KERNFORSCHUNGSZENTRUM

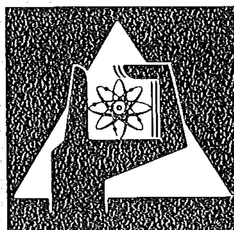
KARLSRUHE

September 1975 KFK 2204

Institut für Reaktorentwicklung

Definition, Übersetzung und Anwendung benutzerorientierter Sprachen als Erweiterung von PL/1 in dem System für das rechnerunterstützte Entwickeln und Konstruieren REGENT

G. Enderle



GESELLSCHAFT
FÜR
KERNFORSCHUNG M.B.H.

KARLSRUHE

Als Manuskript vervielfältigt

Für diesen Bericht behalten wir uns alle Rechte vor

GESELLSCHAFT FÜR KERNFORSCHUNG M. B. H.

KARLSRUHE

KERNFORSCHUNGSZENTRUM KARLSRUHE

KFK 2204

Institut für Reaktorentwicklung

Definition, Übersetzung und Anwendung benutzer-orientierter Sprachen als Erweiterung von PL/1 in dem System für das rechnerunterstützte Entwickeln und Konstruieren REGENT *)

Günter Enderle

Gesellschaft für Kernforschung mbH., Karlsruhe

*) Als Dissertation genehmigt von der Fakultät für Maschinenbau der Universität Karlsruhe (TH)

Kurzfassung :

Das intergrierte System für das rechnergestützte Entwickeln und Konstruieren REGENT dient zur modularen Erstellung von Programmen für die verschiedenen Anwendungsgebiete, zur Steuerung von Datenhaltung und Datenaustausch zwischen Programmen und zur leichten und sicheren Anwendung der Programme durch benutzerorientierte Sprachen. Programme , Daten und Sprache für einen Problembereich bilden ein REGENT-Subsystem.

Die REGENT-Sprachverwaltung PLS (Problem Language System) ermöglicht die Entwicklung von anwendungsbezogenen, subsystemspezifischen Sprachen als Erweiterung der Basissprache PL/1. In dieser Arbeit werden die Übersetzung von Anwendersprachen durch einen Vorcompiler in PL/1 und die Definition von Spracherweiterungen und von Subsystem-Datenstrukturen beschrieben. Die Entwicklung und Anwendung der Steuersprache für ein Fluiddynamik-Subsystem werden gezeigt.

Abstract :

Definition, Translation and Application of User-oriented Languages as Extensions of PL/1 in the CAD-System REGENT

The integrated CAD-system REGENT serves for modular development of programs for different application areas, for management of data storage and data transfer and easy and safe application of programs by means of user-oriented languages. Programs, data and language for an application area form a REGENT-subsystem.

The problem language system of REGENT, PLS (Problem Language System), provides facilities for the development of a problem oriented language for every subsystem as extensions of the base language PL/1. In this paper the translation of problem oriented languages by a precompiler into PL/1 and the definition of language extensions and datastructures for subsystems are described. Development and application of the language for a fluiddynamics-subsystem is shown.

Inhaltsverzeichnis

Seite

Kurzfassung

1. Einleitung 1
   1.1 Höhere Programmiersprachen 1
   1.2 Problemorientierte Sprachen 4
2. Das integrierte CAD-System REGENT 10
   2.1 Integrierte CAD-Systeme 10
   2.2 REGENT 12
3. Entwurfsgrundlagen der REGENT-Sprachverwaltung 15
   3.1 Problemorientierte Sprachen als PL/1-Erweiterung 16
   3.2 Interpretation und Kompilation problemorientierter Sprachen 18
   3.3 Grundstruktur des PLS-Übersetzers 22
   3.4 Sprachdefinition 26
4. Realisierung der REGENT-Sprachverwaltung 31
   4.1 Übersetzung problemorientierter Sprachen 31
      4.1.1 Ablauf der POL-Übersetzung 31
      4.1.2 Aufruf und Abschluß von Subsystemen 34
      4.1.3 Behandlung von Subsystemdatenstrukturen 38
      4.1.4 Behandlung von Fehlern zur Übersetzungszeit 38
   4.2 Definition problemorientierter Sprachen 41
      4.2.1 Das REGENT-Subsystem PLS 41
      4.2.2 Steuerung der Sprachdefinition 42
      4.2.3 Beispiel einer Anweisungsdefinition 44
   4.3 Interaktive Definition und Anwendung von POLs 46
      4.3.1 Zusammenstellen von Stapelaufträgen an der Datenendstation 46
      4.3.2 Interaktive Programmentwicklung 47
      4.3.3 Anforderung von Eingabedaten 48
5. Subsystem-Sprachentwicklungen mit PLS 50
   5.1 Fähigkeiten des REGENT-Subsystems REMAC 50
   5.2 Schritte der Subsystementwicklung 52
   5.3 Entwicklung der REMAC-Steuersprache 54
   5.4 Anwendung der REMAC-Sprache 58
   5.5 Andere REGENT-Subsysteme 71

Seite

6. Effektivitätsbetrachtung 75

7. Zusammenfassung der wichtigsten Ergebnisse 78

Anhang:

A Verwendete Syntaxnotation 81

B PLS-Syntaxbeschreibung 84

B1 Systemanweisungen 84

B2 PLS-Funktionen 89

B3 PLS-Anweisungen 93

B4 Makrozeitanweisungen 104

C Zur Implementierung 107

C1 Implementierung des PLS-Übersetzers 107

C2 Initialisierung und Zerstörung von Subsystemen 114

C3 Definition von POL-Anweisungen und Datenstrukturen 118

D Zeichenklassen und Symboltypen 136

E Automatendiagramm der Symbolentschlüsselung 138

F REMAC-Sprachspezifikation 142

Literatur 150

1. Einleitung

Durch die Steigerung der Rechengeschwindigkeit und der Arbeitsspeichergröße elektronischer Datenverarbeitungsanlagen (DVA) bei gleichzeitiger starker Reduzierung des Preis/Leistungsverhältnisses sind für eine immer größere Anzahl von Problembereichen effektive und rationelle Problemlösungen durch Hilfsmittel der Datenverarbeitung möglich geworden. Immer häufiger sind es nicht DV-Spezialisten, sondern gerade im technisch-wissenschaftlichen Bereich Konstrukteure, Ingenieure und Wissenschaftler, die ihre Problemlösungen mit Hilfe der Datenverarbeitung erzielen. Dadurch gewinnt die Schnittstelle bei der Kommunikation zwischen Mensch und Rechner entscheidende Bedeutung - über diese Schnittstelle wird dem Rechner neben numerischen und alphabetischen Daten ein maschinenlesbares Modell des Problems und (meist) des Lösungsweges mitgeteilt (Eingabe) und die Resultate der Rechnung zurückgegeben (Ausgabe). Die Eingabe von Modellbeschreibung und Lösungsweg erfolgt dabei über eine von der DVA verarbeitete Sprache, wobei man jede Eingabemöglichkeit, von der Betriebssystem-Steuersprache über das Zeigen mit dem Lichtgriffel auf einem Bildschirm bis hin zu den Kommandos eines Operateurs als Sprache bezeichnen kann. Die Eingabe mittels "höherer Programmiersprachen", wie FORTRAN, ALGOL oder PL/1 und die Eingabe mittels "problemorientierter Sprachen" werden hier näher betrachtet.

1.1 Höhere Programmiersprachen

Die Programmiersprachen FORTRAN, ALGOL und PL/1 sind vor allem für den Bereich arithmetischer Aufgabenstellungen, also für Berechnungen, ein geeignetes Hilfsmittel. Da ein weiter Bereich von Problemstellungen einer Lösung durch Anwendung arithmetischer Operationen zugänglich ist, arithmetische Probleme als erste mit Hilfe der Datenverarbeitung gelöst werden konnten und sich eine DVA für Berechnungen besonders effektiv einsetzen läßt, haben diese Sprachen eine weite Verbreitung gefunden und die Bezeichnung "allgemein verwendbare höhere Programmiersprachen" (general purpose higher level languages) erhalten. Diese Sprachen verdanken ihre Verbreitung nicht allein der Tatsache, daß ein weiter Bereich von Aufgabenstellungen durch sie abgedeckt wird - jede Aufgabe, die mit einem FORTRAN-Programm gelöst wird, läßt sich auch mit einem

Programm in Assemblersprache oder Maschinencode lösen -, sondern ihrer leichten und sicheren Anwendung, vor allem der weitgehenden Übereinstimmung der Sprache der Arithmetik (Formeln) und der Programmiersprache. Während kaum ein Ingenieur den beschwerlichen Weg der Programmierung seiner Probleme in Assemblersprache ging, sind Programmiersprachen wie FORTRAN oder (neuerdings in steigendem Maße) PL/1 zu einem selbstverständlichen Werkzeug im technisch-wissenschaftlichen Bereich geworden.

In zunehmendem Maße wurden jedoch immer mehr nichtnumerische Probleme einer Behandlung durch DVA zugänglich, darunter geometrische, topologische und grafische Probleme, Simulation und Textverarbeitung. Wenn diese Bereiche durch höhere Programmiersprachen abgedeckt werden sollen, reichen die Elemente der Sprache selbst nicht mehr zur Beschreibung von Modell und Lösungsweg aus. Daher werden Modellbeschreibung und Ablaufsteuerung durch Unterprogrammaufrufe oder durch vom Programm eingelesene und interpretierte Konstanten vorgenommen.

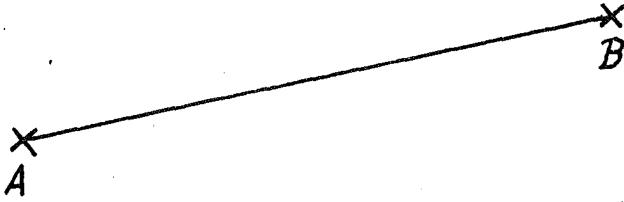
Einlesen von Konstanten

Von einem Eingabemedium (Karten, Lochstreifen, Tastatur) werden Eingabeparameter für ein Programm eingelesen (numerische Konstanten, Zeichenketten). Da an dem Programm selbst nichts mehr geändert wird, besteht die Anwendersprache aus diesen Eingabedaten. "Sprache" ist hier Bezeichnung für die Eingabe für einen Rechner. Es handelt sich allerdings um eine sehr primitive, schwer lesbare Sprache, was ihre Anwendung unbequem, umständlich und zeitraubend macht. Variable und arithmetische Ausdrücke (die gerade mit für die Attraktivität höherer Programmiersprachen verantwortlich sind) sind nicht zulässig. Die Eingabe ist oft formatgebunden, die Reihenfolge der Daten ist starr. Ohne Erläuterung ist die Sprache nicht verständlich, daher fehleranfällig und zeitaufwendig zu kodieren und zu ändern.

Um die verschiedenen Möglichkeiten der Mitteilung einer Problemstellung an die DVA zu verdeutlichen, soll die Zeichnung einer Linie als Beispiel dienen. Ein Ingenieur möchte der DVA mitteilen:

"Zeichne eine Linie von A nach B"

Er erwartet folgendes Ergebnis:



Die Koordinaten von A seien (1.5 , 3), von B (7.5 , 6). Beim Angeben des Zeichenbefehls durch Einlesen von Konstanten könnte die Eingabe wie folgt aussehen:

```
Z     2
1.5   3   7.5   6
```

"Z" soll für "Zeichne Linienzug" stehen, die 2 gibt die Anzahl der Punkte an.

Unterprogrammaufrufe

Alle allgemein verwendbaren höheren Programmiersprachen stellen die Möglichkeit zur Verfügung, Unterprogramme aufzurufen. Das Bereitstellen von Prozeduren für einzelne Anwendungen oder von Unterprogrammpaketen für Anwendungsgebiete ist daher eine der meistbenutzten Techniken, um besondere Fähigkeiten in höheren Programmiersprachen verfügbar zu machen. Die Ablaufsteuerungen der Programmiersprache sind weiterhin verfügbar (Schleifen, Abfragen). Den Unterprogrammen können über Argumentlisten Programmparameter übergeben werden, die Argumente können auch Variable und arithmetische Ausdrücke sein. Der Nachteil von Unterprogrammaufrufen ist die starre Syntax (Prozedurname und Argumente), die Unübersichtlichkeit bei mehr als drei bis fünf Argumenten und die Tatsache, daß die Bedeutung der Argumente nicht ohne weiteres ersichtlich ist.

Der Befehl: "Zeichne eine Linie von A nach B" sieht bei Verwendung von Unterprogrammaufrufen z.B. folgendermaßen aus:

```
X(1)=1.5
X(2)=7.5
Y(1)=3
Y(2)=6
CALL LINE(X,Y,2)
```

Aus den Nachteilen der Benutzung von Sprachen mit hauptsächlich numerischen Fähigkeiten für nichtnumerische Probleme entsprang sehr bald die Erkenntnis, daß die "allgemein verwendbaren höheren Programmiersprachen" in Wirklichkeit Spezialsprachen für den arithmetischen Bereich sind, wie der Name FORTRAN (Abkürzung für: FORMULA TRANSLATOR) schon andeutet. Eine Sprache für alle Probleme zu entwickeln erwies sich als unmöglich. Für viele nichtnumerische Problembereiche wurde daher eine große Anzahl von Sprachen geschaffen. Da die Sprachen auf die jeweiligen Probleme zugeschnitten sind, wurden sie problemorientierte Sprachen (problem oriented language, POL) genannt. Die wichtigsten Arten problemorientierter Sprachen und ihre Vor- und Nachteile werden im folgenden Abschnitt aufgeführt.

1.2 Problemorientierte Sprachen

Problemorientierte Sprachen, benutzerbezogene oder anwendungsorientierte Sprachen sind formale Sprachen, deren Sprachelemente und Sprachstrukturen weitgehend an die in bestimmten Fachgebieten oder Problemkreisen gebräuchliche Terminologie angepaßt sind. Die Beschreibung einer Problemstellung und des Lösungsweges in einer solchen Sprache hat gegenüber der Programmierung in höheren Programmiersprachen folgende Vorteile /1,2,3/:

1) Da die POL-Terminologie der Fachsprache angepaßt ist, ist sie leicht erlernbar.

2) Der Programmierer kann sich mehr auf die Beschreibung von Problem und Problemlösung konzentrieren und wird weniger von programmiertechnischen Problemen abgelenkt. Dies führt zu kürzeren Programmier- und Testzeiten.

3) Ein Programm kann auch von einem Fachmann gelesen und in seiner Wirkungsweise verstanden werden, der die POL-Syntax nicht kennt. Die Modellbeschreibung ist gleichzeitig die maschinenlesbare Darstellung des Problems.

4) POL-Programme sind wegen ihrer leichten Lesbarkeit weitgehend selbstdokumentierend. Eine Überprüfung der Eingabe für eine Rechnung ist leichter möglich.

Von den verschiedenen Arten problemorientierter Sprachen sind vor allem Kommandosprachen, eigenständige höhere Programmiersprachen und Erweiterungen allgemein verwendeter höherer Programmiersprachen hervorzuheben.

Kommandosprachen

Viele problemorientierte Sprachen im technisch-wissenschaftlichen Bereich sind Kommandosprachen. Bei Kommandosprachen werden einzelne ausführbare Anweisungen (Kommandos) sequentiell abgearbeitet und nacheinander ausgeführt. Parameter müssen als Konstante angegeben werden, Variable und arithmetische Ausdrücke sind nicht zulässig. Das Meßdatenauswertesystem SEDAP /4,5/ wird durch eine formatgebundene Kommandosprache gesteuert, Beispiele für nicht formatgebundene Kommandosprachen sind die Sprachen für die Strukturanalyse STRUDL /6/ und DYNAL /7/ und für die Netzplantechnik PROJECT /8/. Hauptnachteile der Kommandosprachen sind das Fehlen von Variablen und arithmetischen Ausdrücken und von Möglichkeiten zur Ablaufsteuerung wie Schleifen, Sprünge und bedingte Anweisungen. Da die Kommandos sequentiell verarbeitet werden, werden Syntaxfehler erst erkannt, wenn das Kommando ausgeführt werden soll. Dadurch muß dann u.U. eine Rechnung ohne Ergebnisse abgebrochen werden, die schon erhebliche Kosten verursacht hat. Nach dem Urteil der Anwender ist dies einer der Hauptmängel des Systems SEDAP.

Die Sprache GRAPHIC /9,10,11,12,13/ für die Erzeugung, Manipulation und Ausgabe grafischer Information ist ebenfalls eine Kommandosprache, jedoch wurde durch Abspeichern der Kommandos in einer internen Datenstruktur die Möglichkeit geschaffen, Unterprogramme, Schleifen, bedingte Anweisungen und Variable zu benutzen. Die Implementierung von Unterprogrammen und Schleifen innerhalb eines interpretierenden Systems hatte eine so starke Verminderung der Effektivität zur Folge, daß eine weite Anwendung des GRAPHIC-Systems vor allem an Kostengründen scheiterte.

In GRAPHIC lautet die Aufforderung, eine Linie von A nach B zu zeichnen:

```
ZEICHNE LINIE VON PUNKT ( 1.5,3 ) NACH PUNKT ( 7.5,6 )
```

oder, wenn die Punkte schon vorher definiert werden:

```
SETZE A = PUNKT ( 1.5,3 )
SETZE B = PUNKT ( 7.5,6 )
ZEICHNE LINIE VON A NACH B
```

Dieses letzte Kommando ist fast die wörtliche Wiedergabe der Problemstellung "Zeichne eine Linie von A nach B", dies stellt ein Idealfall dar bei der Verwendung einer problemorientierten Sprache.

Eigenständige höhere Programmiersprachen

Für bestimmte Anwendungen wurden eigenständige höhere Programmiersprachen entwickelt. Als Beispiele können EXAPT /14/, eine Sprache zur Programmierung numerisch gesteuerter Werkzeugmaschinen, GPSS (General Purpose Simulation System) /15/ zur diskreten Simulation, SNOBOL /16/ zur Zeichenkettenverarbeitung oder ECAP /17/ für die Netzwerkanalyse dienen. In diesen Sprachen überschneiden sich viele Grundfähigkeiten (Variable, Ausdrücke, einfache Arithmetik, Unterprogramme, Sprünge) und Bestandteile des Übersetzers (Syntaxanalyse, Codegenerierung). Da für jede Sprache auch die Grundbestandteile neu implementiert werden müssen, ist der Aufwand zur Erstellung eines Übersetzers beträchtlich, er lohnt sich nur für die wenigen Problembereiche, für die es viele verschiedene Anwender gibt.

Erweiterung bestehender Programmiersprachen

Eine weitere vielbenutzte Möglichkeit zur Erstellung problemorientierter Sprachen ist die Erweiterung bestehender höherer Programmiersprachen um Datentypen, Operationen und Anweisungen für einen Anwendungsbereich. Die vollen Fähigkeiten der Grundsprache (Variable, arithmetische Ausdrücke, Unterprogramme, bedingte Anweisungen) bleiben so erhalten. Der Implementierungsaufwand für Spracherweiterungen ist meist geringer als der Aufwand für völlig neu entwickelte Sprachen. Karl Soop, der eine grafische Sprache als PL/1-Erweiterung entwickelt hat /18/, schreibt zu der Technik der Spracherweiterung: "Reinventing the wheel is avoided". Außerdem müssen Anwender, denen die Grundsprache geläufig ist, nur die Erweiterungen neu erlernen.

Es gibt zahlreiche Beispiele für Spracherweiterungen für bestimmte Fachgebiete:

GPL/1 (Graphical PL/1, /19/) ist ebenso wie
GRAPL/1 (Graphical PL/1, /18/) eine grafische PL/1-Erweiterung.
PL/1-FORMAC /20/, eine Sprache für formale Arithmetik, ist als PL/1-Erweiterung verfügbar.
SIMSCRIPT /21/ und CSMP /22/ sind Sprachen zur diskreten und kontinuierlichen Simulation und basieren auf FORTRAN.
EPL/1 (Extended PL/1, /25/ ist eine diskrete Simulationssprache auf der Grundlage von PL/1.
GRAF (Graphical FORTRAN, /24/) ist eine grafische FORTRAN-Erweiterung.

