition in ALGOL 68

Eine Operatordefinition kann in ALGOL 68 vom Programmierer selbst nach Bedarf vorgenommen werden. Diese Sprache befolgt damit das Orthogonalitätsprinzip, nach dem ein Konzept, hier das der Deklaration, für möglichst viele Sprachstrukturen, hier außer den Variablen auch für Konstante, für Wertarten (Typen) oder Modi und eben auch für Operatoren, angewendet werden kann. Der Aufbau einer "Operationsdeklaration", beispielsweise

op + = (modlo a, modro b) modres: (...),

beginnt mit dem die Deklaration ankündigenden Symbol op, dem die spezielle Operatordarstellung oder Operatorindikation folgt. Die Operatorindikation ist entweder ein Symbol (+), eine vom Programmierer neu eingeführte Indikation mit Symbolcharakter (plus) oder eine nach bestimmten Regeln vorgenommene Fügung von vorhandenen Symbolen (+=). Dem sich anschließenden Gleichsymbol als dem "ist definiert als"-Zeichen folgt die geklammerte formale Operandenliste. Sie enthält stets zwei - für dyadische oder zweistellige Operatoren- Operandenangaben oder aber nur eine - für monadische oder einstellige Operatoren. Die formalen Operanden, linker und rechter, sind mit genauen Wertetyp- oder Modusangaben versehen. Der formalen Operandenliste folgt die Angabe des Resultatmodus für den Wert, der durch Ausführung der Operation auf die Operanden entsteht. Dem sich daran anschliessenden Doppelpunkt (Routinesymbol) folgt eine Einzelklausel, d.h., eine Anweisung oder auch nur ein Ausdruck, welche die Arbeit der Operation beschreibt. Da dies meist nicht durch eine einzige Klausel beschreibbar sein wird, findet man in der Regel eine geklammerte Klausel in der Rolle der Einzelklausel.

Operationsdeklarationen dürfen genau wie Deklarationen für Variable in geschachtelten Blöcken auftreten und haben auf diese Weise unterschiedliche Gültigkeitsbereiche. Sind sie nicht überladen, könnte die Identifizierung genau wie bei Variablen erfolgen ("könnte", da auf die Modusverträglichkeit der Operanden geachtet werden muß).



Die Pfeile geben die von den einzelnen "angewandten" Auftreten der Operatoren in Formeln identifizierten "definierenden" Auftreten der Operatoren in Operationsdeklarationen an, ohne jede Beachtung der Operandenmodi zunächst.

Operatoren haben gegenüber anderen Operatoren Vorrangeinstufungen, die die Abarbeitungsreihenfolge regeln. Normalerweise oder üblicherweise hat der durch "×" angegebene Operator (Multiplikation) gegenüber dem durch "+" angegebenen (Addition) Vorrang oder höhere Priorität, was in ALGOL 68 durch die Prioritätsdeklarationen

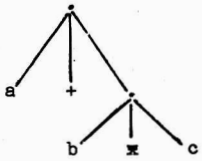
```
prio × = 7; prio + = 6;
```

oder einfacher

```
prio × = 7, + = 6;
```

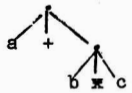
ausgedrückt wird. Der Programmierer kann jedoch durch eingestreute Prioritätsdeklarationen dies für gewünschte Operatoren

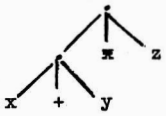
ändern. Während üblicherweise dem Ausdruck a + b ¤ c wegen der größeren Priorität der Multiplikation bei der syntaktischen Analyse der Zerlegungsbaum

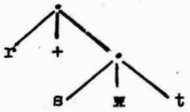


zugeordnet wird, ist dies im folgenden Programm abgeändert worden.

```
begin op + = ... ;
      ... a + b ¤ c ...          hier gilt noch
      begin prio + = 8;
            ... x + y ¤ z ...
                                 aber hier ist
      end;
       .
       .
       .
      begin
            ... r + s ¤ t ...    hier ist wieder
      end
end
```







Prioritätsdeklarationen genügen also auch dem bekannten Prinzip der Gültigkeitsbereiche bei Blockschachtelungen (Orthogonalitätsprinzip!). Wie bei Variablen dürfen Prioritätsdeklarationen nicht mehrfach für dieselbe Operatorindikation im gleichen Definitionsbereich auftreten. Welche Operationsdeklaration durch ein angewandtes Auftreten eines Operators auch identifiziert wird, die Prioritätsidentifizierung folgt genau dem gleichen Prinzip wie die Identifizierung von Variablen.

Dieser Sachverhalt ist von den Sprachkonstrukteuren beabsichtigt, um dem Compiler eine Syntaxanalyse zunächst unabhängig von den im Programm vorhandenen Wertarten (Typen oder Modi) zu ermöglichen. Ein Vorprozessor ordnet den angewandten Auftreten der Operatoren Prioritäten zu. Die anschließende Syntaxanalyse erzeugt in einer Grundstufe für Formeln den entsprechenden Baum. Für die semantische Synthese oder Codegenerierung durch den Compiler muß aber nun das genaue definierende Auftreten, d.h., die gemeinte Operationsdeklaration, identifiziert werden. Dazu werden in der Oberstufe der Syntaxanalyse die Modi der aktuellen Operanden benutzt.