Bei einigen höheren Programmiersprachen ist die Fähigkeit zu ihrer Erweiterung Teil der Sprache. Die Erweiterung kann sich auf die Datenstrukturen beziehen und auf die Operationen, die mit diesen Daten möglich sind (ALGOL68,Simula). In PL/1 kann durch Verwendung des Makroprozessors eine Erweiterung um zusätzliche Anweisungen oder Teilanweisungen vorgenommen werden. Die Nachteile der auf diese Weise erzeugten Spracherweiterungen sind ihre starre Syntax (Operator-Operanden- oder Prozeduraufruf- Syntax) und die beschränkten Möglichkeiten, Spracherweiterungen dauernd zu speichern. In den meisten Fällen müssen die Definitionen der Erweiterungen bei jeder Anwendung der Sprache wieder neu mitkompiliert werden /25/.

Aufgrund von Erfahrungen mit der Anwendung und Entwicklung problemorientierter Sprachen /9,26,27,28/ und den Vor- und Nachteilen, die die Kommandosprachen, die eigenständigen höheren Programmiersprachen und die Spracherweiterungen haben, wurden folgen Anforderungen an Sprachen im CAD-Bereich aufgestellt:

- Eine problemorientierte Sprache soll der Fachsprache, die in dem betreffenden Fachgebiet verwendet wird, möglichst weitgehend angepaßt sein. Der Lernaufwand wird dadurch reduziert, ein Fachmann kann ohne genaue Sprachkenntnisse ein POL-Programm lesen.

- Die Eingabe soll nicht formatgebunden sein. Die Handhabung der Sprache wird dadurch erleichtert und Fehlermöglichkeiten ausgeschaltet.

- Zur Programmablaufsteuerung sollen Schleifen, bedingte Anweisungen und Unterprogramme möglich sein. Oft vorkommende Aufgabenstellungen können dann einmal in Form eines POL-Unterprogramms programmiert und bei Bedarf aufgerufen werden. Iterationen und Steuerung des weiteren Vorgehens aufgrund von vorher berechneten Ergebnissen sind möglich.

- Anstelle von Konstanten sollen in den POLs Variable, indizierte Variable und arithmetische Ausdrücke verfügbar sein. Parametervariationen und Zwischenrechnungen, die der Spracherstellar nicht vorsehen konnte, können dann vorgenommen werden.

- Die problemorientierten Sprachen sollen Ein-/Ausgabe-Anweisungen enthalten. Zum Einlesen von Parametern von externen Einheiten (z.B. Meßdaten), zum Speichern von Ergebnissen oder Zwischenergebnissen und zur Ablaufkontrolle mittels Druckanweisungen sind Ein-/Ausgabe-Anweisungen in den POLs unumgänglich.

- Die Anwendung von POLs soll sowohl interaktiv von einem Datenendgerät als auch im Stapelbetrieb möglich sein.

- Bei fehlerhafter Anwendung von POLs sollen ausreichende, lesbare Fehlermeldungen erfolgen.

- Da man nicht erwarten kann, daß eine problemorientierte Sprache für viele Problembereiche ausreicht und wegen der ständig zunehmenden Vielfalt von Aufgabenstellungen, die einer Problemlösung mit Hilfe einer DVA zugänglich sind, muß eine leicht handhabbare und flexible Möglichkeit zur Definition neuer und zur Modifikation bestehender problemorientierter Sprachen vorhanden sein. Die Sprachdefinition soll auch von einem Ingenieur des jeweiligen Fachgebietes ohne weitreichende Informatik-Ausbildung vorzunehmen sein.

- Für die Anwendung der Sprachen muß ein effektiver Übersetzer bereitgestellt werden.

Der Grundgedanke dieser Forderungen ist, daß durch die Verwendung problemorientierter Sprachen eine leicht und flexibel handhabbare Anwenderschnittstelle für die Benutzung einer DVA geschaffen werden soll, die dem Idealfall:

"Problemstellung = maschinenlesbare Darstellung"
möglichst nahe kommt. Verschiedene Stufen der Anpassung der maschinenlesbaren Eingabe wurden anhand der Problemstellung "Zeichne eine Linie von A nach B" gezeigt:

```
(1)  Z    2
     1.5  3  7.5  6

(2)  CALL LINE(X,Y,2)

(3)  ZEICHNE LINIE VON A NACH B
```

Erst in der letzten Form, bei Verwendung einer problemorientierten Sprache, ist die (fast wörtlich) wiedergegebene Problemstellung gleichzeitig die maschinenlesbare Eingabe für das Zeichenprogramm.

Die in dieser Arbeit beschriebene Sprachverwaltung des Systems REGENT erfüllt die obigen Forderungen an CAD-Sprachen und realisiert die geforderte benutzerfreundliche Schnittstelle zur Anwendung komplexer Programmsysteme.

2. Das integrierte CAD-System REGENT

Bei der Behandlung komplexer Probleme mit Hilfe der Datenverarbeitung kann nicht mehr für jedes Spezialproblem ein Spezialprogramm erstellt werden (stand alone codes). Für jede neue Geometrieanordnung wird z.B. nicht jedesmal ein neues Finite-Elemente-Programm erstellt werden und es kann nicht für jedes Simulationsmodell ein vollständiges neues Simulationsprogramm entwickelt werden. Deshalb werden Programmsysteme benutzt, die einen größeren Teilbereich abdecken. Für einen Anwender eines solchen Systems ist die Benutzerschnittstelle von entscheidender Bedeutung, also die Eingabesprache (=problemorientierte Sprache) und die Ausgabe der Ergebnisse. Der Ersteller eines solchen Systems hat aber neben der Realisierung der Anwenderschnittstelle auch das Problemlöseprogramm selbst zu entwickeln. Neben der Lösung der eigentlichen ingenieurtechnischen Probleme bei den Umsetzung eines physikalischen Modells in ein Rechenprogramm sind dabei in großem Maße auch datenverarbeitungsspezifische Probleme, wie Wahl einer geeigneten Datenstruktur oder Ablaufsteuerung einer großen Anzahl von Teilprogrammen in einem begrenzten Arbeitsspeicherbereich zu lösen.

Als Hilfsmittel für die Erstellung und Anwendung komplexer Programmsysteme wurden integrierte Systeme entwickelt. Nach einer Übersicht über existierende integrierte Systeme wird in diesem Kapitel das im Institut für Reaktorentwicklung entwickelte System REGENT kurz vorgestellt, um die Einbettung dieser Arbeit in das Gesamtvorhaben "System für das rechnerunterstützte Entwickeln und Konstruieren" aufzuzeigen.

2.1 Integrierte CAD-Systeme

Integrierte Systeme für das rechnergestützte Entwickeln und Konstruieren (Computer Aided Design, CAD) sollen dem Ingenieur die Anwendung elektronischer Datenverarbeitungsanlagen erleichtern /1/. Zusätzlich zu den in höheren Programmiersprachen vorhandenen Möglichkeiten bieten sie weitere Fähigkeiten und ermöglichen den leichten Gebrauch von Betriebssystemhilfsmitteln, die sonst nur dem Assembler-Programmierer

verfügbar sind. In der Hierarchie zwischen der Hardware der DVA als unterster Schicht und dem Benutzer eines Programms soll ein integriertes System eine problembezogene Benutzerschnittstelle realisieren (Abb. 1).

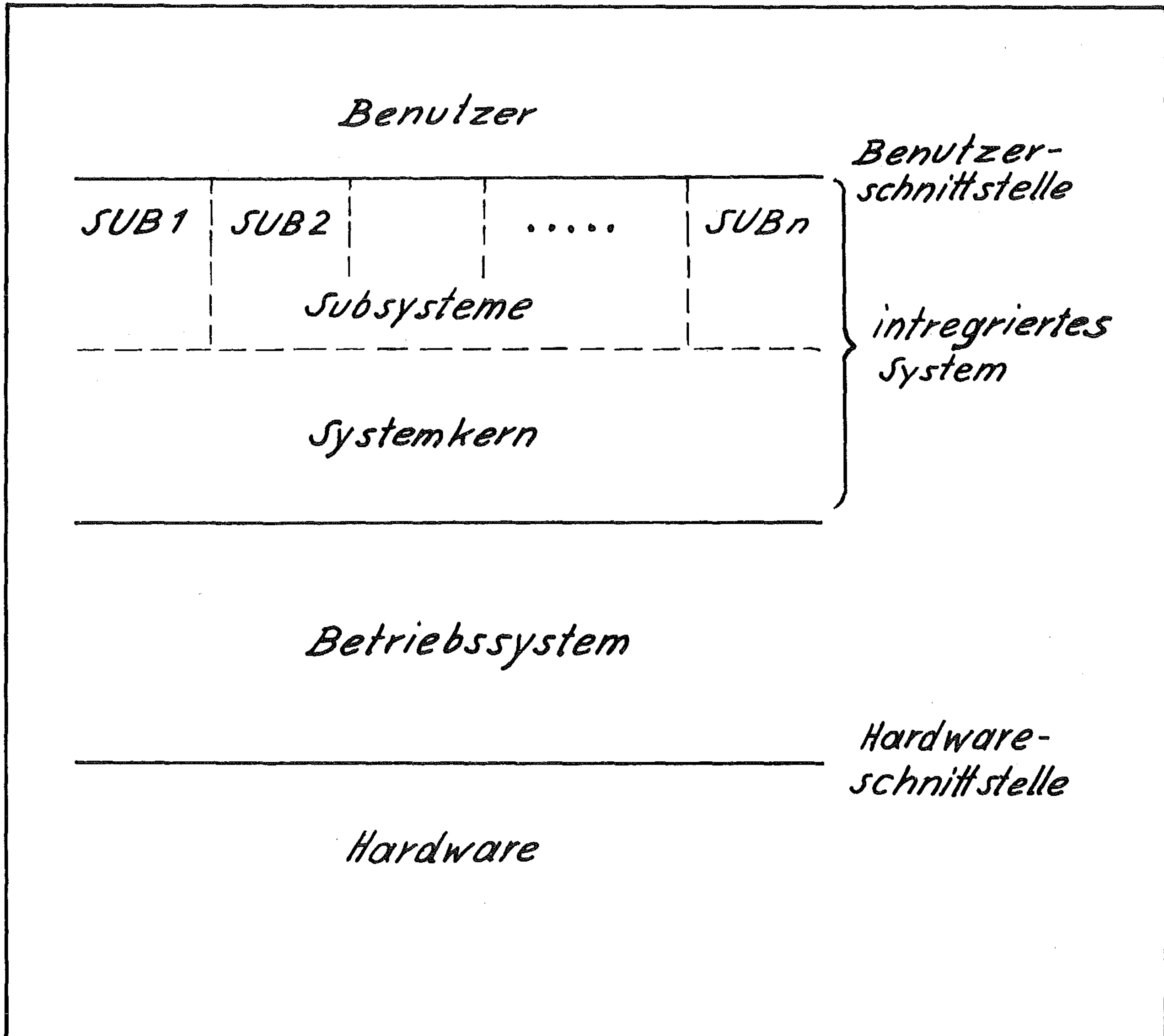


Abb. 1: Programmhierarchie bei Anwendung eines integrierten Systems

Ein Integriertes System besteht aus einem Systemkern und mehreren Subsystemen. Der Systemkern ist das Laufzeitsystem für die Anwendung der Subsysteme, er überwacht und steuert die Abläufe während der Ausführung eines Subsystems.

Folgende Fähigkeiten sind im Systemkern realisiert:

- Dateienverwaltung
- Dynamische Datenverwaltung zur Ausführungszeit
- Modulverwaltung
- Definition und Übersetzung problemorientierter Sprachen
- Testhilfen und Fehlerverwaltung

Von System zu System können einzelne dieser Fähigkeiten mehr oder weniger stark ausgeprägt sein oder ganz fehlen.

Ein Subsystem stellt Fähigkeiten zu Problemlösungen für einen bestimmten Teilbereich zur Verfügung, z.B. ein Netzplantechnik-Subsystem erlaubt die Erstellung, Verwaltung, Änderung und Zeichnung eines Netzplanes, ein Statik-Subsystem die statische Berechnung von Bauwerken.
Ein Subsystem besteht aus:

- Problemlöseprogrammen für das Fachgebiet (Module),
- einer problemorientierten Sprache zur Anwendung des Subsystems,
- Datenstrukturen zur internen Speicherung der Daten und
- Dateien zur langfristigen Datenhaltung.

Eines der meistbenutzten integrierten CAD-Systeme ist ICES (Integrated Civil Engineering System /2,26,29,30/). Die Programmiersprache für Subsystemmodule ist FORTRAN, erweitert um Fähigkeiten für die dynamische Datenverwaltung (Dynamic Arrays) und für den dynamischen Aufruf von Programmodulen von einer Bibliothek. Zur Systemanwendung werden definierbare Kommandosprachen benutzt. Weitere integrierte Systeme im CAD-Bereich sind GENESYS /31/, IST /32,33/, POLO /34/, AED /35/ und SYSFAP /36/.

2.2 REGENT

REGENT (Rechnergestützter Entwurf) /37,38,39,40,41,42,43,44,45/ folgt seinen Vorgängern bezüglich der Gliederung in Kern und Subsysteme und darin, daß Fähigkeiten zur Modulverwaltung, Datenverwaltung und Sprach-

verwaltung verfügbar sind. Auf folgenden Gebieten wurden jedoch Änderungen, Erweiterungen und Verbesserungen vorgenommen:

- Grundsprache zur Erstellung der Subsystemprogramme ist nicht FORTRAN, sondern PL/1. Diese Sprache wurde auch fast ausschließlich zum Programmieren des Systemkerns selbst verwendet.

- Die Subsystem-Steuersprachen enthalten als Grundsprache ebenfalls PL/1.

- Die interaktive Anwendung von REGENT-Subsystemen ist möglich.

- Der Datenaustausch zwischen verschiedenen Subsystemen und das Zusammenwirken mehrerer Subsysteme zur Lösung eines Problems werden erleichtert.

- Parametervariation innerhalb eines Modells ("parametric use", /43/) ist leicht und effektiv möglich.

- Testhilfen und Fehlerbehandlung auch für Subsysteme ist Teil des Systemkerns.

Abb.2 zeigt eine Übersicht über die Teile und Abläufe im REGENT-System. Die mit der Sprachverwaltung zusammenhängenden Teile sind doppelt umrandet. Die Abläufe links von der Mittellinie stellen die Subsystementwicklung dar. Über den Weg: POL-Definition - PLS-Definierer - POL-Statement-Übersetzer wird die Benutzerschnittstelle für ein Subsystem, die problemorientierte Sprache, geschaffen. Der zweite Weg bei der Subsystementwicklung ist die Programmierung der Algorithmen. Nach der Entwicklung des Subsystems ist es auf Bibliotheken (Mitte der Abb.2) in Form von Moduln (ausführbaren übersetzten Programmteilen) und Daten gespeichert. Der rechte Teil der Abbildung zeigt die Subsystem-Anwendung, wobei die Schnittstelle für den Anwender das POL-Programm ist. Auf die Einzelheiten einer Subsystemanwendung wird später eingegangen (Kapitel 5).

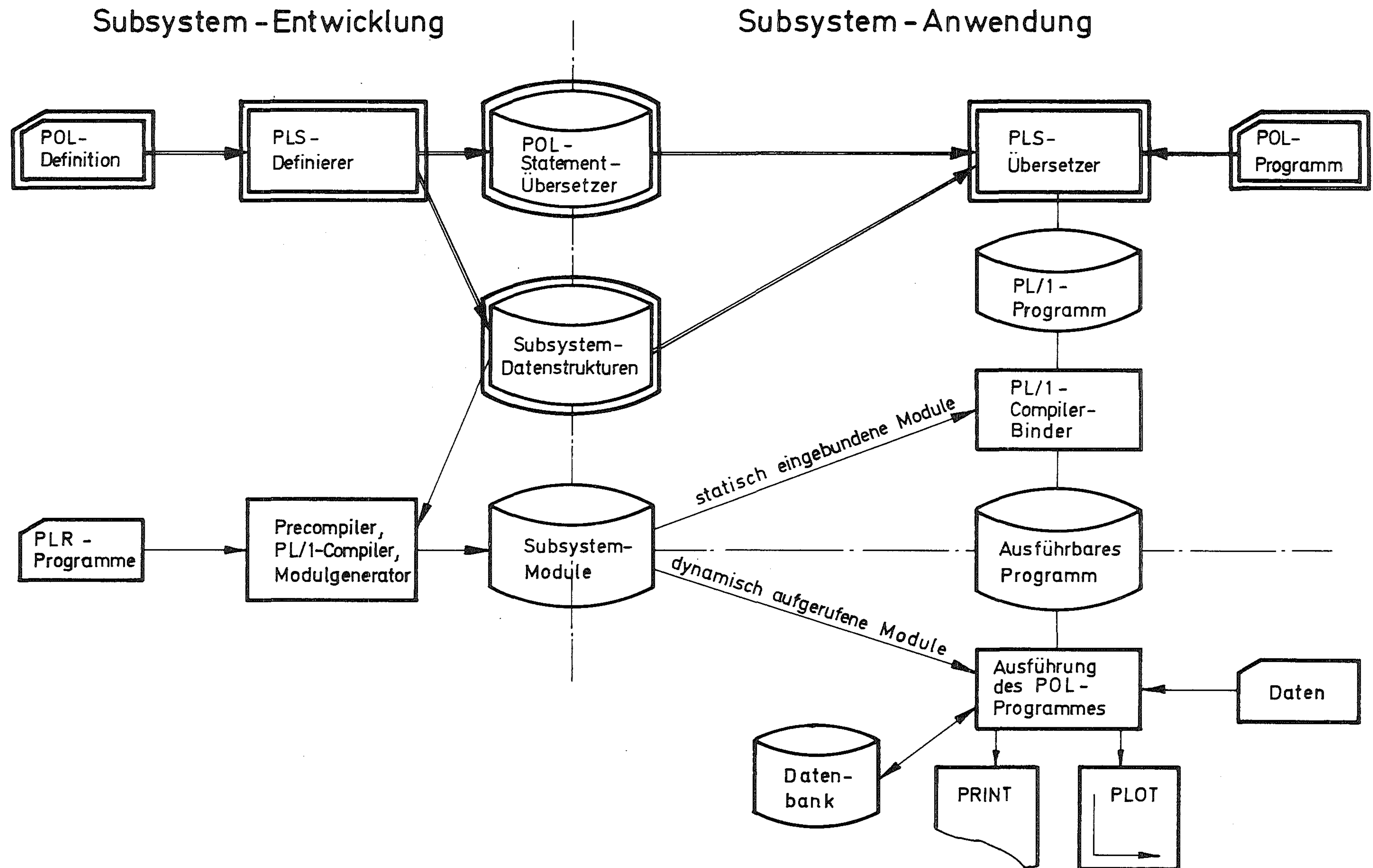


Abb. 2: Anwendung des REGENT - Systems

3. Entwurfsgrundlagen der REGENT-Sprachverwaltung

Das REGENT-Teilsystem PLS (Problem Language System) soll die Definition benutzerorientierter Sprachen zur leichten und sicheren Anwendung von Programmen für Teilbereiche aus dem CAD-Bereich ermöglichen. Die Syntax dieser Sprachen soll flexibel an die im jeweiligen Problembereich gebräuchliche Fachsprache angepaßt werden können. Die Anforderungen an problemorientierte Sprachen, die in der Einleitung aufgestellt wurden, müssen durch PLS realisiert werden. Zusätzlich sind folgende Randbedingungen einzuhalten:

- Die Sprachverwaltung muß sich in das Gesamtsystem REGENT einpassen und mit anderen Systembestandteilen zusammenwirken.

- Da zur Lösung eines Problems mehrere Subsysteme zusammenwirken müssen, muß es möglich sein, von einer Sprache zur anderen beliebig umzuschalten.

Ein Sprachprozessor, der alle Forderungen erfüllt, existierte bisher nicht. Den Kommandosprachen in Systemen wie ICES fehlen die Möglichkeiten, Variable und Ausdrücke, Unterprogramme, Sprünge, Schleifen und Ein-/Ausgabeanweisungen zu verwenden. Die speziellen höheren Programmiersprachen und auch viele Spracherweiterungen sind feste, abgeschlossene Entwicklungen. Sie können nicht an die jeweiligen Problembereiche neu angepaßt werden. Erweiterbare Sprachen und Makroprozessoren mangelt es vor allem an einer flexiblen Syntax, der langfristigen Speicherung von Makrodefinitionen und der Umschaltbarkeit von der Sprache für einen Problembereich auf die Sprache eines anderen Problembereiches. Die speziell auf REGENT abgestimmten Möglichkeiten der Systemsteuerung und -überwachung waren natürlich in bestehenden Sprachprozessoren ebenfalls nicht vorhanden. Es mußte daher ein System zur Definition und Anwendung problemorientierter Sprachen neu entwickelt werden, das alle genannten Forderungen erfüllt.

Das Grundkonzept für diese Sprachverwaltung wird festgelegt durch die Entscheidung über vier Einzelfragen:

- Sind die problemorientierten Sprachen im REGENT-System Kommandosprachen, eigenständige höhere Programmiersprachen oder Erweiterungen einer bestehenden Sprache?

- Werden die problemorientierten Sprachen interpretiert oder kompiliert?

- Wie ist der Übersetzer realisiert, wie liegen die Übersetzungsvorschriften vor, wie erfolgt die Umschaltung zwischen den Sprachen?

- Wie erfolgt die Sprachdefinition - welche Syntax hat die Definitionssprache, wie werden die Sprachdefinitionen analysiert und gespeichert?

Wie diese grundlegenden Fragen entschieden wurden und welches die Gründe dafür waren, wird in den folgenden Abschnitten 3.1 bis 3.4 beschrieben. Zur Realisierung der Sprachverwaltung wurden zum großen Teil bekannte Verfahren aus dem Gebiet des Übersetzerbaus verwendet, neu und ohne Vorbild ist die Kombination: Sprachen mit den genannten hochstehenden Fähigkeiten , Definierbarkeit , langfristige Speicherung und Umschaltbarkeit.

3.1 Problemorientierte Sprachen als PL/1-Erweiterungen

Die Entscheidung über die Frage:

> Sind die problemorientierten Sprachen im REGENT-System Kommandosprachen, eigenständige höhere Programmiersprachen oder Erweiterungen einer bestehenden Sprache?

wurde zugunsten des letzteren entschieden. Bei Kommandosprachen ist es schwierig, Variable, Ausdrücke, Sprüngen. Schleifen, bedingte Anweisungen und Unterprogramme zu realisieren, dies wurde bei dem ICES-Subsystem GRAPHIC sehr deutlich. Bei eigenständigen höheren Programmiersprachen ist der Definitionsaufwand sehr hoch, da jeweils alle sprachlichen Grundfähigkeiten neu definiert werden müssen.

Die problemorientierten Sprachen im REGENT-System sind daher Erweiterungen der Basissprache PL/1. Diese Vorgehensweise hat folgende Vorteile:

- Der mit der Basissprache vertraute Programmierer kann die einzelnen POLs leicht erlernen.

- Eine einheitliche Grundsyntax aller Subsystem-POLs wird erzwungen. Die Erweiterungen müssen sich an die PL/1-Sprachelemente anpassen, die Notation von Kommentaren, numerischen Konstanten, Zeichenketten, Schlüsselworten und Namen ist aus der Basissprache übernommen.

- Der Anwender einer bestimmte POL kann diejenigen PL/1- Fähigkeiten, die er für die betreffende Problemlösung benötigt, neben den eigentlichen POL-Anweisungen benutzen. Es besteht keine Notwendigkeit, aus dem Subsystem herauszugehen und Teilproblemlösungen als Einzelprogramme zu erstellen. Für Standardanwendungen ist dagegen eine genaue Kenntnis der PL/1-Sprache nicht erforderlich.

- Real- und Integervariable und andere Datentypen, PL/1-Strukturen, arithmetische und logische Operatoren, Felder, Unterprogramme, Schleifen und Ein-/Ausgabeoperationen sind in jeder POL von vorneherein vorhanden.

- Da nicht nur die Ausführungsroutinen eines Subsystems in PL/1 geschrieben sind, sondern auch fast alle Teile des REGENT-Kernes einschließlich PLS, ergibt sich eine Sprachkonsistenz für das Gesamtsystem. Die Wartung des Systems wird dadurch erleichtert, der Übertragungsaufwand auf Rechenanlagen anderer Hersteller wird verringert.

PL/1 wurde als Grundsprache für REGENT und somit auch als Basissprache für alle Subsystemsprachen hauptsächlich aus zwei Gründen gewählt: Einmal enthält PL/1 Fähigkeiten (Speicherplatzkontrolle, Listenverarbeitung, Ein-/Ausgabefähigkeiten, Datenaggregate, Zeichenkettenverarbeitung, Fehlerbehandlung, Blockstruktur), die es erlauben, einen Systemkern weitgehend in dieser Sprache zu schreiben und zum zweiten ist PL/1 eine ausreichend weitverbreitete Sprache /46/. PL/1-Übersetzer stehen nicht nur für IBM-Anlagen /47,48/, sondern

auch für Burroughs /49/ und Unidata /50/ zur Verfügung, für weitere Anlagen (CDC, Univac) sind Compiler in der Entwicklung. Ein PL/1-Standardisierungsvorschlag wurde von ECMA/ANSI erarbeitet /51/. Nach Versuchen, die im IRE auf der DVA IBM 370/165 durchgeführt wurden /52/, ist die Effektivität von mit dem IBM PL/1 Optimizing Compiler /53/ übersetzten Programmen bezüglich der Rechenzeit vergleichbar mit FORTRAN-Programmen, die mit dem IBM-H-Extended-Compiler /54,55/ übersetzt wurden. Der Speicherplatzbedarf liegt für die PL/1-Programme dagegen höher. Die besseren Programm-Ablaufsteuerungen und die Blockstruktur erleichtern die Anwendung des strukturierten Programmierens in PL/1 gegenüber FORTRAN.

Bestehende problemorientierte Sprachen auf PL/1-Basis sind PL/1-FORMAC /20/, EPL/1 /23/, GPL/1 /19/, GRAPPL/1 /18/, CPS-POGO /56/ und APL (Associative Programming Language) /57/. Eine einfache Definitionsmöglichkeit für PL/1-Erweiterungen wird durch den PL/1-Makroprozessor und durch PLITRAN /58/ ermöglicht, das in seinen Fähigkeiten allerdings noch unter denen des Makroprozessors liegt.

3.2 Interpretation und Kompilation problemorientierter Sprachen

Die zweite wichtige Entscheidung beim Entwurf der REGENT-Sprachverwaltung mußte die Frage beantworten:

> Werden die problemorientierten Sprachen interpretiert oder kompiliert?

Um Programme ausführen zu können, müssen sie entweder interpretiert oder kompiliert werden. Bei der Interpretation werden einzelne Anweisungen nacheinander von einem Programm, dem Interpretierer, eingelesen und analysiert. Je nach der Aufgabe, die durch die betreffende Anweisung verlangt wird, wird danach zu einem Stück Programmcode verzweigt, der diese Aufgabe lösen kann (Abb.3). Jedes FORTRAN-Programm, das über Eingabekarten veranlaßt werden kann, nacheinander verschiedene Teilaufgaben zu lösen, ist also ein Interpretierer für die aus den möglichen Eingabekarten bestehende Sprache.

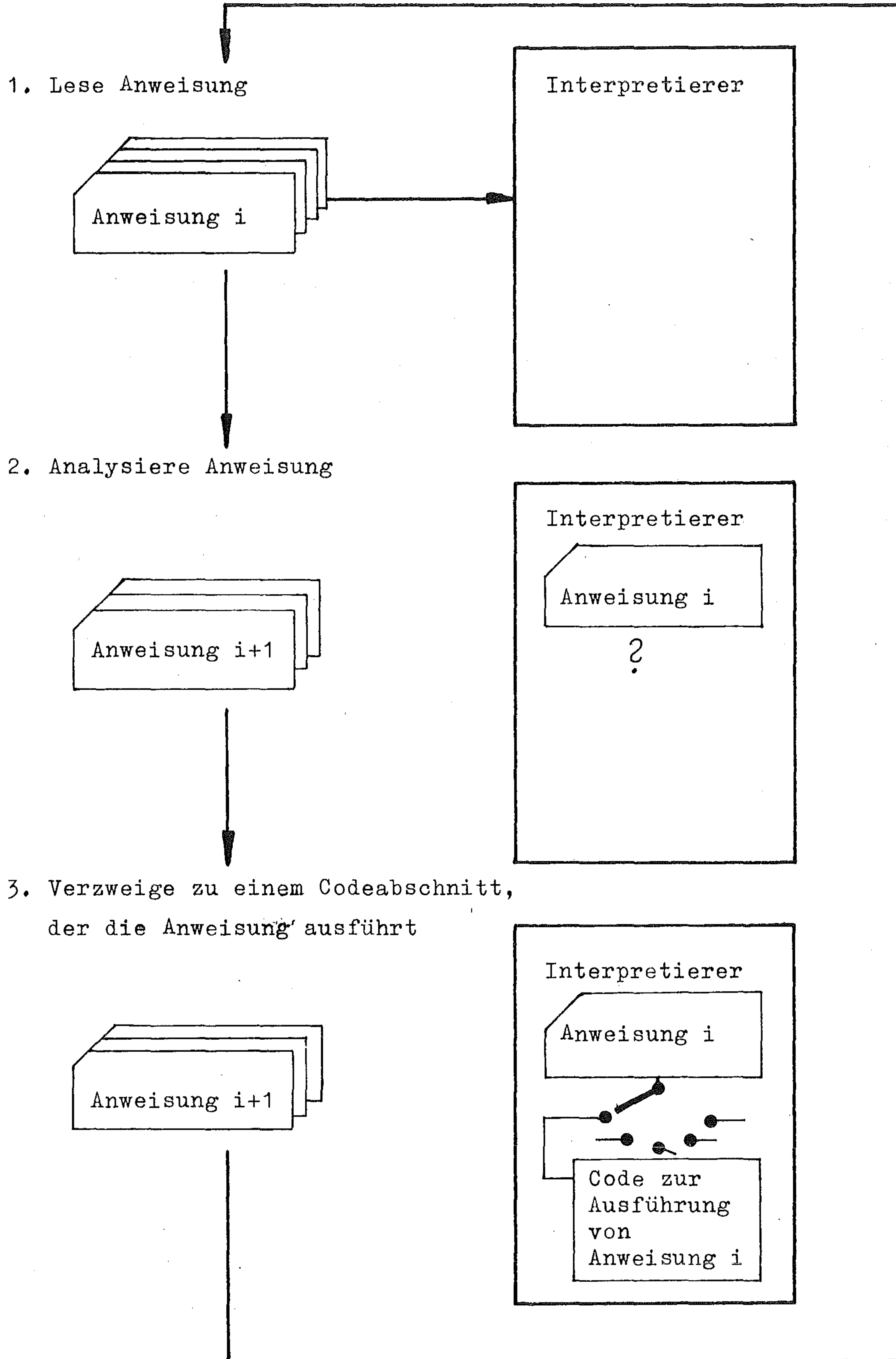


Abb.3: Interpretation von Anweisungen

Bei der Kompilation wird dagegen von einem Programm (dem Compiler) das gesamte Programm eingelesen, analysiert und ein Programm in Maschinencode erzeugt, das die gestellten Aufgaben bewältigen kann. Der generierte Maschinencode wird dann ausgeführt (Abb. 4).

Interpretation hat den Vorteil, daß bei interaktivem Betrieb der Lauf der Rechnung durch den Anwender verfolgt und kontrolliert werden kann. Nachteilig ist die verminderte Effektivität bei interpretierten gegenüber kompilierten Programmen. Programme, die mit dem (interpretierenden) PL/1-Checkout-Compiler /59/ ausgeführt werden, sind z.B. bis zu 10 mal langsamer als solche, die mit dem PL/1-Optimizing Compiler kompiliert und anschließend ausgeführt werden. Dabei ist im zweiten Fall die Kompilierzeit mit eingerechnet. Wenn Unterprogramme,bedingte Anweisungen, Schleifen und Sprünge in der POL möglich sind, ist eine direkte Interpretation Anweisung nach Anweisung nicht möglich. Erst nachdem in einem Schritt Information über Datendeklarationen und gegebenenfalls über die Blockstruktur des Programms gesammelt und ausführbare Anweisungen in eine interne Form gebracht wurden, kann in einem zweiten Schritt der interne Programmtext interpretiert werden. Auf diese Weise geht der IBM-PL/1-Checkout-Compiler vor /60/.

Neben der höheren Effektivität hat die Kompilation von POLs den weiteren Vorteil, daß während der Übersetzungsphase der Kernspeicher nicht von Problemlösungsprogrammen beansprucht wird, wie aus einem Vergleich von Abb. 3 und Abb. 4 ersichtlich ist. Deshalb kann der Übersetzer komfortabler und vielseitiger gemacht werden. Bei der POL-Interpretation müssen die der POL zugehörigen Problemlöse-Programme den Kernspeicher mit dem Interpretierer teilen oder der Interpretierer muß zeitweise auf Externspeicher ausgelagert werden. Es ist wünschenswert, ein POL-Programm wahlweise interpretieren oder kompilieren zu können, um ein Programm sowohl interaktiv unter Anwenderkontrolle als auch effektiv im Stapelbetrieb ausführen zu können.

Für die REGENT-Sprachverwaltung PLS fiel die Entscheidung zugunsten der Kompilation von POLs. Außer den angeführten Vorteilen einer Kompilation und den Schwierigkeiten, die sich ergeben, wenn Sprachen

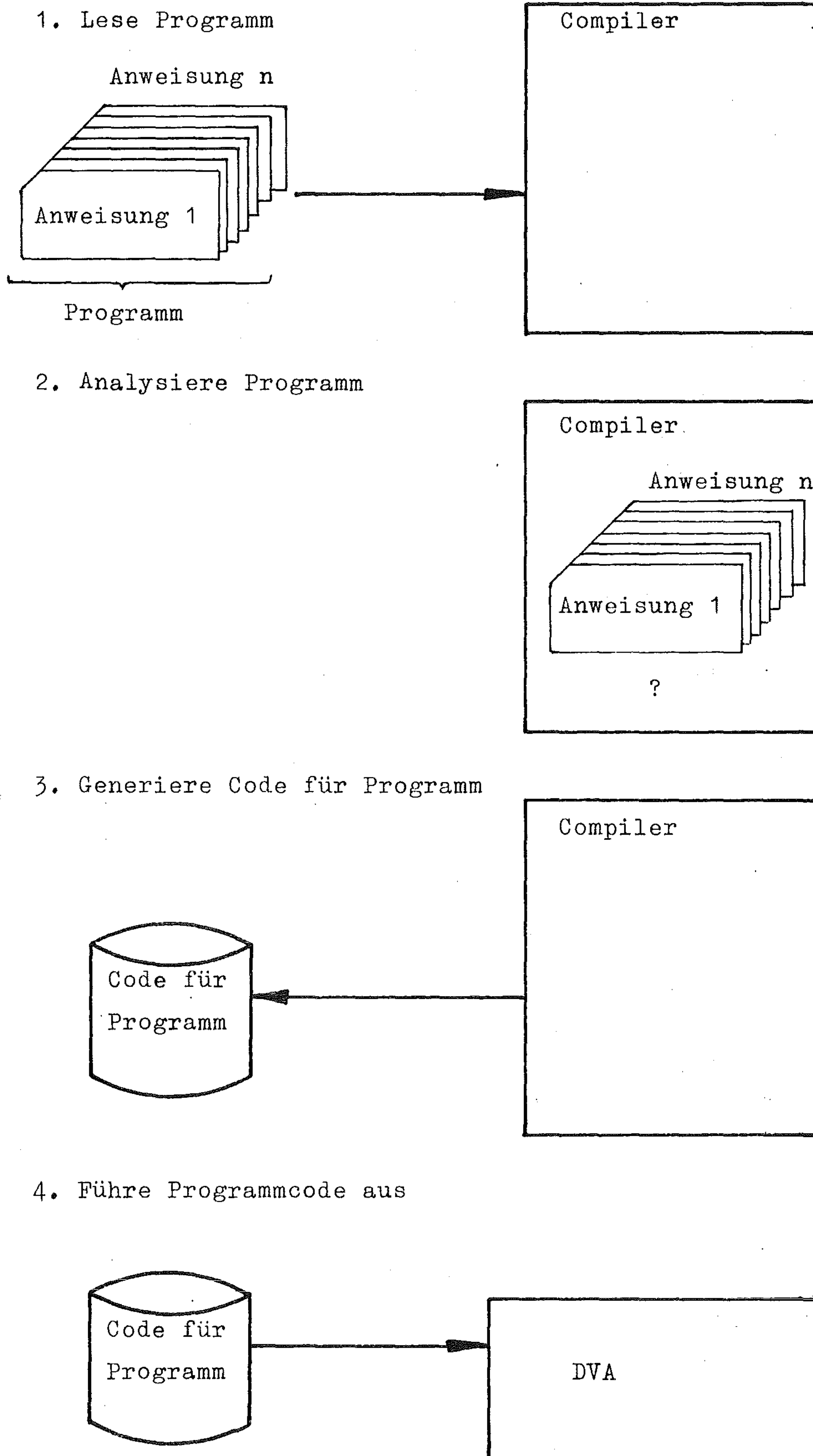


Abb.4: Kompilation eines Programmes

mit Schleifen, Sprüngen und Unterprogrammen interpretiert werden sollen, wurde diese Entscheidung auch wesentlich durch den erforderlichen Aufwand für die Erstellung eines Übersetzers mitbestimmt. Da die POLs in REGENT PL/1-Erweiterungen sind, müssen auch alle PL/1-Anweisungen übersetzt werden können. Eine POL-Kompilation kann aber aufgeteilt werden in eine POL-Vorkompilation, in der nur die POL-spezifischen Anweisungen betrachtet und übersetzt werden und in eine reine PL/1-Kompilation, die man dann dem schon vorhandenen PL/1-Compiler überlassen kann. Ein Interpretierer dagegen läßt eine derartige Aufgliederung in einen neu zu erstellenden Teil und einen vorhandenen Teilinterpretierer nicht zu.

3.3 Grundstruktur des PLS-Übersetzers

Die REGENT-Sprachverwaltung PLS besteht aus zwei Hauptbestandteilen - dem PLS-Übersetzer zur Anwendung problemorientierter Sprachen und dem PLS-Subsystem zur Definition solcher Sprachen. Eine Sprache, die für ein neues REGENT-Subsystem definiert wurde, kann anschließend zur Anwendung dieses Subsystems verwendet werden. Für die Realisierung des Übersetzers mußte die Frage entschieden werden:

> Wie liegen die Übersetzungsvorschriften vor, wie erfolgt die Umschaltung zwischen verschiedenen Sprachen?

Der Übersetzer muß alle als PL/1-Erweiterungen vorliegenden POLs in ausführbaren Maschinencode kompilieren können. Um nicht einen PL/1-Compiler erstellen zu müssen, wurde der Übersetzer in zwei Teile getrennt - in einen Vorübersetzer (im folgenden PLS-Übersetzer genannt), der in der Lage ist, POL-spezifische Anweisungen in legale PL/1-Anweisungen zu übersetzen und in einen PL/1-Übersetzer, der das vorübersetzte Programm in Maschinencode übersetzt (Abb. 5).

Der PLS-Übersetzer mußte neu entwickelt werden, während der PL/1-Übersetzer der für die verwendete DVA vorhandene PL/1-Compiler ist. Nach der Übersetzung muß das Programm noch gebunden werden, dies ist jedoch für diese Betrachtung nicht wesentlich.