3. Problematik der Operatoridentifizierung

Bei überladenen Operatoren wird der Modus der aktuellen Operanden zur Identifizierung herangezogen.

```
op + = (real a,b) real: (...);
op + = (int a, b) int: (...);
real x, y; int i, j;
                    3 + 4
                    i + j
                3.14  + 2.72
                    x + y
```

Liegen im selben Gültigkeitsbereich zwei Operationsdeklarationen für denselben Operatorindikator vor, so muß bei angewandten Auftreten zwischen ihnen entschieden werden. Während einige "Anpassungen", wie z.B. Entreferenzieren, d.h., der Übergang von der Adresse zum adressierten Wert, automatisch vollzogen werden können, darf das bei anderen nicht geschehen. Eine solche verbotene automatische Anpassung des Operandenmodus ist das Erweitern von 'int' auf 'real'. Im obigen Beispiel wäre dann eine eindeutige Operatoridentifizierung für das erste angewandte Auftreten nicht möglich. Eindeutig bleibt dagegen bei x + y das erste definierende und bei i + j das zweite definierende Auftreten als das zu recht identifizierte.

Im selben Definitionsbereich dürfen deshalb keine zwei Operationsdeklarationen gültig sein, deren formale Operanden durch die bei Operanden erlaubten Anpassungen modusmäßig ineinander überführbar sind.

Die bei Operanden in Formeln erlaubten Anpassungen gehören zu der Anpassungsklasse "fest", welche

Entreferenzieren
Entprozedurieren und
Vereinen

umfassen. Man sagt "Operanden stehen in fester Programmposition". Die bereits formulierte Kontextbedingung über das gleichzeitige Auftreten von Operationsdeklarationen kann dann wie folgt gefaßt werden:

> Im gleichen Definitionsbereich dürfen keine zwei Operationsdeklarationen für denselben Operatorindikator auftreten, deren formale Operanden in "fester Verwandtschaft" stehen.

Feste Verwandtschaft bedeutet dabei die Möglichkeit des Überführens der Modi durch feste Anpassungen ineinander oder genauer gesagt: Es gibt für zwei Modi m_1 und m_2, die in fester Verwandtschaft stehen, einen Modus m_0 (möglicherweise identisch mit m_1 oder m_2), so daß durch feste Anpassungen m_0 in m_1 und auch m_0 in m_2 überführbar sind.

So sind die Operationsdeklarationen

```
op + = (ref real a, b) ...;
op + = (real a,b) ...;
```

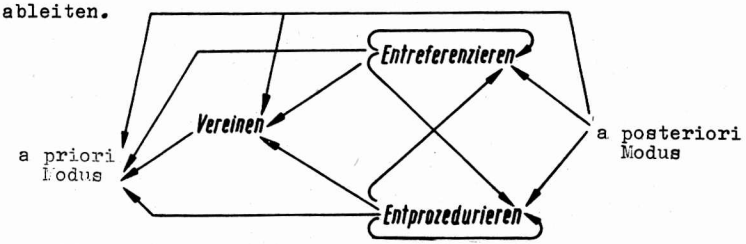
nebeneinander verboten, weil dann für

```
x + y
```

keine eindeutige Identifizierung möglich ist.
Bei der Operatoridentifizierung sind also nicht nur die formalen und aktuellen Operandenmodi zu vergleichen. Es ist vielmehr zu überprüfen, ob die aktuellen Operandenmodi durch eine Folge fester Anpassungen in die formalen Operandenmodi überführbar sind.

4. Feste Anpassungen und die Modusliste

Der Revidierte ALGOL 68 - Bericht (Wij 75) läßt für feste Anpassungen nur ganz bestimmte Reihenfolgen als erlaubt zu. Aus den entsprechenden Syntaxregeln kann man die folgende Graphik ableiten.



Mit einem unangepaßten oder a priori-Modus geht man rechts in die Graphik hinein, es ist dies der aktuelle Operandenmodus, und sucht einen solchen Weg zu finden, daß der angepaßte oder a posteriori-Modus entsteht, der durch den formalen Operandenmodus gegeben ist. Jeder Durchlauf durch ein Kästchen bedeutet eine implizite, d.h., vom Programmierer nicht explizit notierte aber vom Compiler einzufügende Anpassungsoperation, die auf den Operanden anzuwenden ist.

$$\text{Operand mit a posteriori Modus} = \left\{ \underline{\text{unite}} \right\}_0^1 \left\{ \begin{matrix} \underline{\text{deref}} \\ \underline{\text{deproc}} \end{matrix} \right\} * \quad \text{Operand mit a priori Modus}$$

Das Entprozedurieren wird bei Bezeichnungen für parameterlose Prozeduren gebraucht. So sei etwa "random" die Bezeichnung für eine als Zufallszahlgenerator arbeitende Prozedur, die bei jedem Aufruf eine neue Zufallszahl (aus dem Intervall (0,1) und gleichverteilt) liefert. Die Größe "random" hat den Modus 'procedure yielding real' und in der Formel

```
3.14 + random
```

wird, um einen real-Wert als rechten Operanden zu erhalten, nicht entreferenziert sondern eben entprozeduriert. Die Prozedur wird dabei aufgerufen und mit ihrem Resultatwert wird weiter gearbeitet. Entreferenzieren und entprozedurieren kön-

nen sich beliebig oft ablösen, denn es kann beispielsweise eine Prozedur geben, die eine Referenz zu einer weiteren Prozedur liefert, die ihrerseits erst eine Referenz zu einer real-Größe liefert, u.ä. andere Kombinationen.