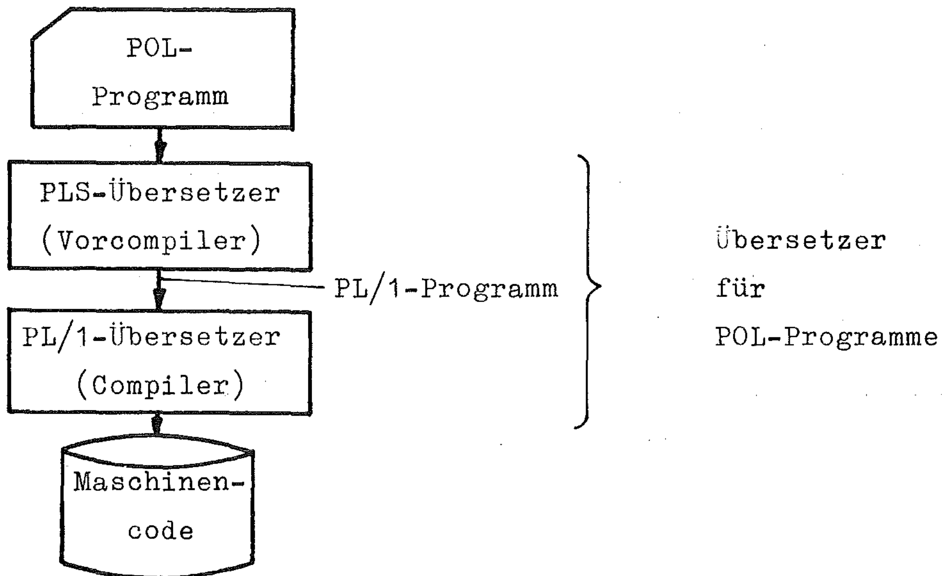


Abb. 5: Vorübersetzer und Compiler

Der PLS-Übersetzer muß viele verschiedene POLs übersetzen, er muß also Übersetzungsvorschriften für jede Sprache und eine Umschaltemöglichkeit besitzen. Dafür gibt es drei prinzipielle Möglichkeiten:

(1) Ein Übersetzer mit Vorschriften für alle POLs

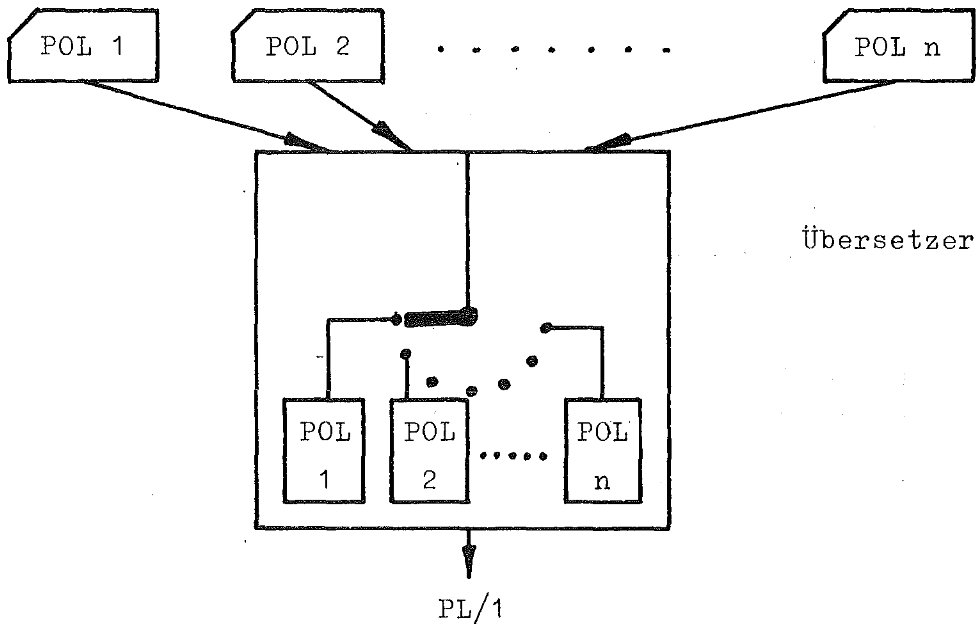


Abb. 6: Ein Übersetzer mit Übersetzungsvorschriften für alle Sprachen

(2) Je ein Übersetzer für jede POL

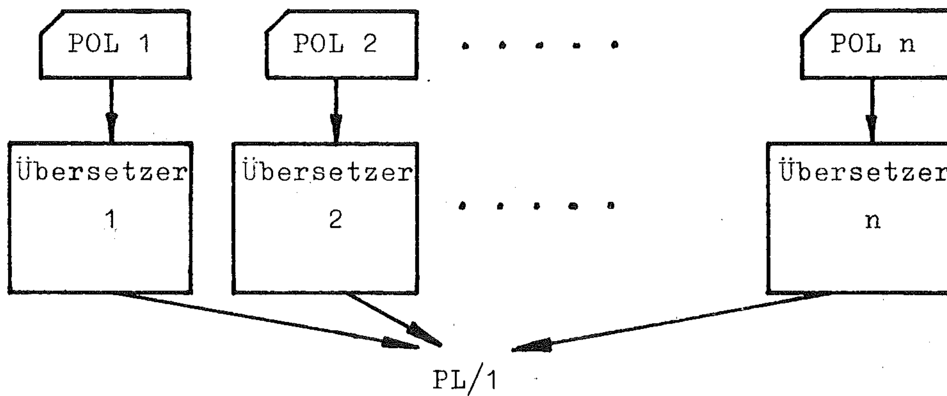


Abb. 7: Zuordnung je eines Übersetzers zu jeder Sprache

(3) Eine Kombination aus (1) und (2), ein Übersetzerteil gemeinsam für alle POLs und Einzelteile, die zu jeder POL gehören.

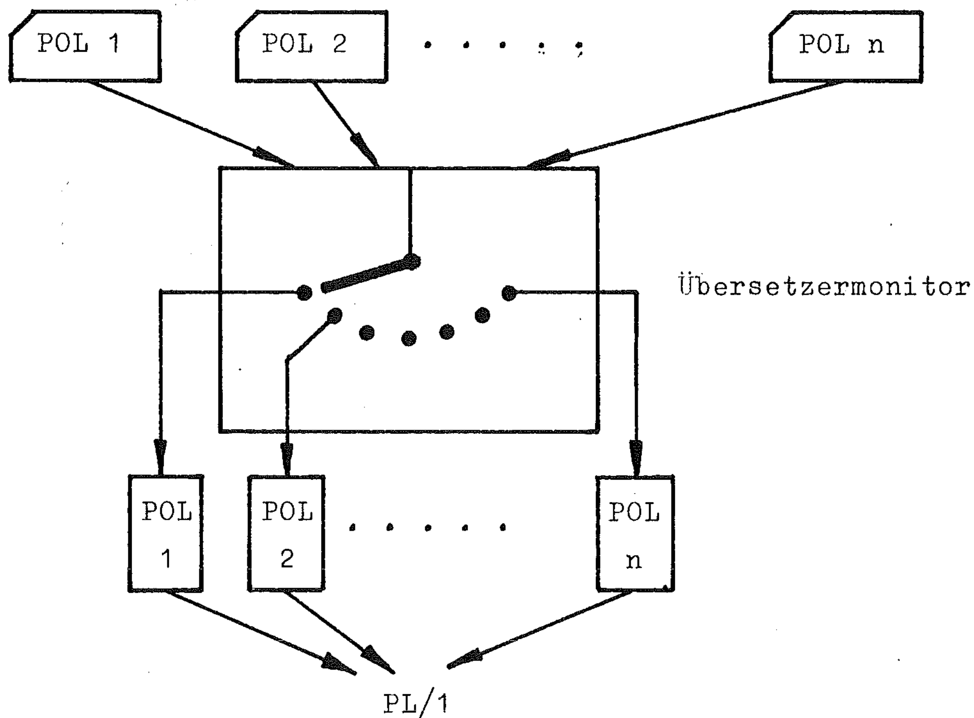


Abb. 8: Übersetzer bestehend aus Monitor und POL-spezifischen Einzelteilen

Diese letzte Möglichkeit wurde als Grundlage für den PLS-Übersetzer benutzt. Sie stellt die effektivste Realisierungsmethode dar und entspricht der Sprachstuktur: Eine allen POLs gemeinsame Basissprache

und POL-spezifische Bestandteile. Alle Aufgaben, die für alle POLs gleichermaßen bei der Übersetzung bewältigt werden müssen und die Umschaltung von einer POL zur anderen werden im Hauptteil des Übersetzers vorgenommen, dem Übersetzermonitor. (Ein Monitorprogramm ist ein Programm, das eine Reihe anderer, ihm untergeordneter Programme aufruft und deren Ablauf überwacht.) Die POL-spezifischen Bestandteile müssen die Übersetzungsvorschriften für die jeweilige problemorientierte Sprache enthalten. Diese Vorschriften können in verschiedener Weise vorliegen:

- Als Tabellen (in ICES verwendet), sie sind vor allem für die Übersetzer von Kommandosprachen geeignet.

- Als verkettete Listen (linked lists), da solche Listen flexibler abgearbeitet werden können als Tabellen, gestatten sie eine flexiblere Syntax der POL-Anweisungen und -Teilanweisungen.

- In der Quellform der POL-Definition, die bei Bedarf jeweils interpretiert wird (verwendet im OS/360-Macroassembler /61/ und im PL/1-Makroprozessor /47/).

- Als kompilierter Code, der bei seiner Ausführung die POL-Übersetzung vornimmt.

Die letzte Möglichkeit ist dann überlegen, wenn eine einmal definierte POL sehr oft ausgeführt wird. Da dies im REGENT-System der Fall ist, wurde die Methode gewählt, POL-Definitionen zu kompilieren und als fertige ausführbare Programmodule für die Verwendung durch den Übersetzermonitor bereitzustellen. Jeder derartige Programmodul kann eine ganz bestimmte POL-Anweisung übersetzen, er wird daher Anweisungstreiber genannt. Der Ausdruck stammt ebenso wie die Methode, für je eine Anweisung ein Programm bereitzustellen, aus dem Bereich des Compilerbaus. In PLS muß jedoch für jede POL ein besonderer Satz von Anweisungstreibern verfügbar sein, die Anweisungstreiber müssen überdies durch die POL-Definition automatisch erzeugt werden.

Neben den Anweisungstreibern bestehen die einer POL und damit einem REGENT-Subsystem zugeordneten Bestandteile des Übersetzers aus Tabellen, die den Subsystemzustand beschreiben und aus Datenstrukturen, die das Subsystem benutzt.

Die wichtigsten Entwurfsentscheidungen für den Übersetzer problemorientierter Sprachen können folgendermaßen zusammengefaßt werden:

- Die POL-Übersetzung wird in zwei Schritten vorgenommen, durch Vorkompilation der POL in PL/1 (PLS-Übersetzer) und anschließende Kompilation (PL/1-Compiler).
- Der PLS-Übersetzer besteht aus einem allen POLs gemeinsamen Hauptteil, dem Übersetzermonitor, und Bestandteilen für die einzelnen POLs.
- Die Übersetzung von POL-Anweisungen wird durch kompilierte und ablauffähige Programmodule, Anweisungstreiber, vorgenommen.

3.4 Sprachdefinition

Während der PLS-Übersetzer dem Anwender eines REGENT-Subsystems die Benutzung einer problemorientierten Sprache ermöglicht, muß der Entwickler eines Subsystems die Möglichkeiten haben, für sein neues Subsystem eine problemorientierte Sprache neu zu definieren. Die beiden Hauptaspekte der Sprachdefinition sind:

- Welche Syntax und Semantik hat die Definitionssprache, mit der neue Sprachen definiert werden können?
- Wie werden die Sprachdefinitionen analysiert, abgespeichert und dem Übersetzer zur Verfügung gestellt?

Die Definitionssprache stellt auch hier die Schnittstelle zum Anwender, also zum Subsystementwickler, dar. Sie ist daher wesentlich für die leichte, auch einem Ingenieur ohne Informatikausbildung mögliche Anwendung der Sprachdefinition (Abb. 9). Durch die Sprachdefinition wird Syntax und Semantik der problemorientierten Sprache festgelegt. Für die Syntaxdefinition gibt es zwei verschiedene grundsätzliche Methoden, die deskriptive Definition und die prozedurale Definition.

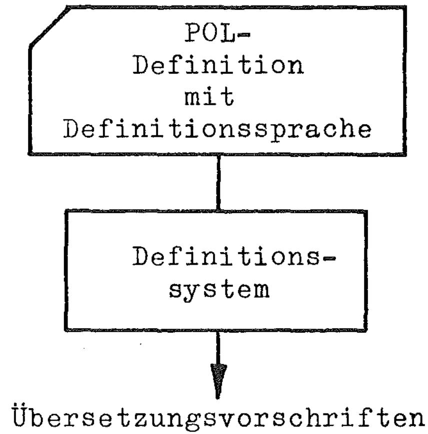


Abb. 9: POL-Definitionssystem

Deskriptive Syntaxdefinition

In einer geeigneten formalen Notation wird hierbei die Syntax der POL beschrieben, die Erstellung eines Syntaxanalysators zur Abarbeitung der so definierten POL wird automatisch anhand der Syntaxbeschreibung vorgenommen. Als Beispiel wird die Syntax einer POL-Anweisung mit der in Anhang A beschriebenen Syntaxnotation vorgenommen und danach eine mögliche formale Syntaxbeschreibung, die sich an die BNF-Notation (Backus-Naur-Form, /62/) anlehnt, gegeben:

Syntax der Anweisung:

$$\text{PLOT} \left[\text{DEVICE} \begin{Bmatrix} \text{PRINTER} \\ \text{PLOTTER} \end{Bmatrix}\right] \left[\text{FORMAT} \quad x * y \quad \text{CM}\right] \;;$$

Mögliche Syntaxdefinition:

```
anweisung ::= PLOT [device] [format];
device    ::= DEVICE (PRINTER | PLOTTER)
format    ::= FORMAT xwert * ywert CM
xwert     ::= realexpression
ywert     ::= realexpression
realexpression ::= ......
```

Es zeigt sich, daß eine solche formale Syntaxbeschreibung nicht die nötige Einfachheit und Handhabbarkeit besitzt, daß sie für die Sprachdefinition für Subsystemsprachen vorteilhaft verwendet werden könnte. Daher benutzen auch weder andere integrierte Systeme noch die allgemein

bekannten Makroprozessoren diese Methode. Eine Schwierigkeit ist außerdem die Unterbringung der Semantik der POL innerhalb der formalen Syntaxbeschreibung.

Prozedurale Syntaxdefinition

Hier wird nicht beschrieben, wie die Syntax ist, sondern wie sie analysiert wird, also wie die einzelnen Anweisungen einer POL abgearbeitet werden. Die Definition besteht also aus der Angabe von Anweisungen zur Übersetzung der POL. Für obiges Beispiel ist das Ablaufdiagramm für die Abarbeitung der Anweisung in Abb.10 dargestellt.

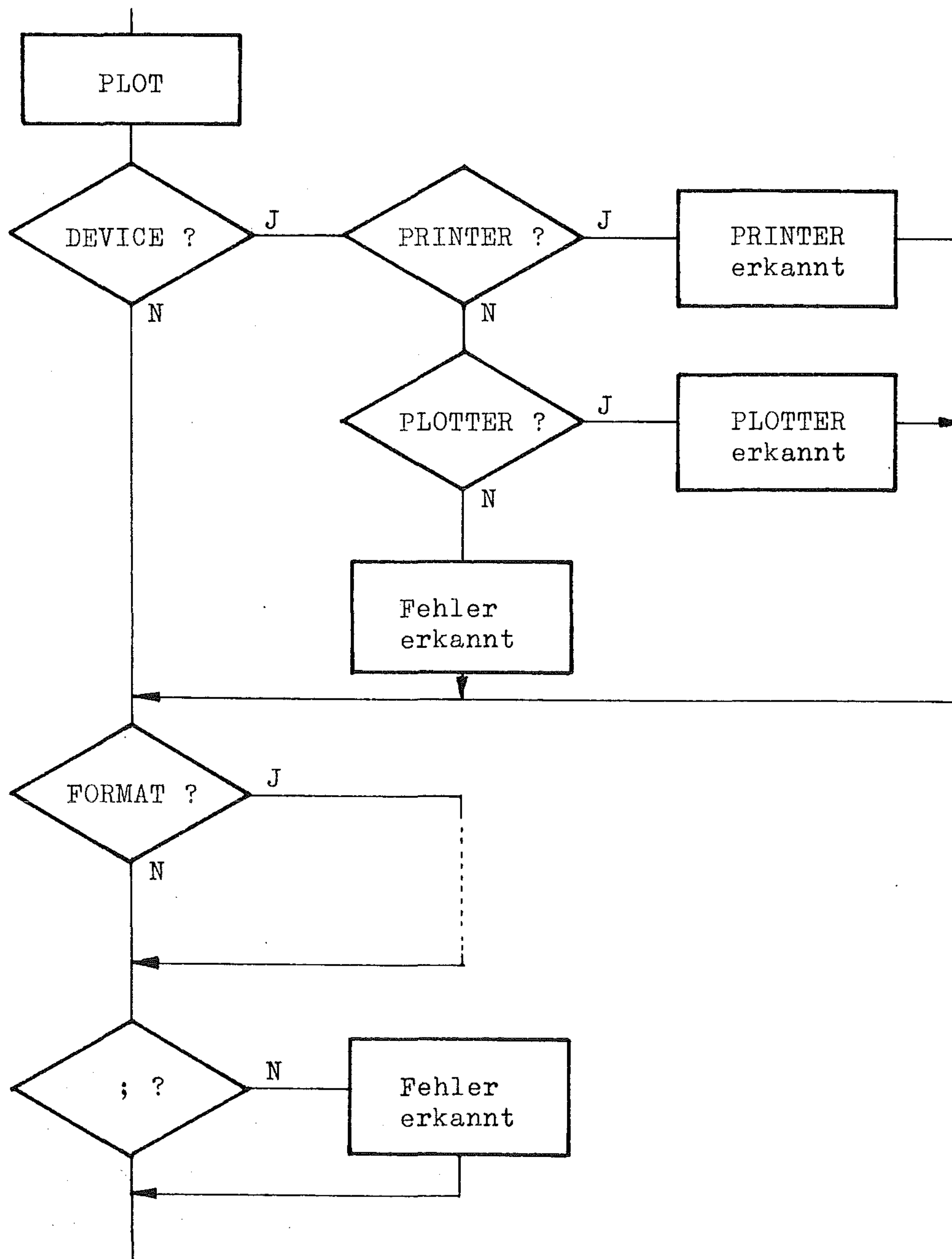


Abb. 10: Abarbeitung einer POL-Anweisung

Durch diese Methode kann auch bei erkannten Fehlern ("Fehler erkannt") leicht eine abgestufte Fehlerreaktion vorgesehen werden. Darüber hinaus ist die Unterbringung der Semantik in einem solchen Ablauf sehr einfach. Da im REGENT-System POLs in PL/1 übersetzt werden, besteht die Realisierung der semantischen Fähigkeiten einer POL-Anweisung in der Erzeugung von PL/1-Text (PL/1-Anweisungen und Teile solcher Anweisungen). Diese Texterzeugung kann in die Abarbeitungsroutine eingefügt werden (im Beispiel an den Stellen: "PRINTER erkannt", "PLOTTER erkannt"). Aus diesen Gründen wurde zur Syntax- und Semantikdefinition von POL-Anweisungen für REGENT-Subsysteme die Methode der Beschreibung der Abarbeitungsvorschrift gewählt. Da der Subsystem-Ersteller im REGENT-System ein Ingenieur sein wird, der eher gewöhnt ist, in Abläufen als in abstrakten Strukturen zu denken, ist diese Methode auch dem voraussichtlichen Anwender der Sprachdefinition angemessen.

Eine problemorientierte Sprache muß nicht vollständig in einem Schritt definiert werden, vielmehr können einzelne POL-Anweisungen nacheinander definiert werden. Zur Angabe der Abarbeitungsvorschrift für eine Anweisung ist eine geeignete Sprache erforderlich. Da aus Abb. 10 ersichtlich ist, daß die Sprache zumindest die in PL/1 schon vorhandenen Fähigkeiten: Verzweigungen und Abfragen haben muß, lag es nahe, nicht eine eigene neue Sprache zu entwickeln, sondern auch für die Definitionssprache den Weg der Erweiterung der System-Grundsprache PL/1 zu gehen. Die Programme zur Definition einer POL-Anweisung können also volles PL/1 benutzen. Diese Eigenschaft war bisher nur bei einigen Makroprozessoren in stark eingeschränktem Maße vorhanden /25/. Außer PL/1 enthält die Definitionssprache Funktions-Unterprogramme und einige besondere Anweisungen zur Abarbeitung der POL-Anweisung und zur Erzeugung von PL/1-Text.