Das mögliche sich daran anschließende Vereinen wird für solche Operanden in Operationen gebraucht, die mit mehreren Datentypen arbeiten sollen. Obwohl die Ein-/Ausgabeanweisungen keine Operatoren oder Formeln sind, sind ihre aktuellen Parameter auch in fester Position und das nötige Vereinen kann an ihnen erläutert werden. Man erwartet von

print (x),

daß es unterschiedliche Druckbilder für 'int', 'real', 'bool', 'char' und 'string'-Parameter liefert, denn man kann dem Nutzer nicht zumuten, daß er mit unterschiedlichen Ausgabebefehlen für jeden Ausgabemodus hantiert. In der Deklaration für "print" stehen demnach

```
proc print = (union(int, real, bool, char, string) w) void:
    case w in (int i): ...,
              (real r): ...,
              (bool b): ...,
              (char c): ...,
              (string s): ...
    esac;
```

und erst in der Konformitätsfallklausel case ... esac wird nach den Datentypen getrennt das Druckbild zusammengestellt. Wird nun aktuell

print (3.14)

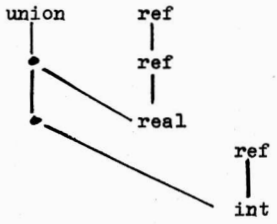
verlangt, so muß der real-Wert 3.14 zu einem 'union of int real bool char string'-Wert vereint werden. Der Compiler fügt eine Anpassung unite ein und zwar so, daß dem real-Wert das Kennzeichen 'real' zugefügt wird, damit dies dann bei der Abarbeitung der print-Prozedur entsprechend erkannt wird.

Die Prüfung der festen Anpaßbarkeit und Bestimmung der nötigen Anpassungsoperationen sind eng mit den Modi verbunden. Die

Effektivität dieser Aufgabe des Compilers hängt in großem Maße von der internen Darstellung der Modi ab. In einer Modusliste ist jeder in einem speziellen Programm auftretende Modus als eine verkettete Datenstruktur, als ein Baum, eingetragen. Die Modusliste ist also ein Wald. Sie ist reduziert in dem Sinne, daß kein Modus doppelt auftritt und Teilmodi auch Teilbäume sind. Sie ist normiert in dem Sinn, daß vereinte Modi entflochten sind und ihre Teilmodi in festgelegter Reihenfolge stehen.

Beispiel:

Eintrittspunkt	Marke	down	next
100	ref	101	nil
101	ref	102	nil
102	real	nil	nil
103	ref	104	nil
104	int	nil	nil
105	union	106	nil
106		102	107
107		104	nil



Die Eintrittspunkte

100	beschreiben die durch	ref ref real
101		ref real
102		real
103		ref int
104		int
105	union (real, int)	dargestellten

Modi. Da nur Binärbäume vorgesehen sind, machen sich einige Hilfsknoten nötig (siehe Vereinigungsmodus). Ein Modusvergleich reduziert sich auf den Vergleich der zugeordneten Eintrittspunkte, d.h., auf einen Adressen- oder Namensvergleich.

Der als Beispiel genommene Vereinigungsmodus union (real, int) ist identisch mit union (int, real) und auch mit dem noch nicht entflochtenen union (int, union (int, real)). Modusindikationen sind in dieser Liste nicht mehr zu finden. Die Liste ent-

hält nur entwickelte Modi, die sich aus den Modusbausteinen von ALGOL 68 zusammensetzen.

5. Operatorindikationen und die Operatorliste

Im Revidierten ALGOL 68 gibt es eine Anzahl von Symbolen, die Grundoperationen darstellen. Nur für dyadische Operationen dürfen verwendet werden

$$< \quad > \quad = \quad / \quad \times \quad \pi \quad .$$

Ihre normale Bedeutung ist identisch mit der in der Mathematik bekannten. Sowohl für dyadische als auch für monadische Operationen dürfen verwendet werden

∨ ∧ & ¬ ~ (logische Operatoren, & auch für "und", ~ auch für "nicht")
≠ ≤ ≥
÷ % (% auch für ÷)
∈ ⌊ ⌈ Element von, untere- und obere Grenze von)
⊥ (plus i mal)
↑ ↓ (Synchronisierungsoperationen, ↑ auch für Potenz)
+ −

und dann noch etliche weitere, wenn der Zeichenvorrat der Implementierung dies zuläßt, wie etwa "!" und "?".
Außer diesen Grundoperatoren, eventuell sind in einigen Implementierungen einige der zweiten Gruppe (z.B. ≤ und ≥) nicht verfügbar, können weitere aus ihnen zusammengesetzt werden. Für dyadische Operatoren kann man einen ersten aus einer der beiden Gruppen auswählen und einen aus der ersten Gruppe folgen lassen. Es ist also möglich

$$\pi\pi \quad >= \qquad <= \qquad = / \text{ oder } / =$$

oder %π u.ä. zu bilden. Damit können Lücken im Operatorangebot des Implementierungsalphabets ausgeglichen werden. Für monadische Operatoren muß das erste Zeichen aus der zweiten Gruppe stammen, als z.B. %π.
An so aufgebaute oder auch an Grundoperatoren darf noch ein

$$:= \quad \text{oder} \quad =:$$

angehängt werden.

Außerdem sind weitere Operatoren nach dem Belieben des Programmierers möglich in der Form

p teilt machtmit oder add u.a.

Diese sind nicht kombinierbar!
Nach diesen Vorschriften lassen sich auch längere Ketten monadischer Operatoren in Formeln bereits von einem Vorprozessor auflösen:

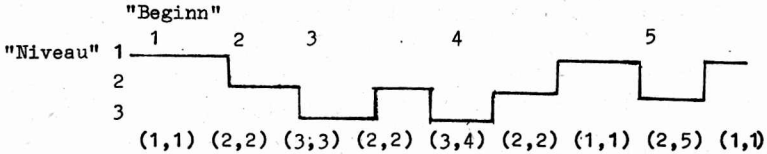
(das m steht für monadisch, das d für dyadisch).

Der Vorprozessor stellt eine Operatorliste auf, identifiziert die Prioritäten und setzt für die Operatorindikationen in die interne Form eines Programmes einfach jeweils ein

Operator-mit-Priorität-p-Symbol

mit einem Eintrittspunkt in die Operatorliste ein. Für die Grundstufe der Syntaxanalyse ist nur die Priorität von Belang. Der mitgeführte Eintritt in die Operatorliste ist jeweils der Anfang einer Kette von Datenstrukturen, die die definierenden Auftreten eines überladenen Operators (Operatorindikation) unter Beachtung der Blockstruktur des Programmes verbinden.

Es liege beispielsweise die folgende Programmblockverschachtelung vor:



Die einzelnen Definitionsbereiche können durch einen Zweikomponentenzähler

(n, b)

eindeutig bestimmt werden. Dabei ist n das Niveau oder die Verschachtelungstiefe und b der Beginnzähler oder Blockanfang-

zähler. Beim Abstieg oder Blockanfang wird der neue Zähler aus

$$(n+1,\ b+1)$$

d.h.

$$n := n + 1$$
$$b := b + 1$$

erhalten. Beim Aufstieg wird

$$n := n - 1$$
$$b := b - k, \text{ wobei } 1 \leq k \leq b - 1$$

gebildet. Das k ist die kleinste Zahl aus dem genannten Bereich, so daß das b das größte der b_i aus den im Programm auftretenden Zählerständen (n, b_i) ist. Damit ergeben sich die oben angeführten Zählerstände für die einzelnen Programmabschnitte.

Ein Vorprozessor legt noch vor der Syntaxanalyse eine Operatorliste entsprechend der Programmschachtelung und den definierenden Auftreten von Operatorindikationen an. Die Liste verkettet überladene Operatoren im gleichen Niveau von vorn nach hinten und in eingeschachtelten Niveaus von innen nach außen. Die Verkettung ist somit zugleich die Suchrichtung bei der Operatoridentifizierung. Die Moduseintragungen für die Operanden können vom Vorprozessor noch nicht eingetragen werden, da die formalen Deklarierer aus den Operatordefinitionen erst analysiert werden müssen.

Beispiel einer Operatorliste,

Die Spalte "Routine" beinhaltet einen Hinweis (zunächst auf den Interntext des Quellprogramms, später auf das Objektprogramm, und zwar an die Stelle, an der der Code für die Operationsroutine beginnt). Dies wird bei der Objektcodegenerierung als Sprung- oder Rufadresse zur Operationsabarbeitung verwendet.

Die Spalten "Zähler" und "Indikation" brauchen bei der Implementierung nicht mehr mitgeführt zu werden. Sie stehen hier lediglich zur Illustration. Natürlich muß der Vorprozessor während des Aufbaus der Operatorliste entsprechende Informationen halten.

Es wird angenommen, daß die Operatoren "+" und "-" in den oben als Beispiel einer Blockschachtelung gegebenen Programmabschnitten wiederholt und damit in unterschiedlichen Niveau überladen definiert werden.

Eintrittspunkt	Zähler	Indikation	Verkettung d. Überldg.	1.Mod. 1.Opd.	2. Mod. r.Opd.	Mod. Re.	Routine
150	(1,1)	+	151				
151		+	152				
152		+	170				
153		-	154				
154		-	168				
155	(2,2)	+	156				
156		+	161				
157		-	158				
158		-	162				
159	(3,3)	+	155				
160		-	157				
161	(2,2)	+	166				
162		-	167				
163	(3,4)	+	164				
164		+	155				
165		-	157				
166	(2,2)	+	150				
167		-	153				
168	(1,1)	-	169				
169		-	176				
170		+	171				
171		+	178				
172	(2,5)	-	174				
173		+	175				
174		-	153				
175		+	150				
176	(1,1)	-	177				
177		-	nil				
178		+	nil				

Die Eintrittspunkte in die Operatorliste sind im Beispiel für den Operator + -
bei angewandten Auftreten
im Abschnitt (n,b)

(1,1)	150	153
(2,2)	155	157
(2,5)	173	174
(3,3)	159	160
(3,4)	163	165

und diese Eintrittspunkte setzt der Vorprozessor im Internprogrammtext mit ab, da diese die Entscheidung über die Operatorindikation beinhalten und außerdem den Suchbeginn bei der Operatoridentifikation darstellen. Die Syntaxanalyse übermittelt dann nach Festlegung der Operanden- und Resultatmodi, sowie nach der Bestimmung der Baumstruktur diese Eintrittspunkte an den Teilprozeß der Operatoridentifizierung.