Aufbau des Definitionssystems

Das Definitionssystem gehört logisch zu den REGENT-Grundbestandteilen. Trotzdem wurde es als Subsystem realisiert. Dadurch können die Fähigkeiten des PLS-Übersetzers zum Umschalten von einer Sprache auf die andere und zum Übersetzen problemorientierter Sprachen auch für die Sprachdefinition genutzt werden. Die Definitionssprache zur

Definition neuer POLs ist also selbst eine problemorientierte Sprache, die Sprache des REGENT-Subsystems PLS.

Die Abarbeitunsvorschriften für POL-Anweisungen sind in Definitions-Unterprogrammen, die PL/1-Anweisungen und einige Erweiterungen enthalten, niedergelegt. Diese Unterprogramme müssen nun in eine Form gebracht werden, daß sie der PLS-Übersetzer zur POL-Übersetzung verwenden kann. Da der PLS-Übersetzer fertig kompilierte, ablauffähige Programmodule (Anweisungstreiber) erwartet, müssen also die Definitions-Unterprogramme vorübersetzt, kompiliert, gebunden und als Lademodule abrufbereit gespeichert werden.

Die wichtigsten Eigenschaften des REGENT-Definitionssystems für problemorientierte Sprachen können wie folgt zusammengefaßt werden:

- Das POL-Definitionssystem ist ein REGENT-Subsystem.
- Die Definitionssprache enthält volles PL/1 und nur wenige Erweiterungen.
- Die Definition erfolgt durch Angabe der Abarbeitungsvorschrift.
- Die POL-Definitionen werden kompiliert und als Lademodule langfristig gespeichert.

4. F lisierung r REGENT-Sprachverwaltung

In sem Kapite wird die Realisierung des PLS-Übersetzers und des Def tionssystems nach den im vorigen Kapitel entwickelten Grundkor ten beschrieben. Ein Abschnitt über interaktive Definition und Anv ung problemorientierter Sprachen schließt sich an.

4. bersetzung problemorientierter Sprachen

4. Ablauf der POL-Übersetzung

Di ür REGENT-Subsysteme definierten problemorientierten Sprachen we en durch einen Vorcompiler nach PL/1 übersetzt. Abb. 11 zeigt den Ab uf einer POL-Übersetzung ausführlich. Das vorübersetzte PL/1-Pr gramm wird dann durch den PL/1-Compiler der DVA kompiliert und anschließend ausgeführt. Der PLS-Übersetzer besteht aus einem speicherresidenten Monitor, der für die Übersetzung aller POLs benötigt wird und aus Bestandteilen, die den einzelnen Subsystemen zugeordnet sind und nur geladen und ausgeführt werden, wenn die jeweilige Subsystemsprache übersetzt wird. Im Übersetzerkern sind u.a. Fähigkeiten zur lexikalischen Analyse von POL- und PL/1-Anweisungen, zur Syntaxprüfung von PL/1-Anweisungen und zum Laden und Aufrufen der benötigten POL-spezifischen Übersetzerbestandteile realisiert. Die POL-spezifischen Bestandteile des Übersetzers sind im wesentlichen die Anweisungstreiber, die in der Lage sind, eine oder mehrere POL-Anweisungen zu kompilieren. Sie sind in Form von ausführbaren Lademoduln auf einer Bibliothek gespeichert ("Subsystem-POLs" in Abb.11). Außerdem gehören zu einem Subsystem Tabellen und Datenstrukturen, sie sind in einer zweiten Bibliothek abgelegt.

Der Monitor analysiert das POL-Programm soweit syntaktisch, um Spracherweiterungen, PL/1-Anweisungen und fehlerhafte Anweisungsnamen erkennen zu können. Die PL/1-Anweisungen werden unverändert in den übersetzten Programmtext übernommen, für die Übersetzung einer POL-Anweisung wird dynamisch der ihr zugehörige Anweisungstreiber von einer Bibliothek gerufen. Nach der Übersetzung einer Anweisung bleiben die

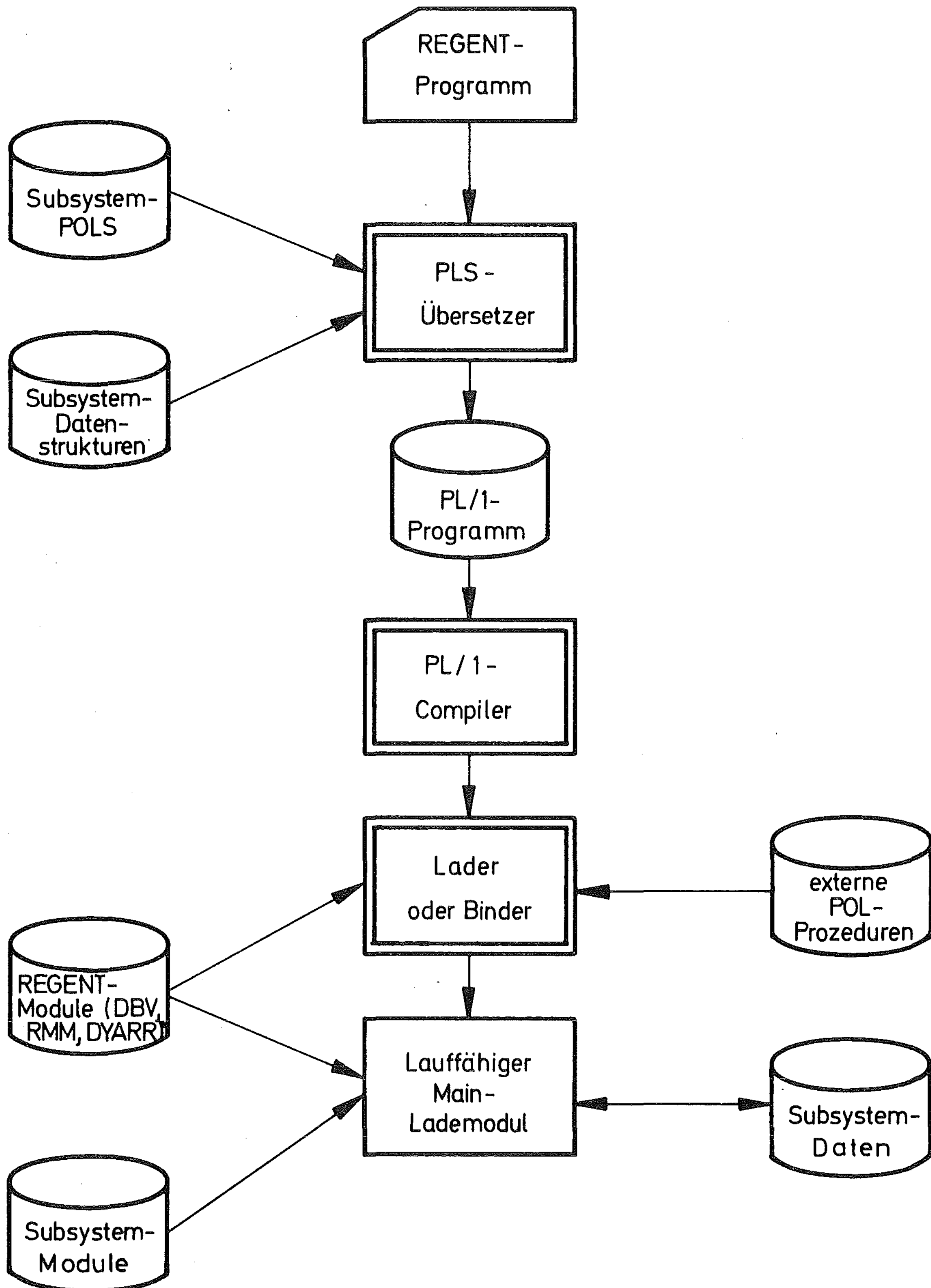


Abb. 11: Ablauf eines REGENT-Programmes

Treibermodule solange inaktiv im Arbeitsspeicher liegen, wie noch genügend freier Platz verfügbar ist. Ist für einen neu zu ladenden Modul zu wenig freier Speicherplatz vorhanden, werden inaktive Module überlagert. Die Treiberroutinen können über eine besondere Schnittstelle auf Fähigkeiten des Kerns zugreifen. Die Zuordnung eines Anweisungstreibers zu einer POL-Anweisung geschieht über eine Tabelle, die dem jeweiligen Subsystem zugeordnet ist und beim Subsystemstart geladen wird.

Um den richtigen Anweisungstreiber von der Bibliothek laden zu können, müssen die einzelnen Anweisungen eines POL-Programms erkannt werden. Folgende Arten von POL-Anweisungen sind möglich:

- Nicht modifizierte PL/1-Anweisungen
- Modifizierte PL/1-Anweisungen und neue Subsystem-Sprachanweisungen
- Systemanweisungen
- Fehlerhafte Anweisungen.

PL/1-Anweisungen und modifizierte PL/1-Anweisungen können außer bei Zuweisungen und Nullanweisungen (leeren Anweisungen, sie bestehen nur aus ";") an ihren Namen erkannt werden. Subsystem-Sprachanweisungen können ebenfalls mit einem Schlüsselwort beginnen. Es gibt jedoch auch die Möglichkeit, POL-Anweisungen zu verwenden, die mit einem bestimmten Datentyp beginnen (ganze Zahlen, reelle Zahlen, Zeichen- oder Bitketten, Operatoren). Ob eine derartige Datentyp-Anweisung vorliegt, wird geprüft, wenn die betrachtete Anweisung keine Zuweisung, keine Nullanweisung und keine Schlüsselwort-Anweisung ist. Wenn auch keine Datentyp-Anweisung erkannt wird, liegt eine fehlerhafte Anweisung vor (Abb. 12, diese Abbildung wurde mit dem REGENT-Subsystem FLODRA erzeugt, das Flußdiagramme zeichnen kann).

Systemanweisungen werden ebenfalls an ihrem Schlüsselwort erkannt, es sind Nicht-PL/1-Anweisungen, die nicht zu einem Subsystem gehören, sondern zur Steuerung des Ablaufs einer POL-Programmausführung dienen. Ihre wichtigste Anwendung ist der Aufruf und der Abschluß von Subsystemen. Außerdem gibt es Systemanweisungen zum Testen von Programmen

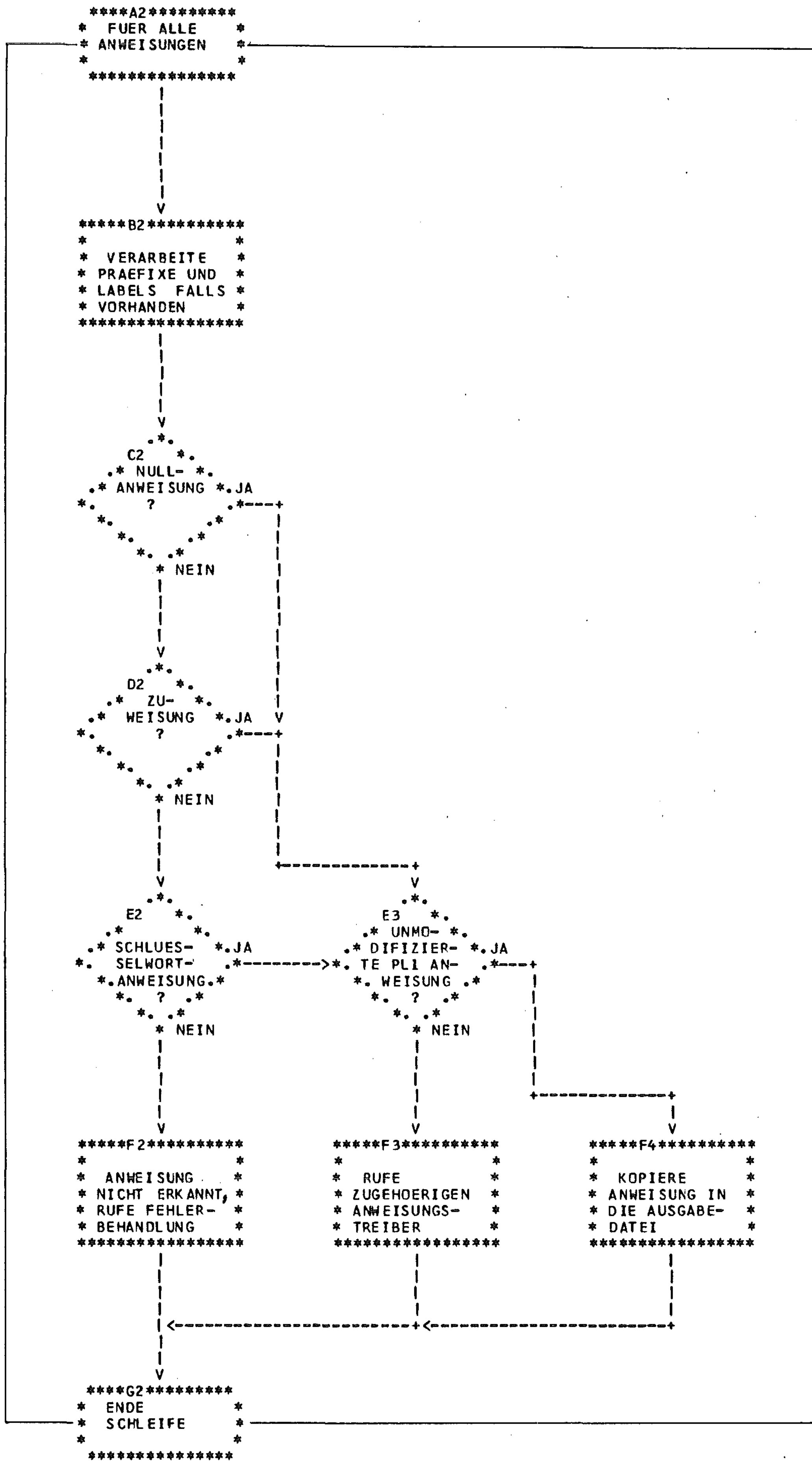


Abb. 12: Erkennen von POL-Anweisungen

und zur Ausgabe von Verwaltungsinformation aus dem REGENT-Systemkern. Auf der PROCEDURE-Anweisung eines POL-Programms ist eine zusätzliche Option mit dem Namen REGENT anzugeben, die dem Setzen von Systemkern-Parametern dient. Die REGENT-Option und die Systemanweisungen sind im Anhang B1 zusammengestellt.

Vor POL-Anweisungen und Subsystem-Anweisungen können ebenso wie vor PL/1-Anweisungen mehrere Präfixe stehen, Bedingungs-Präfixe und Marken (condition prefixes, label prefixes). Sie werden vom Monitor des Übersetzers erkannt und können in der PL/1-üblichen Weise verwendet werden. Sie werden jedoch auch den Treiberroutinen zur Verfügung gestellt, so daß diese für Sonderfälle neue Arten von Präfixen oder neue Anwendungen bereitstellen können.

Die Ausgabe des generierten PL/1-Textes erfolgt auf eine sequentielle temporäre Datei im Kartenformat, die anschließend dem PL/1-Compiler als Eingabe dient. Eine Liste der Eingabe, versehen mit Anweisungsnummern (erforderlich für Fehlermeldungen), kann auf die Standardausgabedatei SYSPRINT oder eine beliebige andere Datei ausgegeben werden. Eine gedruckte Ausgabe des vorübersetzten PL/1-Programmes kann entfallen, da der Compiler eine solche Liste ausdruckt.

Einzelheiten zur Implementierung des PLS-Übersetzers (Syntaxanalyse, Interface vom Monitorprogramm zu den Treiberroutinen) sind im Anhang C1 aufgeführt.

4.1.2 Aufruf und Abschluß von Subsystemen

Ein Subsystem wird eröffnet durch eine ENTER-Subsystem-Anweisung und durch eine END-Subsystem-Anweisung beendet. Die ENTER-Anweisung bewirkt die Suche des angegebenen Subsystemnamens in einer speicherresidenten Subsystem-Tabelle. Wenn der Name gefunden ist, wird die Anweisungstabelle für das betreffende Subsystem von einer Datenbibliothek gelesen. Während die Hauptaufgabe der Subsystemtabelle die Zuordnung von Subsystemnamen und Anweisungstabelle ist, stellt die Anweisungstabelle für ein Subsystem den Zusammenhang von Anweisungen und Treibermodulnamen dar.

Im erzeugten PL/1-Text wird an der Stelle des Subsystemaufrufes ein BEGIN-Block eröffnet. Danach folgen zu jedem Subsystem gehörende Datenstrukturdeklarationen (vgl. Abschnitt 4.1.3). Für jedes Subsystem kann eine Übersetzungszeit-Prozedur bereitstehen, die beim Aufruf eines Subsystems zur Übersetzungszeit des POL-Programms PL/1-Anweisungen für Initialisierungen erzeugen kann. Diese Initialisierungsroutine wird dynamisch aus der Anweisungstreiberbibliothek geladen. Die ENTER-Anweisung bewirkt also im generierten PL/1-Text das Eröffnen eines Blockes, das Einfügen von Deklarationen und von Anweisungen zur Subsysteminitialisierung.

Die END-Subsystem-Anweisung erzeugt im generierten PL/1-Text den Abschluß des Subsystemblockes durch eine PL/1-END-Anweisung. Der Platz der Anweisungstabelle wird freigegeben.

Subsystemaufrufe können geschachtelt werden. In einem inneren Subsystemblock sind nur die Spracherweiterungen dieses Subsystems gültig, jedoch kann auch auf die im umfassenden Block deklarierten Datenstrukturen zugegriffen werden (siehe Abb. 13). Auf diese Weise ist eine Kommunikation von Daten zwischen verschiedenen Subsystemen möglich. Die Umschaltung von einer Sprache zur anderen durch eine hierarchische Blockstruktur von Sprachebenen, entsprechend den BEGIN- und PROCEDURE-Blöcken in PL/1 ist ein Weg, der zum erstenmal im PLS-System gegangen wurde. Dies ist eine einer Sprache mit Blockstruktur adäquate Methode der Umschaltung der Gültigkeit von Sprachen. Außerdem stellt sie sicher, daß die Subsystem-Datenstrukturen auch nur während der Zeit der Anwendung des Subsystems vorhanden und zugreifbar sind. Wenn Subsysteme hintereinander aufgerufen werden, kann eine Kommunikation zwischen ihnen über in einem äußeren Block deklarierte Datenaggregate (Variable, Felder, Strukturen) oder auch über Dateien erfolgen. Letzteres ist auch über mehrere Rechenläufe hinweg möglich.
Beispiel dazu:

```
DECLARE X  ..... , F FILE;   Deklariere Datenaggregat X und Datei F
ENTER SUB1;                  Eröffne Subsystem SUB1
    :                        Berechne X, beschreibe F
END SUB1;                    Ende Subsystem SUB1
ENTER SUB2;                  Eröffne Subsystem SUB2
    :                        Verwende X, lese F
END SUB2;                    Ende Subsystem SUB2
```

Anweisungen	Sprachanweisungen gültig von Subsystem	Datenstrukturen zugreifbar von Subsystem
. . .	NUR SYSTEM-ANWEISUNGEN	
ENTER A;		
. . . .	A	A
ENTER B;		
. . . .	B	A,B
ENTER C;		
. . . .	C	A,B,C
END C;		
. . .	B	A,B
END B;		
. . .	A	A
ENTER D;		
. . . .	D	A,D
END D;		
. . .	A	A
END A;		
. . .	NUR SYSTEM-ANWEISUNGEN	

Abb.13: Schachtelung von Subsystemaufrufen

4.1.3 Behandlung von Subsystemdatenstrukturen

Während der Übersetzung problemorientierter Programme werden durch den PLS-Übersetzer zwei Bibliotheken benötigt. Die eine enthält die zur Übersetzung von POL-Anweisungen benötigten Anweisungstreiber-Module, die andere ist eine Datenbibliothek im Kartenformat und enthält neben den Subsystem- und Anweisungstabellen die Deklaration globaler Subsystemdatenstrukturen. Beliebig viele Deklarationen können für jedes Subsystem definiert sein, sie werden beim Subsystemaufruf in den erzeugten PL/1-Text kopiert und können dadurch während der Rechnung referiert werden. An die Verarbeitungsmodule des Subsystems können sie als Parameter übergeben werden. Außer PL/1-Datentypen können zusätzlich auch die REGENT-Datentypen DYNAMIC ENTRY /63/, DYNAMIC ARRAY, BASEDESCRIPTOR /64/ und BANK /65/ deklariert werden.