6. Aufgabenstellung der Operatoridentifizierung

Die Operatoridentifizierung wird während der Syntaxanalyse immer dann aufgerufen, wenn die Originalsyntax von ALGOL 68 die Abarbeitung des Prädikats

TAO identified in NEST

verlangt. Dabei ist hier TAO eine Sammelbezeichnung für Operatorindikation und NEST entspricht der Operatorliste. Die Prüfung der Kontextbedingungen, d.h., die Abarbeitung der entsprechenden Prädikate für Operatoren ist vorausgegangen. Die Operatorliste ist "sauber", d.h., im selben Gültigkeitsbereich gibt es keine fest verwandten überladenen Operatoreintragungen. In der Operatorliste sind auch bereits alle Modi für die Operanden und Resultate eingetragen worden. Diese Eintragungen sind Adressenangaben, welche in die Modusliste zeigen.

Die Syntaxanalyse ruft die Operatoridentifizierung auf mit den Parametern

P1: Ein- oder zwei Modusangaben für die a priori-Modi der

aktuellen Operanden (monadischer oder dynamischer Operator) als Referenzen zur Modusliste

P2: Einen Eintrittspunkt in die Operatorliste (vom Vorprozessor übernommen).

Als Hilfsmittel stehen zur Verfügung:

H1: Operatorliste mit den Verkettungen überladener Operatoren in Suchrichtung

H2: Modusliste als Wald von Bäumen.

Beide Listen sind in der beschriebenen Art "sauber", d.h., Kontextbedingungen wurden bearbeitet.
Die Operatoridentifizierung soll als Resultat liefern

R1: Das definierende Auftreten des identifizierten Operators als eine Referenz zur Operatorliste, womit auch die zu aktivierende Routine (letzte Spalte) bekannt ist.

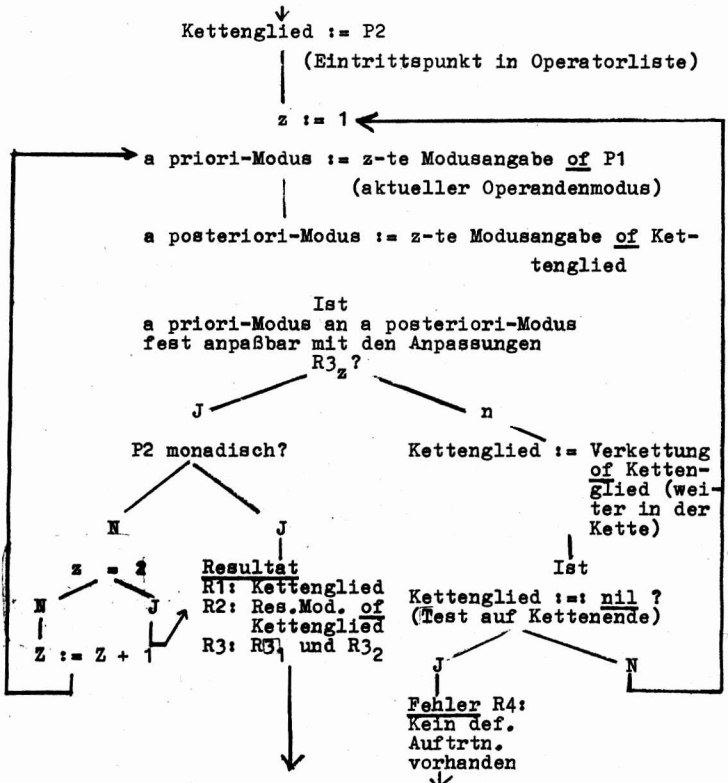
R2: Den Resultatmodus, denn dieser kann wiederum ein Operandenmodus für eine weitere Operation sein, und muß dem Analyseprogramm übergeben werden. Er ist durch das identifizierte Auftreten in der Operatorliste (vorletzte Spalte) bekannt.

R3: Die Folge der benötigten Anpassungsoperationen für die einzelnen Operanden.

R4: Gegebenenfalls eine Fehlermeldung.

7. Lösungsplan für die Operatoridentifizierung

Der Lösungsplan wird zunächst in grober Form als Flußdiagramm gegeben wegen der Anschaulichkeit und später verfeinert (strukturierte Programmierung).



Das Revidierte ALGOL 68 enthält eine Besonderheit bzgl. der Operatoridentifizierung, die durch folgendes Beispiel illustriert wird.

```
begin op p = (real x,y) real; x+y;
        .
        .
        .
      begin op p = (ref real x,y) real; x-y;
            real u := 3.14, v := 2.71;
```

```
        .
        .
        .
    u p v
        .
        .
        .
  3.14 p 2.71
        .
        .
        .
  end
end
```

Obwohl die beiden Operatoren p fest verwandt sind, liegt doch keine Verletzung der Kontextbedingung vor, da sie in unterschiedlichen Definitionsbereichen liegen. Das angewandte Auftreten

u p v

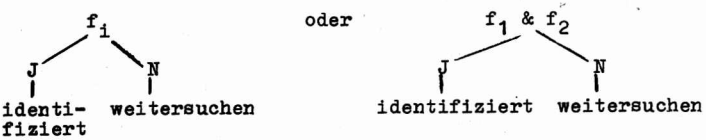
identifiziert (Operandenmodus 'ref real') das innere p und die Abarbeitung liefert den real-Wert 0.43, während das angewandte Auftreten

3.14 p 2.71

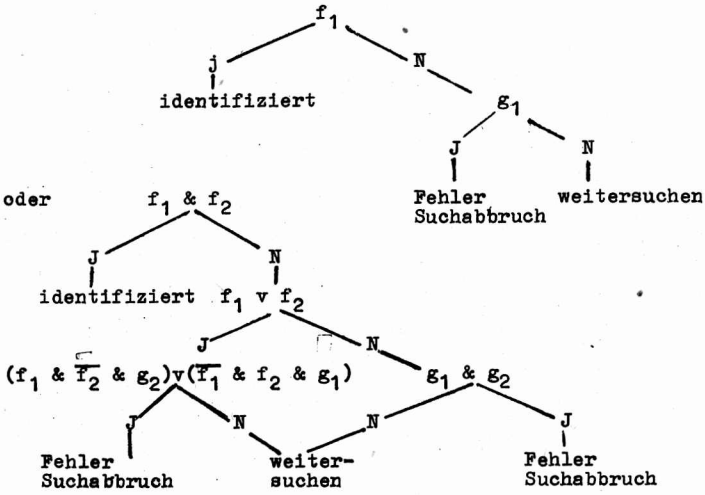
(Operandenmodus 'real') nach dem bisher gesagten das äußere p identifizieren müßte, weil ja 'real' nicht fest an 'ref real' anpaßbar ist. Die Abarbeitung liefert dann - sicherlich zum Erstaunen des Programmierers - den Wert 5.85!
Solche Fälle sollen vermieden werden und ein derartiges Auftreten, nämlich "3.14 p 2.71" wird als fehlerhaft ausgewiesen. Es ist also bei fester Nichtanpaßbarkeit des a priori-Modus an den a posteriori-Modus noch zu prüfen, ob vielleicht umgekehrt feste Anpaßbarkeit vorliegt.

Wenn f_i feste Anpaßbarkeit des i-ten a priori-Modus an den

i-ten a posteriori-Modus bedeutet und g_i entsprechend umgekehrt, so muß der bis jetzt gültige Abfragegraph



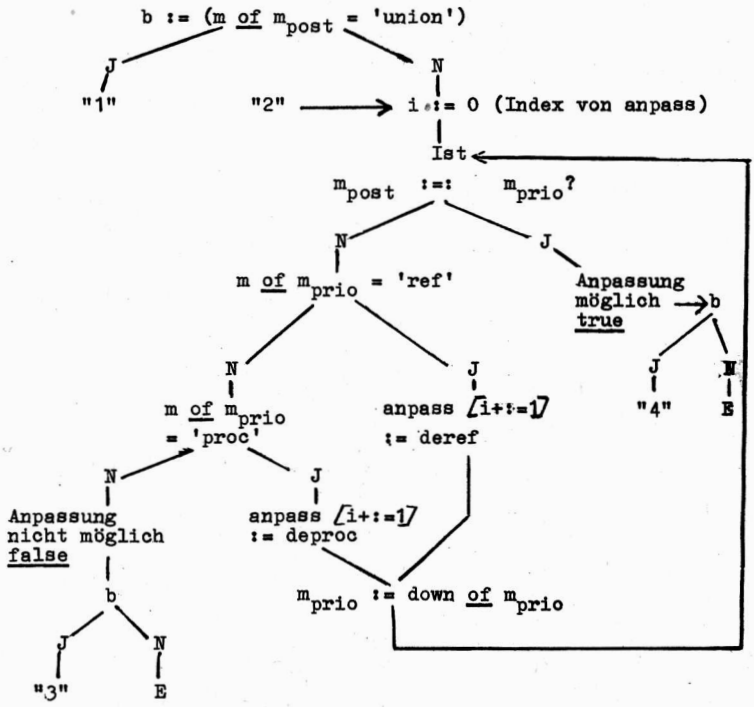
bei monadischen bzw. dyadischen Operatoren ergänzt werden zu



8. Prüfung auf Anpaßbarkeit

Die Prüfung auf Anpaßbarkeit (feste) und das Bestimmen der Anpassungsoperationen erfolgt gemäß der Eingänge in die Modusliste, die den a priori- und a posteriori-Modi m_{prio} und m_{post} entsprechen. Zunächst erfolgt ein Test auf 'union' bei m_{post}. Im nein-Fall wird m_{prio} solange entreferenziert und entprozeduriert bis der Restmodus identisch m_{post} ist oder aber kein 'ref' bzw. 'proc' mehr vorhanden ist. Die benötigten Anpassungsoperationen werden in einem Vektor von Prozeduradressen

/1 : flex 0/ ref proc anpass gespeichert. Dieser Vektor stellt das Teilresultat R3 dar. Im Modusbaum entspricht das Entreferenzieren und Entprozedurieren einem Abstieg nach unten (down).



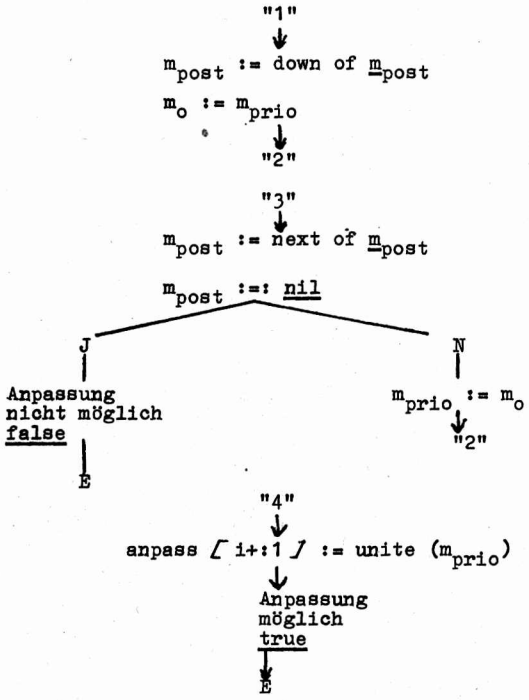
Liegt ein Vereinigungsmodus beim m_{post} vor, so müssen alle seine Bestandteile durchgeprüft werden, bis einer mit möglicher fester Anpassung gefunden wird. Sollte keiner enthalten sein, so verläuft das Verfahren negativ. Das Weiterschalten auf einen neuen Bestandteil geschieht durch

m_{post} := next of m_{post}.