Eine spezielle Subsystemdatenstruktur, der Subsystem-Common, wird automatisch über eine Pointervariable an alle Subsystemmodule übergeben. Bei der Modulgeneration wird die Deklaration dieses Subsystem-Commons in das Programm eingefügt, so daß der Subsystem-Common in allen Teilen des Subsystems referiert werden kann. Er stellt somit eine bequeme Kommunikationsmöglichkeit für das Subsystem dar. Der Common enthält auch immer den Namen des Subsystems, so daß dieser für Fehlermeldungen des Systemkerns verwendet werden kann. Die Commondeklaration wird wie andere Datenstruktur-Deklarationen am Beginn des Subsystemblocks in den erzeugten PL/1-Text kopiert.

4.1.4 Fehlerbehandlung zur Übersetzungszeit

Werden während der Vorübersetzung Fehler festgestellt, so erfolgt eine Fehlermeldung. Die Fehlerbehandlung erfolgt durch standardisierte Fehlermodule, die bei Bedarf dynamisch aufgerufen werden, sie befinden sich in der Treibermodulbibliothek. Das Laden der Fehlermodule geschieht über eine speicherresidente Interface-Routine im Monitor des Übersetzers, der die Fehlernummer mitgeteilt wird. Die Übergabe von zur Fehlermeldung oder -behebung notwendigen

Daten wird über einen Vektor von Zeigern (PL/1-Pointers) vorgenommen, die auf die betreffenden einzelnen Variablen oder Datenaggregate zeigen.

Eine Fehlerroutine enthält einen Standardteil, der für alle erfaßten Fehler gleich ist und eine Meldung ausgibt. Sie besteht aus Fehlernummer, erläuterndem Text und der Anweisungsnummer der fehlerhaften Anweisung. Die Meldung erfolgt auf die Standardausgabedatei (SYSPRINT), außerdem erscheint sie als Kommentar im generierten PL/1-Text. Abb. 14 zeigt eine Liste des Vorübersetzers mit Fehlermeldungen. In diesem Beispiel wird auch die Liste der Anweisungen auf SYSPRINT ausgegeben, deshalb stehen die Fehlermeldungen jeweils vor der zugehörigen Anweisung.

Der zweite Teil einer Fehlerroutine ist für jeden vorkommenden Fehler verschieden. Hier kann zusätzliche Information über die Fehlerursache ausgedruckt werden und, falls möglich, kann der Versuch einer Fehlerkorrektur erfolgen.

Eine Anzahl von Fehlern wird vom Übersetzer-Monitor festgestellt, die Behandlung von Fehlern innerhalb von Subsystem-Sprachanweisungen muß vom Subsystemersteller bei der Sprachdefinition vorgesehen werden. Dazu werden die Hilfsmittel bereitgestellt, die auch der Übersetzer selbst verwendet.

```
    1       T: PROC OPTIONS(MAIN) REGENT(NODA);                          00000520

***** PRECOMPILE TIME ERROR NO.   48 IN STATEMENT NO.      2
***** IN ENTER STATEMENT SUBSYSTEM NOSUBSYSTEM NOT FOUND

    2      ENTER NOSUBSYSTEM;                                            00000530

***** PRECOMPILE TIME ERROR NO.   31 IN STATEMENT NO.      3
***** STATEMENT NOT COMPLETELY PROCESSED, ";" ASSUMED
***** NEXT ITEM: "PRINT"

    3          PRINT POOLTABLE                                           00000540
    4          PRINT POOLDUMP;                                           00000550
    5      ENTER PLS;                                                    00000560

***** PRECOMPILE TIME ERROR NO.   32 IN STATEMENT NO.      6
***** STATEMENT NOT FOUND, WRONG KEYWORD OR DATATYPE
***** STATEMENT NAME: NOSTATEMENT

    6             NOSTATEMENT;                                           00000570

***** PRECOMPILE TIME ERROR NO.   31 IN STATEMENT NO.      7
***** STATEMENT NOT COMPLETELY PROCESSED, ";" ASSUMED
***** NEXT ITEM: "DATA"

    7             SUBSYSTEM 'TEST' KEY 'TESTTEST'                        00000580
    8             DATA COMMON;                                           00000590

***** PRECOMPILE TIME ERROR NO.   66 IN STATEMENT NO.      9
***** DECLARATIONLIST IN DECLARE-STATEMENT DOES NOT START
***** WITH NAME, NUMBER OR "(", NEXT ITEM:"

    9                  DCL 1,
                              " PTR,
   10                         2 BIN FIXED;                               00000610
   11             END DATA;                                              00000620
***** PRECOMPILE TIME ERROR NO.   31 IN STATEMENT NO.     12
***** STATEMENT NOT COMPLETELY PROCESSED, ";" ASSUMED
***** NEXT ITEM: "LIST"

   12             LIST SUBSYSTEMS                                        00000630
   13             LIST STATS'PLS';                                       00000640
   14             LIST STATS'GIPSY';                                     00000650
   15          END PLS;                                                  00000660
   16       END T;                                                       00000670
```

Abb.14: Liste des Übersetzers mit Fehlermeldungen

4.2 Definition problemorientierter Sprachen

4.2.1 Das REGENT-Subsystem PLS

Der PLS-Übersetzer dient zum Vorkompilieren aller bestehenden Subsystemsprachen im REGENT-System. Zum Definieren neuer Subsystem-Sprachen dient das REGENT-Subsystem PLS. Subsysteme können initialisiert und zerstört werden. Datenstrukturen und Sprachanweisungen können definiert und gelöscht werden. Dazu sind in der Sprache des Subsystems PLS POL-Anweisungen vorhanden (siehe Anh.B3). Durch Anwendung des PLS-Subsystems werden auf der Datenstrukturbibliothek und der Treibermodulbibliothek des PLS-Übersetzers Datendeklarationen, Tabellen und Module eingebracht, geändert oder gelöscht. Anschließend kann der PLS-Übersetzer mit den geänderten Bibliotheken arbeiten (siehe Abb.15). Das Subsystem PLS wird wie jedes andere REGENT-Subsystem gerufen und abgeschlossen, POL-Definitionen sind also möglich zwischen "ENTER PLS;" und "END PLS;".

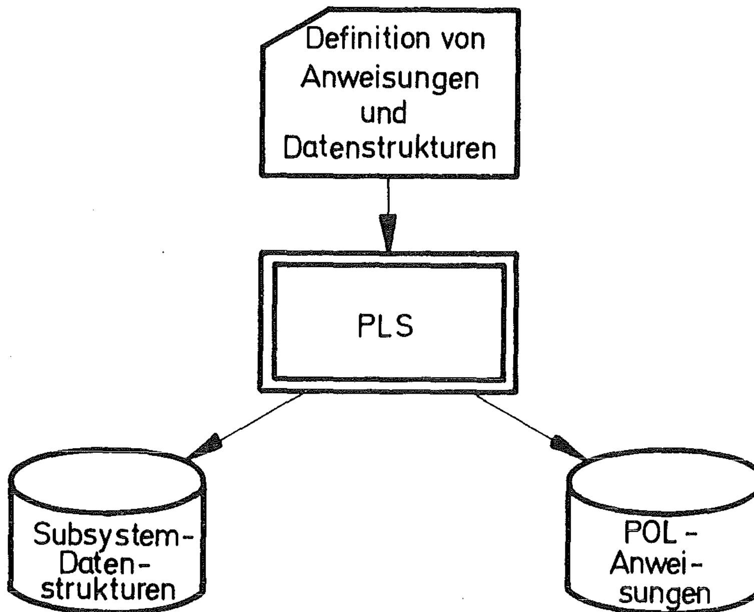


Abb. 15: Definition problemorientierter Spracherweiterungen

4.2.2 Steuerung der Sprachdefinition

Ein Subsystem wird durch die INITIATE-Anweisung initialisiert. Dabei wird der Name des Subsystems festgelegt, unter dem es später durch "ENTER ...." aufgerufen werden kann. Ein Schlüsselwort, das bei der Initialisierung angegeben wird, ist danach bei Änderungen oder Ergänzungen sowie beim Zerstören des Subsystems erforderlich. Das Subsystem ist so vor unautorisiertem Zugriff geschützt.

Die Subsystemsprachdefinition wird durch die PLS-Anweisungen SUBSYSTEM und STATEMENT gesteuert. Die SUBSYSTEM-Anweisung dient dazu, die folgenden POL-Definitionen einem bestimmten Subsystem zuzuordnen. Das bei der Initialisierung festgelegte Schlüsselwort muß dabei angegeben werden.

Zwischen den Anweisungen STATEMENT und END STATEMENT wird Syntax und Übersetzung einer Subsystemanweisung definiert. Diese Folge von Anweisungen stellt eine Makrodefinition dar und entspricht einer Preprocessorfunction des PL/1-Makroprozessors /47/ oder der Folge MACRO ... MEND des OS/360 Macroassemblers /61/. In dieser Makrodefinition wird angegeben, wie die zu definierende Subsystemanweisung von links nach rechts abgearbeitet wird und welcher PL/1-Text anstelle dieser Sprachanweisung generiert werden soll. Die Definition für eine Subsystemanweisung wird durch das PLS-Subsystem in einen ausführbaren Treibermodul umgesetzt, der vom Übersetzer zur Übersetzung dieser Anweisung verwendet werden kann.

Neben allen PL/1-Anweisungen können innerhalb einer Anweisungsdefinition besondere Anweisungen und Funktionsaufrufe verwendet werden, die dazu dienen, die zu definierende POL-Anweisung syntaktisch zu analysieren und entsprechende PL/1-Anweisungen zu erzeugen. Da die in einer Anweisungsdefinition aufgeführten Anweisungen zur Übersetzungszeit dieser Anweisung (während der Makro-Expansion) ausgeführt werden, heißen sie Makrozeitanweisungen. Die zusätzlich zu PL/1 verfügbaren Makrozeitanweisungen sind im Anhang B4 zusammengestellt. Die zur Abarbeitung der POL-Anweisung verwendeten Funktionen heißen PLS-Funktionen. Sie sind im Anhang B2 beschrieben.

Realisiert werden die zusätzlichen Makrozeitanweisungen und die PLS-Funktionen durch Zugriff auf die Monitorroutinen des Übersetzers über eine Interfacedatenstruktur. Der PL/1-Text, der anstelle der POL-Anweisung generiert wird, heißt Ersetzungstext ("replacement text" im PL/1-Makroprozessor, "prototype statements" im OS/360-Assembler).

Es gibt bei einer Anweisungsdefinition drei verschiedene Sprachebenen, je nach dem Zeitpunkt, zu dem die Sprachanweisungen ausgeführt werden. Dies gilt für alle Makroprozessoren, allein bei PLS können jedoch durchgängig auf allen Sprachebenen die gleichen Anweisungen verwendet werden, nämlich PL/1-Anweisungen. Die Verfügbarkeit des vollen PL/1-Sprachumfangs auf allen Sprachebenen ist eine Eigenschaft, die PLS von allen anderen bekannten Makroprozessoren unterscheidet.

Die erste Sprachebene besteht aus den Definitionszeitanweisungen (PLS-Anweisungen). Sie werden bei der Definition einer Anweisung ausgeführt. Die zweite Ebene ist die der Makrozeitanweisungen, die während der Übersetzung einer Anweisung ausgeführt werden. Die dritte Sprachebene schließlich besteht aus den Anweisungen im Ersetzungstext, es sind Ausführungszeitanweisungen, die bei der Ausführung einer überstzten POL-Anweisung wirksam werden. Sie stehen bei der Definition mittels PLS zwischen "EXEC;" und "END EXEC;".

Beispiel:

```
          ENTER PLS;
          SUBSYSTEM 'TEST' KEY 'TEST';
(1)       PUT LIST ('DEFINITIONSZEIT');
          STATEMENT 'ANWEISUNG';
(2)           PUT LIST ('MAKROZEIT');
              EXEC;
(3)               PUT LIST ('AUSFUEHRUNGSZEIT');
              END EXEC;
          END STATEMENT;
          END PLS;
```

In diesem Beispiel wird für das Subsystem TEST die Anweisung ANWEISUNG definiert. Bei dieser Definition wird die Druckanweisung (1) ausgeführt.

Später kann die neu definierte Anweisung angewandt werden:

```
ENTER TEST;
   ANWEISUNG;
END TEST;
```

Dabei wird bei der Übersetzung der Anweisung die Druckanweisung (2) ausgeführt. Bei der Ausführung des übersetzten und kompilierten Programmes erst wird die Druckanweisung (3) wirksam.

Einzelheiten zur Initialisierung und Zerstörung von Subsystemen werden in Anhang C2 behandelt. Auf die Einordnung der formalen Syntax der POLs in die PL/1-Syntax, auf Makrozeitanweisungen und PLS-Funktionen sowie auf Fragen der Implementierung wird im Anhang C3 ausführlich eingegangen.

4.2.3 Beispiel einer Anweisungsdefinition

Anhand eines vereinfachten Beispiels, das dem REGENT-Subsystem REMAC entnommen ist, wird die Definition einer Anweisung gezeigt. Die Anweisung dient der Angabe der Zeichnungsart (Druckerplot oder Zeichnung mit Zeichenmaschine) und der Zeichnungsgröße. Anhand dieses Beispiels wurde mit einem Ablaufdiagramm der Abarbeitungsvorschrift im Abschnitt 3.4 (S.28) die prozedurale Definition einer Anweisung erläutert. Die Anweisung hat folgende Syntax (beschrieben mit der in Anhang A angegebenen Syntaxnotation):

PLOT [DEVICE [{PRINTER | →PLOTTER}]] [,] [FORMAT [R] r [,] [Z] z] ;

r und z seien beliebige arithmetische Ausdrücke zur Angabe der Zeichnungsbreite und -höhe. Im erzeugten PL/1-Text sollen folgende Anweisungen erzeugt werden:

PLPRINT='1'B; falls DEVICE PRINTER angegeben wurde,
PLPLOT='1'B; falls DEVICE PLOTTER angegeben wurde,
RFORM=r; und ZFORM=z; falls FORMAT spezifiziert wurde.

e Definitijon für die POL-Anweisung lautet:

```
.)    STATE}'ENT 'PLOT';
)         D    DEV BIT(1) INITIAL('0'B);
)         I   3IDENTIFIER('DEV') THEN DO;
)              IF BIDENTIFIER('PRI') THEN DEV='1'B;
)              ELSE SKIPB ID('PLO');
)              IF DEV THEN EXEC PLPRINT='1'B;;
')             ELSE EXEC PLPLOT='1'B;;
3)             END;
9)        SKIP(',');
10)       IF IDENTIFIER('FORMAT') THEN DO;
11)            SKIP ID('R');
12)            EXEC RFORM=NEXT_EXPRESSION;;
13)            SKIP(',');
14)            SKIP ID('Z');
15)            EXEC; ZFORM=NEXT_EX; END EXEC;
16)            END;
17)   END .TAT;
```

Diese Definition folgt dem (vereinfachten) Ablaufdiagramm in Abb.10 (S.28). In Zeile (3) wird geprüft, ob DEVICE angegeben wurde, in den Zeilen (4) und (5) wird abhängig vom Vorhandensein von PRINTER oder PLOTTER die Makrozeitvariable DEV auf '1'B gesetzt oder auf '0'B belassen, in Zeile (6) und (7) wird der PL/1-Text "PLPRINT='1'B;" oder "PLPLOT='1'B;" erzeugt. Die Zeilen (10) bis (16) erzeugen die Anweisungen "RFORM=r;" und "ZFORM=z;" falls FORMAT in der POL-Anweisung angegeben wurde. In den Zeilen (5), (9), (11), (13) und (14) wird mit Hilfe der SKIP-Anweisung Fülltext übergangen.

Die Anweisungen (1) und (17) sind PLS-Anweisungen, sie werden zur Definitionszeit ausgeführt. Die Zeilen (2) bis (16) werden zur Makrozeit ausgeführt, es sind Makrozeitanweisungen. In den Zeilen (6), (7), (12) und (15) steht zwischen EXECUTE und END EXECUTE bzw. zwischen EXECUTE und dem zweiten Semikolon Ersetzungstext. In Zeile (3) und (4) wird die PLS-Funktion BIDENTIFIER benutzt um das Vorhandensein von

DEVICE und PRINTER festzustellen. Entsprechend wird in Zeile (9) mit Hilfe der PLS-Funktion IDENTIFIER festgestellt, ob die Benennung "FORMAT" vorhanden ist. In Zeile (12) und (15) wird mit Hilfe der PLS-Funktion NEXT_EXPRESSION der nächste arithmetische Ausdruck aus der POL-Anweisung gewonnen.

Die neu definierte Anweisung kann z.B. in folgender Weise verwendet werden:

```
PLOT DEVICE PLOTTER, FORMAT R 10 Z Z1+1;
```

Durch die angegebene Definition werden daraus folgende PL/1-Anweisungen erzeugt:

```
PLPLOT='1'B;
RFORM=10;
ZFORM=Z1+1;
```

4.3 Interaktive Definition und Anwendung von POLs

Die interaktive Anwendung einer DVA hat den Vorteil kürzerer Antwortzeiten und besserer Kontrolle des Programmablaufes. Der Programmanwender steuert dabei seine Programmaufträge von einer Datenendstation (Terminal) aus. Bei der Definition von Subsystemsprachen und bei der Anwendung dieser Sprachen sind verschiedene Stufen der Interaktivität möglich. Die folgenden Ausführungen sind bezogen auf das Betriebssystem OS/360 mit der Timesharing Option (TSO, /66/).

4.3.1 Zusammenstellen von Stapelaufträgen an der Datenendstation

Ein Stapelauftrag wird am Terminal in einer Datei zusammengestellt und anschließend als ganzes in die DVA eingebracht. Der Auftrag verhält sich von da an wie jeder andere (beispielsweise von einem Kartenleser eingelesene) Stapelauftrag. Diese "Submit"-Funktion des TSO ist also dem "Remote Job Entry" ähnlich, bei dem eine externe Eingabeeinheit räumlich von der DVA getrennt installiert ist.

Die Interaktivität beschränkt sich hier auf das Eingeben und Ändern der Eingabe und das Betrachten der Ausgabe eines Stapelauftrages. Die Antwortzeiten sind auch für kurzlaufende Programme verhältnismäßig lang. Jedoch ist ein Submit für alle Aufträge möglich, die auch im Stapelbetrieb laufen, also auch für solche mit extremen Anforderungen an die Resourcen der Rechenanlage (Speicher, Zentraleinheit, externe Einheiten). Ein Submit ist daher sowohl für die POL-Definition als auch für Subsystemanwendungen jederzeit möglich.

4.3.2 Interaktive Programmentwicklung

Hierbei wird ein Programm im Vordergrund des TSO wiederholt übersetzt und unter Anwenderkontrolle ausgeführt. Dabei kann es laufend getestet und verbessert werden (benutzerkontrollierter Dialog). Für anwendungsbezogene REGENT-Programme wird nacheinander der PLS-Übersetzer, einer der PL/1-Compiler und wenn erforderlich der Linkage Editor aufgerufen und anschließend das übersetzte Programm selbst. Zwischen den einzelnen Stufen kann das Programm geändert und ein oder mehrere Schritte wiederholt werden (siehe Abb. 16). Die Benutzerfreundlichkeit dieser Methode ist in erster Linie von den Antwortzeiten abhängig. Bei großer Auslastung der DVA können die Antwortzeiten länger als 30sec werden, trotzdem der PLS-Übersetzer wesentlich schneller ist als die anschließenden Compilerläufe. Die Ausführung von Programmen im Vodergrund ist begrenzt auf Programme mit kurzen Rechenzeiten.

4.3.3 Anforderung der Eingabedaten

Es ist möglich, Subsystemsprachen speziell für den interaktiven Betrieb zu konzipieren, so daß Daten, die erforderlich sind, vom Terminal während der Vorübersetzung angefordert werden. Der Subsystemanwender ruft also lediglich vom Terminal das Subsystem auf und wird dann aufgefordert, die erforderlichen Daten einzugeben. Dies kann in einfacher Weise dadurch erfolgen, daß der Anwender aufgefordert wird, aus einer Reihe von vorgestellten Auswahlmöglichkeiten eine Alternative auszuwählen (Menü-Technik).

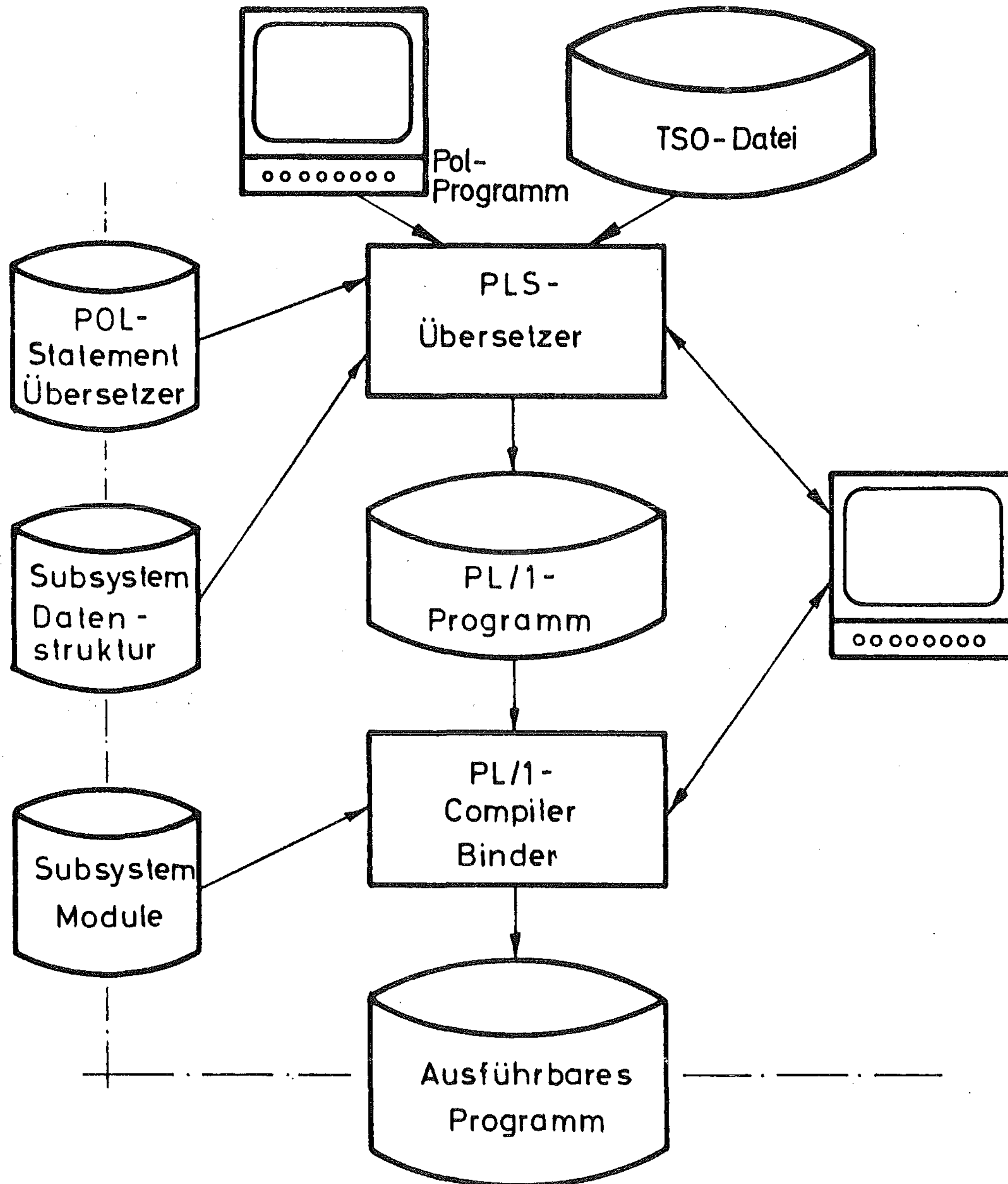


Abb. 16: Interaktive POL-Programm-Entwicklung

Wenn die Eingabedaten vollständig sind, kann das vorübersetzte Programm entweder in den Hintergrund abgesetzt oder im Vordergrund ausgeführt werden. Wenn in einer Subsystemanwendung der Rechenablauf festliegt und nur noch einzelne Steuerparameter verändert werden können, so ist es zweckmäßig, diese Daten vom Benutzer während der Ausführung durch Ein-/Ausgabeanweisungen anzufordern. Auch hier ist die Menütechnik anwendbar. Der Dialog wird vom im Vordergrund des TSO ausgeführten Programm gesteuert (Abb. 17).

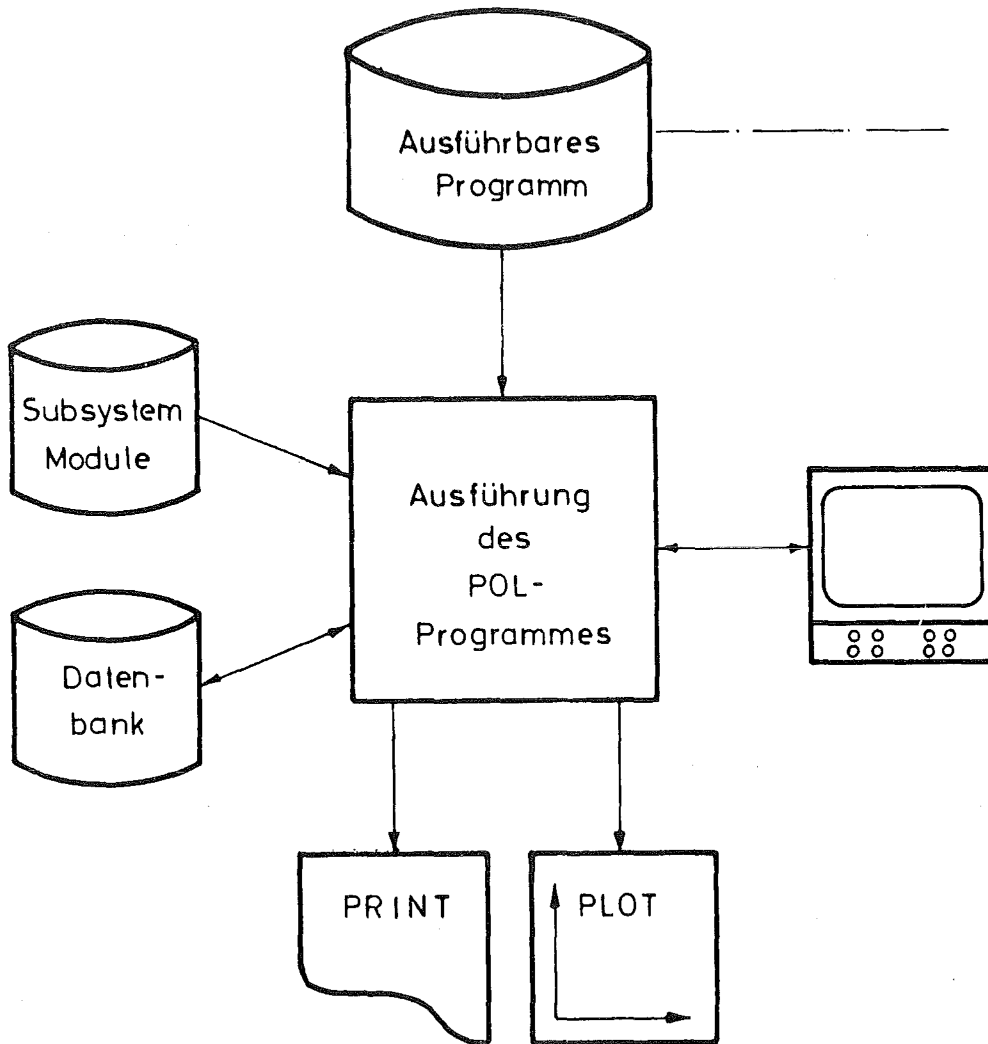


Abb. 17: Interaktive Programmausführung mit REGENT

Die Menütechnik ist besonders für das Arbeiten an einem Bildschirmterminal geeignet. Da der Anwender Schritt für Schritt angeleitet wird, die notwendigen Daten einzugeben und Entscheidungen über die gewünschte Alternative zu treffen, ist für diese Art des Dialoges wenig oder keine Vorbildung erforderlich. Die Methode kann erst dann nicht mehr vorteilhaft angewandt werden, wenn die Antwortzeiten zwischen den einzelnen Dialogschritten, bedingt durch längere Zwischenrechnungen, zu lang werden.

5. Subsystem - Sprachentwicklungen mit PLS

In diesem Kapitel werden die einzelnen Schritte der Subsystementwicklung aufgeführt und anhand des REGENT-Subsystems REMAC näher erläutert. Dabei werden vor allem Sprachdefinition und beispielhafte Subsystemanwendungen gezeigt. Anhand von Vergleichen werden die Vorteile der Anwendung einer problemorientierten Sprache verdeutlicht.

5.1 Fähigkeiten des REGENT-Subsystems REMAC

REMAC (REGENT Marker and Cell) ist ein Subsystem für die zeitabhängige zweidimensionale Berechnung von Strömungsvorgängen in inkompressiblen Flüssigkeiten nach der Marker and Cell - Methode. Es wurde aus einem am Los Alamos Scientific Laboratory erstellten FORTRAN-Programm entwickelt /67/. Der Ort und die Geschwindigkeit einer Flüssigkeit abhängig von der Zeit innerhalb eines Kontrollraumes kann berechnet werden. Abb. 18 zeigt die Kontrollraum-Geometrie.

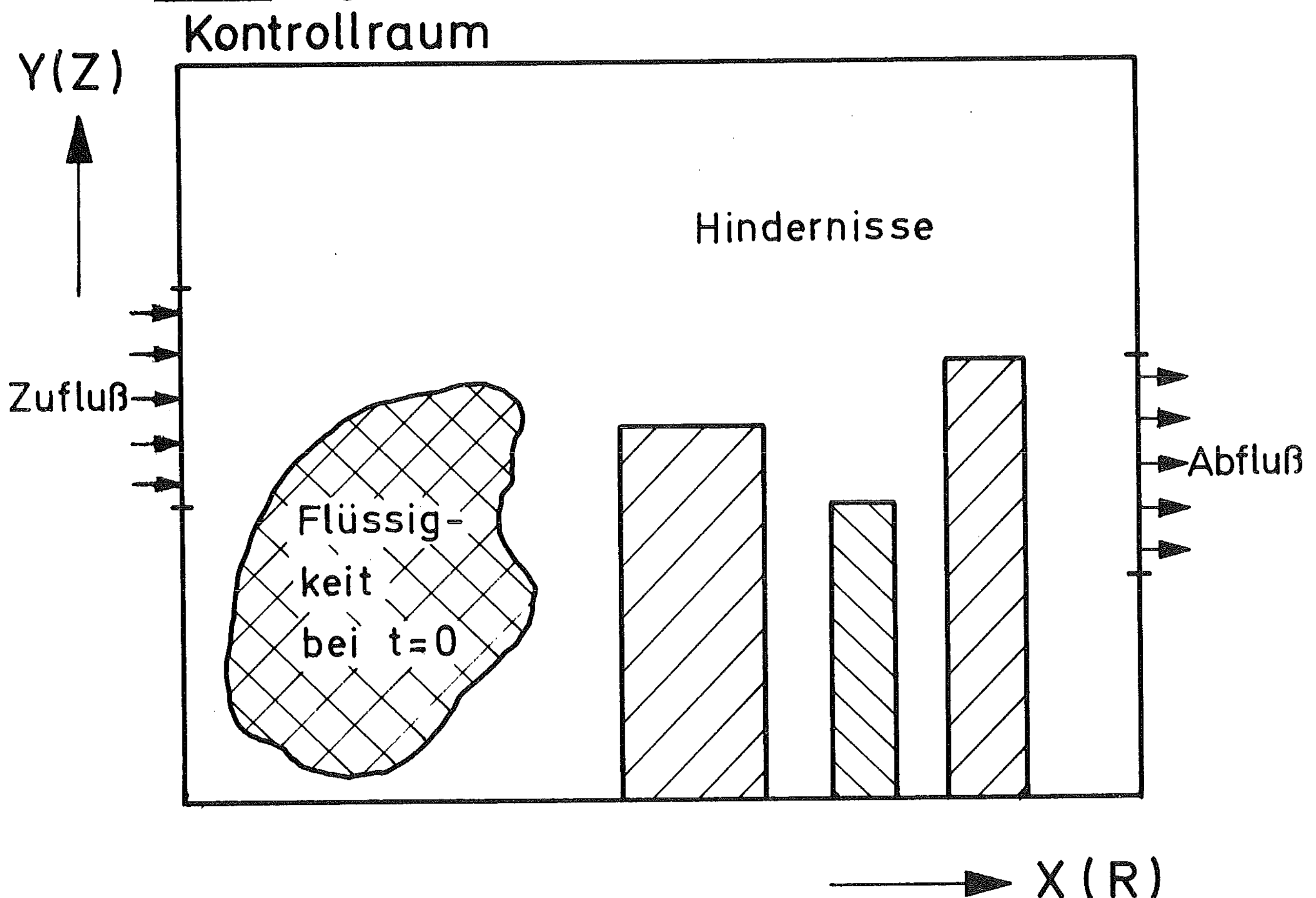


Abb. 18: REMAC - Geometrie

Es kann in einer ebenen X-Y-Geometrie oder einer rotationssymmetrischen R-Z-Geometrie gerechnet werden. Innerhalb des rechteckigen Kontrollraumes können bis zu drei ebenfalls rechteckige Hindernisse vorhanden sein. Der Ort und die ortsabhängige Geschwindigkeit von Flüssigkeit zur Zeit t=0 und die kinematische Viskosität der Flüssigkeit können vorgegeben werden. In ebener Geometrie kann an der linken Wand eine Zuflußöffnung und an der rechten Wand des Kontrollraumes eine Abflußöffnung vorhanden sein. Die Zuflußgeschwindigkeit muß in diesem Falle angegeben werden. Eine weitere Randbedingung ist das Vorhandensein bzw. Fehlen von Haftung an jeder der vier Wände. In X- und Y-Richtung (oder R- und Z-Richtung)kann jeweils eine konstante Gravitation g_x und g_y herrschen.

Außer den geometrischen und physikalischen Daten werden an das Programm Steuergrößen für den Gang der Rechnung (Maschenweite, Anfangs- und Endzeit der Rechnung, Iterationsschwellen) und zur Steuerung der Ausgabe der Ergebnisse (Druck- oder Zeichenausgabe in welchen Zeitabständen) übergeben.

Bei der Marker and Cell-Methode wird die Navier-Stokes-Gleichung unter Beachtung der Kontinuitätsgleichung in einem ortsfesten Maschennetz (daher: "Cell") für inkompressible Flüssigkeiten gelöst. Aus den Geschwindigkeitswerten für jede Masche bei einem Zeitpunkt werden die Geschwindigkeiten einen Zeitschritt später ausgerechnet. In diese Rechnung müssen die Randbedingungen an den freien Flüssigkeitsoberflächen eingehen. Da sich die freien Oberflächen im Raum bewegen, muß nach jedem Zeitschritt festgestellt werden, durch welche Maschen die Oberfläche geht. Dies wird durch fiktive, massenlose Markierungspartikel ("markers") in der Flüssigkeit erreicht. Die Marker werden zumZeitpunkt t=0 gleichmäßig in der Flüssigkeit verteilt, je nach der lokalen Geschwindigkeit, die in Abhängigkeit von ihren Ortskoordinaten zwischen den Geschwindigkeiten der umliegenden Maschen interpoliert wird, werden sie nach jedem Zeitschritt verschoben. Eine Oberflächenzelle ist dann eine Zelle, die selbst Marker enthält, aber eine Nachbarzelle ohne Marker besitzt. Die Marker können auch zur visuellen Veranschaulichung der Flüssigkeit dienen, wie die REMAC-Zeichnungen auf den Seiten 62, 63 und 69, 70 zeigen.

Die Klasse von Problemen, die mit REMAC gerechnet werden können, ist begrenzt (nur zweidimensionale Probleme, starre Geometrie, nur beschränkte Variationsmöglichkeiten der Rand- und Anfangsbedingungen). Die Anwendersprache muß deshalb auch nur die zugelassenen Möglichkeiten der Modellbildung wiedergeben können, diese allerdings mit größtmöglicher Benutzerfreundlichkeit und Anwendungssicherheit.

5.2 Schritte der Subsystementwicklung

Die Entwicklung von neuen REGENT-Subsystemen erfolgt in folgenden Einzelschritten, die speziell auf REMAC bezogenen Tätigkeiten sind jeweils zur Verdeutlichung aufgeführt.

(1) Initialisierung des Subsystems mit PLS, Festlegen des Namens und des Schlüsselwortes.

```
INITIATE SUBSYSTEM 'REMAC' KEY 'IRE6';
```

Das neue Subsystem erhält den Namen REMAC, bei Änderungen muß von nun an das Schlüsselwort IRE6 angegeben werden.

(2) Definition des Subsystem-Common und anderer Subsystemdatenstrukturen mit PLS. Der Subsystem-Common kann anschließend in allen Problemlösemodulen des Subsystems angesprochen werden.

```
DATASTRUCTURE COMMON;
DECLARE 1,
     2 VISCOSITY BINARY FLOAT,
     2 X_CELL_NUMBER BINARY FIXED,
     2 Y_CELL_NUMBER BINARY FIXED,
       .
       .
       .
END DATASTRUCTURE;
```

Alle für das Subsystem relevanten Daten, auf die verschiedene Module während der Rechnung zugreifen müssen, werden im Subsystem-Common gespeichert (z.B. kinematische Zähigkeit der Flüssigkeit, Maschenanzahl in X- und Y- Richtung).

(3) Kodieren der Problemlösemodule. Dies geschieht mit dem Modulgenerator der REGENT-Modulverwaltung in Verbindung mit dem PLR-Precompiler, der das Ansprechen von REGENT-Systemkernfähigkeiten in leichter Weise ermöglicht. Die Problemlösemodule werden in einer Bibliothek abgelegt, von wo sie bei der Ausführung des Subsystems durch die Modulverwaltung aufgerufen werden können.

Hier werden die Algorithmen zur Realisierung der REMAC-Fähigkeiten erstellt. Die in FORTRAN geschriebenen SMAC-Programme konnten als Grundlage genommen werden, so daß der Aufwand für die Programmierung reduziert wurde. Die Programme wurden in PL/1 neu programmiert und im wesentlichen in drei Module gegliedert: Initialisierung, Rechnung und Ausgabe. Außer der Konvertierung nach PL/1 wurden auch wesentliche Verbesserungen vorgenommen, darunter der Einbau einer Zeitschrittweiten-Automatik und Maßnahmen zur Verbesserung der numerischen Stabilität /68/. Die Datenstrukturen wurden mit Hilfe der REGENT-Dynamic-Arrays dynamisiert und dadurch die Problemgröße automatisch an den Speicherplatz angepaßt. Der Speicherplatzbedarf wurde geringer. Nach der Erstellung der Programmodule werden sie nicht mehr verändert, sie werden bei der Anwendung von REMAC für ein spezielles Problem lediglich benutzt.

(4) Definition der Subsystem-Sprache auf Papier. Die Syntax der POL-Anweisungen und die daraus zu erzeugenden PL/1-Anweisungen werden festgelegt.

Es handelt sich um die Definition zur Anwendung des Subsystems, also um die zur Kommunikation zwischen Ingenieur und DVA entscheidende Anwenderschnittstelle. Mit dieser Sprache können alle durch REMAC

gegebenen Fähigkeiten zur Lösung spezieller Problemstellungen benutzt werden. Die REMAC-Sprache muß daher die möglichen Modelle widerspiegeln. Daher wurde die Sprache an die Ausdrucksweise angepaßt, die bei der schriftlichen Modellbeschreibung für die von REMAC lösbare Problemklasse benutzt wird. Die Sprachspezifikation befindet sich im Anhang F.