Den ersten Bestandteil findet man durch

m_{post} := down of m_{post}.

Der m_{prio} muß bei jedem Weiterschalten wieder auf den Ursprungsstand gebracht werden.



Der Vermerk (m_{prio}) bei der Anpassungsoperation "unite" bedeutet das Anfügen einer Moduskennzeichnung, nämlich für den nach etlichen vorangegangenen "deref" und "deproc" verbliebenen m_{prio}, an den Operandenwert, so daß die anzusprechende identifizierte Operation den vorliegenden aktuellen Fall erkennt und einen entsprechenden Zweig in der Konformitätsklausel (bestimmt in der Routine der Operation vorhanden) auswählen kann.

Nach der Syntax des Revidierten ALGOL 68 ist es nicht möglich, mehrere anpaßbare Modi in einem Vereinigungsmodus zu finden. Es dürfen nämlich keine Bestandteile eines solchen Modus in fester Verwandtschaft stehen.

9. Balance

Eine besondere Problematik resultiert aus dem Verwenden bedingter Klauseln als Operanden oder auch abgeschlossener Klauseln, welche serielle Klauseln mit mehreren Ausgängen umklammern.

Beispiel:

```
  if b then 3 else 3.14 fi + 2.72
 (if b then goto m1 fi; 2 exit m1: 3.14) + 2.72
```

Die ersten Operanden liefern (zunächst entweder einen int- oder einen real-Wert, was sich erst zur Laufzeit entscheidet. Von evtl. vorhandenen Operationsdeklarationen

```
op + = (real a,b) real: (...);
op + = (int a, real b) real: (...);
```

muß aber eine bereits während der Compilierzeit identifiziert werden. Dies geschieht durch die sogenannte Balance. Bedingte oder geklammerte Klauseln sind keine Anpassungsobjekte, sondern dies sind nur ihre Bestandteile. Die Programmposition der bedingten oder geklammerten Klausel, hier "fest", wird nur an einen Bestandteil vererbt. Alle anderen Bestandteile sind "stark". Die Auswahl des Haupterben (der die elterliche Firma oder den elterlichen Hof weiterführt) ist Inhalt der Balance (im Testament). Bestandteile im Sinne der Syntax sind direkte Nachkommen, d.h., in den obigen Beispielen die

then-Klausel (3)
else-Klausel (3.14)

oder der

1. Klauselzug (2)
2. Klauselzug (3.14)

Die "feste" Balance muß so beschaffen sein, daß sowohl durch feste Anpassungen des Haupterben als auch durch starke Anpas-

sungen der Nebenerben stets derselbe Modus erreichbar ist.

An der Stelle eines a priori-Modus für einen Operanden zur Operatoridentifizierung kann also die Syntaxanalyse auch die Angabe

balance (m_1, ..., m_r)

mit einer Liste von Modi für die Bestandteile liefern. Das ist sogar rekursiv zu verstehen, da wiederum einer der m_i eine "balance" sein kann. In diesem Fall werden einfach im Algorithmus die eingeschobenen Modi der Bestandteil-Balance in die Liste der m_i eingefügt, so daß r entsprechend größer wird.

Der Anpassungsalgorithmus wird bei fester Balance erneut modifiziert, da alle m_i als Haupterben durchgeprüft werden müssen bis entweder ein Erfolg vorliegt oder die Liste der m_i erschöpft ist.

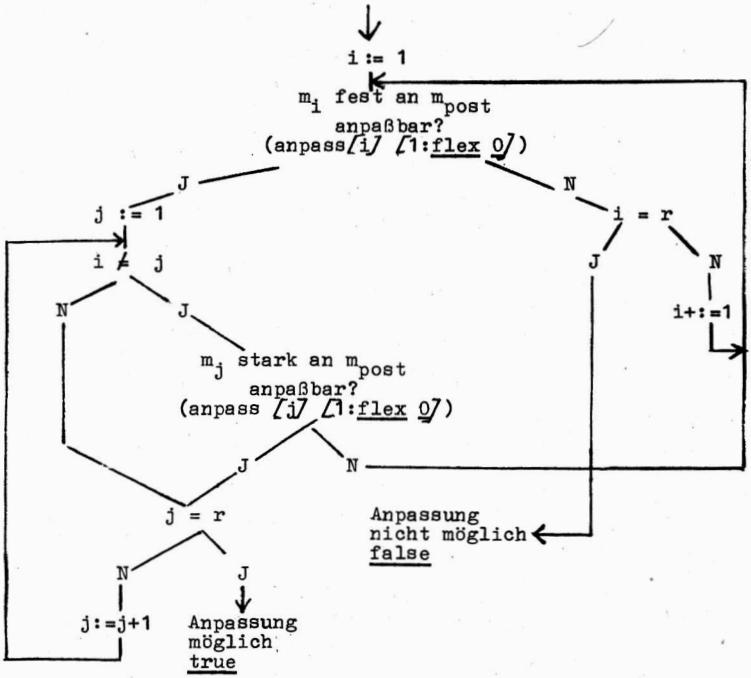
Es entsteht dabei möglicherweise nicht nur ein Vektor von Anpassungen, sondern ein Vektor von Vektoren, für jeden Bestandteil der Balance einer. (s.S.)

Es ist möglich, den größten gemeinsamen Modus der m_i einer Balance in einer Programmposition zu bestimmen und dann von ihm ausgehend die Operatoridentifizierung durchzuführen. Es dürfen dann höchstens noch weitere feste Anpassungen an die "anpass [i] [1:flex 9]" angefügt werden. Der größte gemeinsame Modus einer Balance kann als ihr a priori-Modus angesehen werden. Dieser wird überdies auch gebraucht, um beim Sonderfall das Suchabbruchs die feste Anpaßbarkeit des formalen Operandenmodus an den aktuellen - eben den größten gemeinsamen der Balance - zu prüfen.

Im Beispiel

```
if b then 3 else 3.14 fi + 2.72
```

muß "fest" an 3.14 vererbt werden, denn nur durch die starke Anpassung "wide" (von 'int' zu 'real') kann aus der 3 eine 3.0 erzeugt werden. Der größte gemeinsame Modus ist 'real'.



Im Beispiel

if b then i else x fi + 2.72

muß "fest" an x vererbt werden, denn nur durch die starken Anpassungen "deref" und "wide" kann der Wert von i ein real-Wert werden. Der größte gemeinsame Modus ist wieder 'real'.

Im Beispiel

if b then x else 3.14 fi + 2.72

kann "fest" an x (dies wird durch den Algorithmus auch ausgewählt, d.h. als erster möglicher Erbe gefunden) aber auch an 3.14 vererbt werden. Die Anpassung "deref" für x ist fest, aber auch stark. Semantisch ist diese Balance eindeutig, wenn

auch syntaktisch die Erbschaft nicht eindeutig ist. Der größte gemeinsame Modus ist 'real'.

10. Reihen und Strukturen als Operanden

Im Prinzip bestehen bei der Operatoridentifizierung für Reihen oder mehrfache Werte, wie z.B. Vektoren oder Matrizen, als Operanden und ebenso für Strukturen oder Datensätze (records) keinerlei Probleme. Es können aber die Werte für Reihen und Strukturen auch als kollaterale Klauseln oder wie man dann sagt, als Reihen- oder Struktur-Displays dargestellt werden.

```
[1:3]int i1 := (11,12,13), i2 := (21,22,23);
mode person = struct (string name, int alter, ref person
                                                  partner);
person otto := ("otto", 34, nil), anna := ("anna", 29, nil);
mode gitter =struct (int x,y,z);
gitter p := (11,12,13), q := (21,22,23);
```

Die Formeln

```
i1 + i2
otto + anna
p + q
```

bereiten keine Schwierigkeiten. Es werden Operationen für die Vektoraddition bzw. für die Eheschließung (Eintragen der partner-Referenzen) bzw. für Gitterpunkte identifiziert. Problematisch würde es dagegen bei

```
(11,12,13) + (21,22,23);
```

werden, da der Modus 'row of int' oder 'gitter' nicht erkannt werden kann. Im Revidierten ALGOL 68 wird auf ganz einfache Weise, so wie Alexander den gordischen Knoten zerhieb, festgelegt, daß Reihen- und Struktur-Displays nur in starken Programmpositionen zugelassen sind. Als Operanden sind sie somit verboten. Der genannte Fall kann nicht eintreten.

Literaturverzeichnis

Branquart, P.u.a. An optimized translation process and its application to ALGOL 68 (Brüsseler Imple-

mentierung)
MBLE Laboratoire de Recherche, Report R 204, 1974

Giersich, H., Kerner, I.O.
Operatoridentifizierung im Revidierten ALGOL 68 - Forschungsberichte der Rostocker Arbeitsgruppe "Programmiersprachen", Wilhelm-Pieck-Universität Rostock, Sektion Mathematik, 1975

Kerner, I.O., Külzer, K., Riedewald, G.
Vorprozessor für die Rostocker Implementierung Revidiertes ALGOL 68-ESER
Forschungsberichte der Rostocker Arbeitsgruppe "Programmiersprachen", Wilhelm-Pieck-Universität Rostock, Sektion Mathematik, 1976

Kolbig, W., Lorenzen, H.-P.
Prüfung auf feste Anpaßbarkeit im Revidierten ALGOL 68
Forschungsberichte der Rostocker Arbeitsgruppe "Programmiersprachen", Wilhelm-Pieck-Universität Rostock, Sektion Mathematik, 1976

Raguse, R., Lüdtke, H.H.
Eintragungen in die Modusliste für die Rostocker Implementierung
Revidiertes ALGOL 68-ESER
Forschungsberichte der Rostocker Arbeitsgruppe "Programmiersprachen", Wilhelm-Pieck-Universität Rostock, Sektion Mathematik, 1976

Rodger, W.W.
THINK - A Biography of the Watsons and the IBM
Stein and Day, Publishers, New York 1969

van Wijngaarden, A.u.a. Report on the Revised ALGOL 68
Acta Informatica 5 (1975) 1-3, 1-236

Wössner, H. Operatoridentifizierung in ALGOL 68
Dissertation TU München, Institut für Mathematik, 1970

eingegangen: 23. 4. 1976

Anschrift des Verfassers

Doz. Dr.sc.nat. I.O. Kerner
Wilhelm-Pieck-Universität Rostock
Sektion Mathematik
DDR 25 Rostock
Universitätsplatz 1