(5) Programmieren der Anweisungsdefinitionen.
Die durch die Spezifikation festgelegte Syntax und Semantik der Sprache wird jetzt durch Definition mit dem PLS-Subsystem realisiert. Die Sprachedefinition wird im nächstem Abschnitt näher behandelt.

(6) Testen des Subsystems.

Natürlich wird diese Abfolge von Schritten bei der Entwicklung eines Subsystems nicht ein einziges Mal erfolgen, sondern sich öfter wiederholen. Wird z.B. der Subsystem-Common geändert (Schritt 2), so muß mindestens auch Schritt 3 wiederholt werden.

5.3 Entwicklung der REMAC-Steuersprache

Der erste Schritt der Entwicklung einer Subsystemsprache ist die Spezifikation der Anweisungen, die erforderlich sind, um die Eingabeparameter für das Subsystem zu beschreiben und den Ablauf der Rechnung zu steuern. Während die Algorithmen in einer für Arithmetik geeigneten Sprache programmiert wurden (FORTRAN oder, im Falle REMAC, PL/1), muß die Sprache zur Anwendung dieser Algorithmen die geometrischen und physikalischen Eigenschaften des Modells beschreiben können, das die Problemstellung wiedergibt. Die Anweisungen sollen problemnah und einprägsam sein. Entsprechend den zur Verfügung stehenden Subsystemmodulen und deren Steuerparameter muß die Umsetzung von Sprachanweisungen geplant werden. Die Spezifikation der REMAC-Sprache ist im Anhang F aufgeführt.

Zu der REMAC - Sprache gehören 14 Anweisungen:

CONTROLROOM	- Angabe der Kontrollraumgeometrie
CELL	- Maschenweite und -anzahl
OBSTACLE	- Lage und Größe der Hindernisse
INFLOW	- Zuflußbeschreibung
OUTFLOW	- Abflußbeschreibung
LIQUID	- Verteilung, Geschwindigkeit und Zähigkeit der Flüssigkeit
WALL SLIP	- Haftung der Flüssigkeit an den Wänden
GRAVITY	- Angabe der Schwerkraft
TIME	- Anfangs- und Endzeiten der Rechnung und Anfangszeitschrittweite
NUMERICAL PARAMETERS	- Numerische Parameter: Iterationsschwellen, Art der Zeitschrittweitensteuerung
SOLVE	- Starten der Rechnung
RESET	- Zurücksetzen der Parameter vor einer neuen Rechnung
PRINT	- Steuern der Druckausgabe
PLOT	- Steuern der Zeichenausgabe

Die Worte der REMAC-Sprache lehnen sich an die Ausdrücke an, die in der SMAC-Programmbeschreibung zur Modellbeschreibung der gerechneten Problemfälle verwendet wurde. Ein Beispiel soll dies verdeutlichen:

SMAC-Modellbeschreibung (/67/, S.35)	entsprechende REMAC-Sprachanweisung
gravity points straight up, $g_x=0$, $g_y=1.0$	GRAVITY X 0 Y 1.0;
the plots are at times t=0 and t=3	PLOT EVERY 3. ;
Rigid walls are free-slip	WALL SLIP FREE;
Fluid inflow: NX=NY=4,UL=4,L1=10,L2=18	INFLOW FROM 10 TO 18, VELOCITY 4;
Obstacle: L5=8,L6=16,L7=12	OBSTACLE FROM 8 TO 16, HEIGHT 12;

Man kann feststellen, daß die Eingabesprache für REMAC teilweise sogar verständlicher ist als die theoretische Modellbeschreibung.

Um die Sprachanweisungen mittels PLS realisieren zu können, muß die Übersetzungsvorschrift in PL/1 bekannt sein. Diese richtet sich nach den Anforderungen der Subsystemmodule. Die Übergabe von Daten an die Module des Subsystems REMAC erfolgt im wesentlichen über die Subsystem-Common-Datenstruktur, die mit der Anweisung DATASTRUCTURE COMMON definiert wurde. Für die CONTROLROOM-Anweisung (vgl. S. 142) heißt die Übersetzungsvorschrift:

a) Wenn in ebener Geometrie gerechnet wird, setze die Commonvariable PC = 1.0, wenn in Zylindergeometrie gerechnet wird, setze PC = 0.0.

b) Falls die Kontrollraumbreite und -höhe gegeben ist, dann setze R = Breite und Z = Höhe.

Aus der Syntax der Anweisung und dieser Vorschrift ergibt sich folgende STATEMENT-Definition:

```
STATEMENT 'CON.TROLROOM';                               Statementname
    SKIPB ID('GEO');                                    Füllwort Geometrie
    IF BIDENTIFIER('CYL') THEN EXEC PC=0.0;;            CYLINDRICAL
    ELSE IF BID ('PLA') THEN EXEC PC=1.0;;              PLANE
    ELSE EXEC PC=1.0;;                                  Standardwert
    SKIP(',');                                          Komma übergehen
    IF BID('SIZ') THEN DO;                              SIZE angegeben
        IF ID('R') | ID('X') THEN DO;                   R oder X ist als erstes
            EXEC R=NEXT_EXPRESSION;;                           angegeben
            SKIP ID ('Z');
            SKIP ID('Y');
            EXEC Z=NEXT_EXPRESSION;;
            END;
        ELSE IF ID('Z') | ID ('Y') THEN DO;             Z oder Y ist als erstes
            EXEC Z=NEXT_EXPRESSION;;                           angegeben
            SKIP ID('R');
            SKIP ID('X');
            EXEC R=NEXT_EXPRESSION;;
            END;
        ELSE DØ;                                        Weder R,X noch Z,Y sind
            EXECUTE;                                    angegeben
            R=NEXT_EXPRESSION;
            Z=NEXT_EX;
            END EXEC;
            END;
        END;
END STATEMENT;                                          Ende der Definition
```

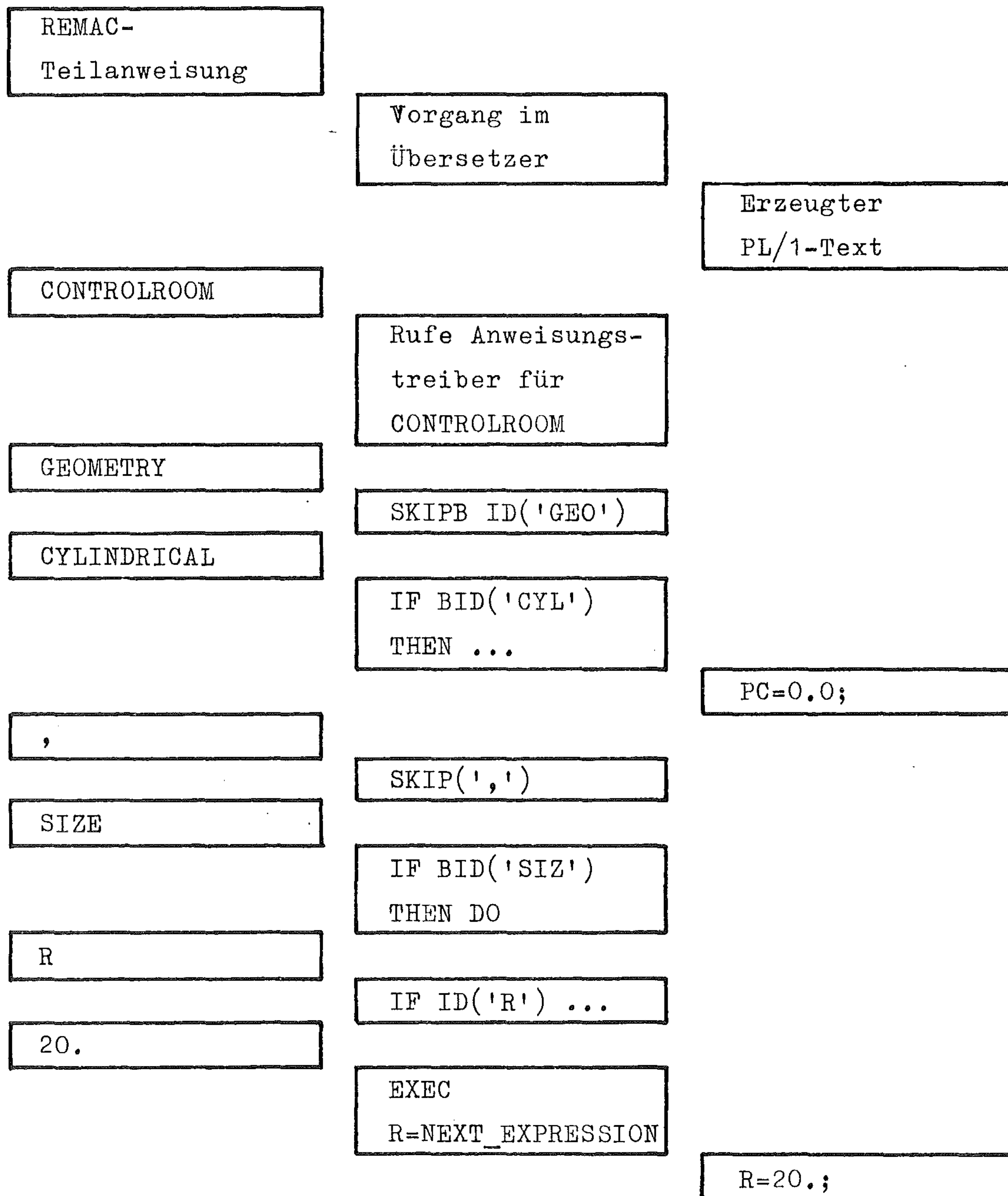
Zusätzlich zu der in der Spezifikation angegebenen Reihenfolge von R (Kontrollraumbreite) und Z (Kontorollraumhöhe) wird durch obige Definition auch die umgekehrte Reihenfolge erlaubt.

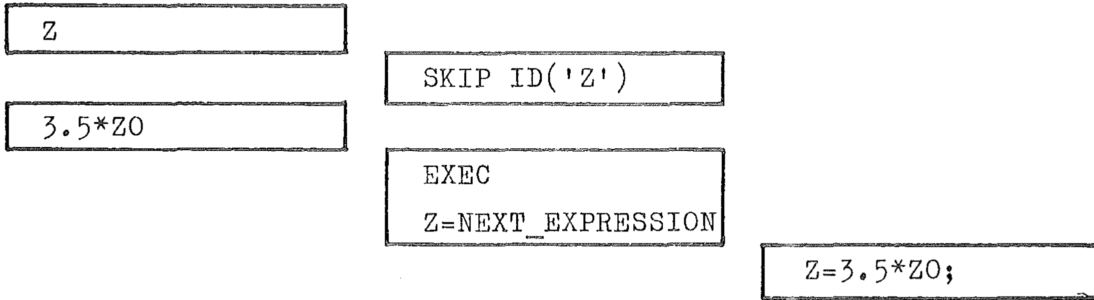
Der Ablauf der Analyse der POL-Anweisung, die Vorgänge im Übersetzer und die Erzeugung des PL/1-Textes sollen anhand einer CONTROLROOM-Anweisung erklärt werden.

Anweisung:

```
CONTROLROOM GEOMETRY CYLINDRICAL,
            SIZE R 20. Z 3.5*ZO;
```

Ablauf der Übersetzung:





5.4 Anwendung der REMAC-Sprache

Anhand des Anwendungsbeispiels eines fallenden Tropfens soll die Anwendung des Subsystems REMAC mit Hilfe der REMAC-Sprache gezeigt werden. Der Kontrollraum ist zylindrisch, er enthält weder Hindernisse noch einen Zu- oder Abfluß. Zur Zeit t=0 befindet sich ein Flüssigkeitstropfen unterhalb der Kontrollraumdecke auf der Zylinderachse. Eine Schwerkraft in -Z-Richtung beschleunigt den Tropfen bei t>0 in Richtung Boden, wo er auftrifft und zerplatzt (Abb. 19).

Die Zähigkeit der Flüssigkeit sei 10^{-6}, es soll mit 60*30 Maschen der Größe 0.25*0.25 gerechnet werden. Die Zeitschrittweite für die Rechnung soll 0.05 betragen, die Rechnung soll bei t=4 abgebrochen werden. Eine Zeichnungsausgabe auf Drucker und Zeichenmaschine soll alle 0.25 Zeiteinheiten geschehen.

Die hier angegebenen Daten und Abb. 19 stellen die Modellbeschreibung und Ablaufparameter in vom Menschen lesbarer Form dar. Für die Rechnung müssen diese Daten aber in maschinenlesbare Form gebracht werden. Im ursprünglichen FORTRAN-Programm wurde die Eingabe durch formatiertes Lesen von Werten vorgenommen, die Eingabe für das Problem "fallender Tropfen" sieht folgendermaßen aus:

```
   60    30      0.25      0.25      0.05      30       0.0      0.0
 FALLING DROP

1.-1.-1. 1.   1.E-4  2.E 4      0.0  - 9.81
   0.0     0.25      -1.      -1.       3.8    0
  0
  2     2      0.0      0.0
```

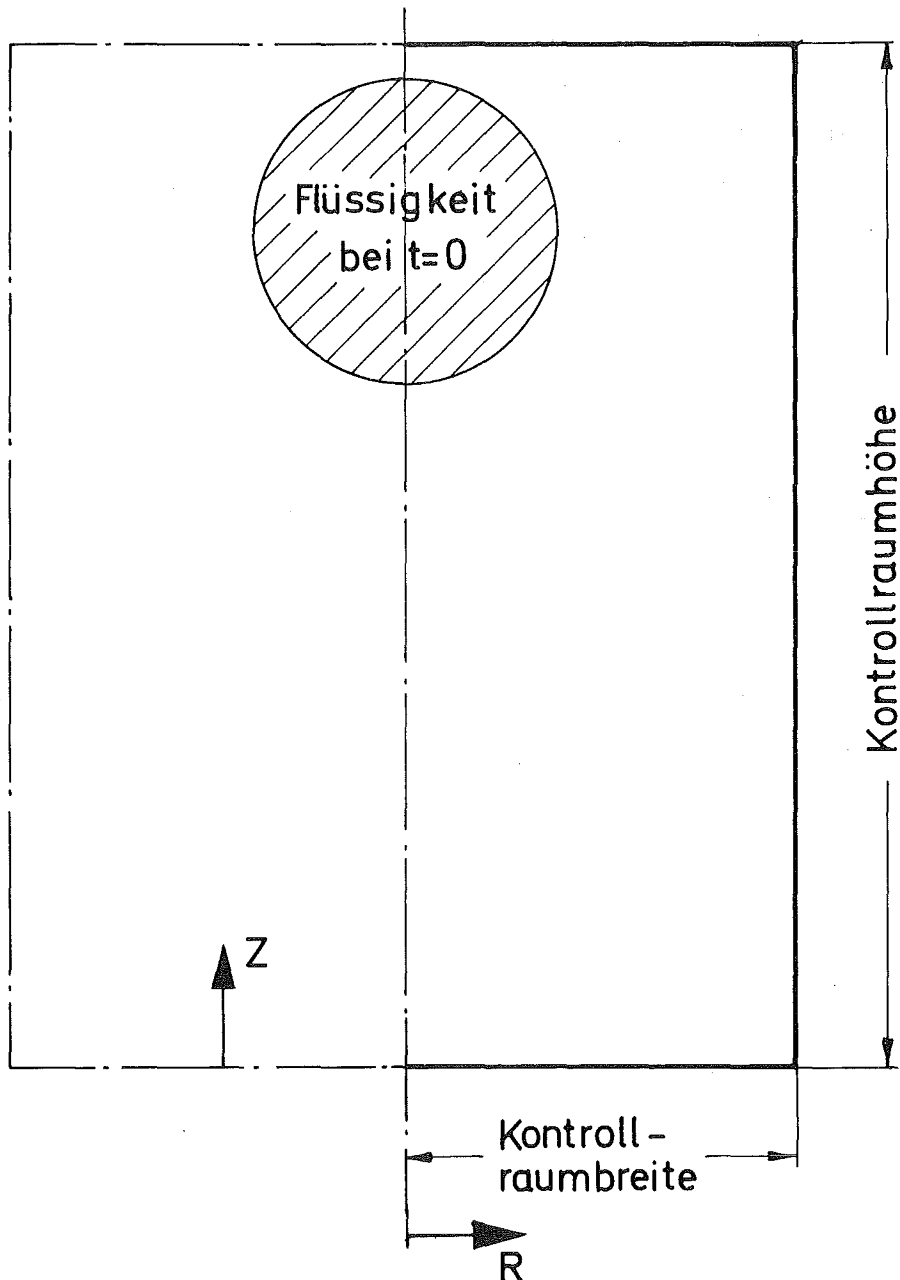


Abb. 19: REMAC-Beispiel, Tropfen zur Zeit t=0

Es ist sofort ersichtlich, daß es ohne weitere Unterlagen unmöglich ist:

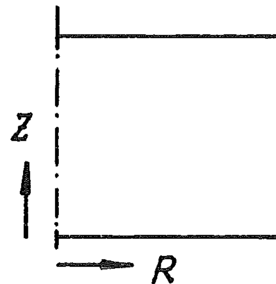
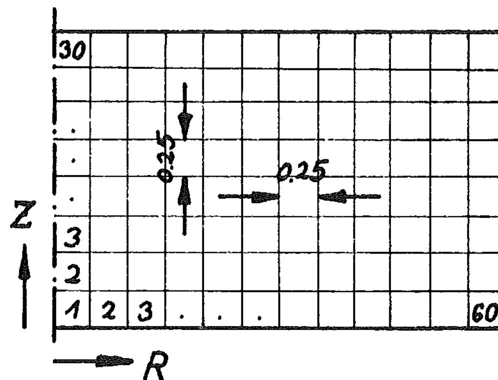
- den Zusammenhang zwischen der obigen schriftlichen Modellbeschreibung und der maschinenlesbaren FORTRAN-Eingabe herzustellen,
- zu überprüfen, ob diese Eingabe auch wirklich die Modellbeschreibung realisiert,
- Änderungen an bestimmten Modellparametern durchzuführen.

Selbst mit den nötigen Unterlagen (der Eingabebeschreibung) ist die Überprüfung oder Änderung der Eingabe umständlich und fehleranfällig. Wenn der Wert "30" auf der ersten Karte eine Spalte zuweit links steht, ist das auf dem gedruckten Abbild der Lochkarte kaum zu erkennen, für das Programm wird jedoch dadurch aus 30 der Wert 300. Die folgende Eingabe enthält z.B. drei Fehler, die erst nach genauem Vergleich mit obiger richtiger Eingabe festgestellt werden können:

```
   60  30       0.25      0.25      0.05      30       0.0       0.0
FALLING DROP

1.-1.-1.      1.E-4  2.E-4      0.0  -9.81
    0.0     0.25       -1.       -1.       3.8      0
   0
   2    2      0.0      0.0
```

Das gleiche Modell mit den gleichen Parametern wird durch die REMAC-Sprache folgendermaßen beschrieben:

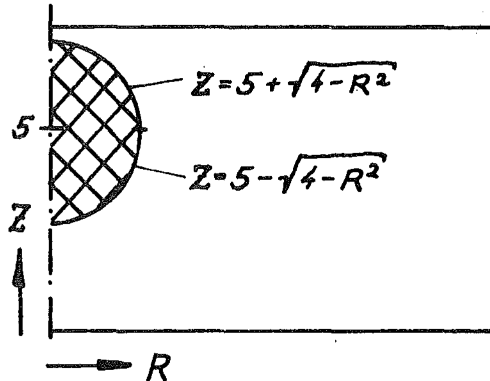
REMAC-Anweisungen	Auswirkungen
ENTER REMAC;	
NAME='FALLING DROP IN CYLINDER COORDINATES';	Titel des Problems
CONTROLROOM CYLINDRICAL;	
CELL NUMBER R 60 Z 30, SIZE R 0.25 Z 0.25;	

REMAC-Anweisungen | Auswirkungen

```
LIQUID VISCOSITY 1.0E-6;
```

Zähigkeit der Flüssigkeit

```
LIQUID DOMAIN (R<2)&((Z-5)>-SQRT(ABS(4-R*R)))
                    &((Z-5)< SQRT(ABS(4-R*R)));
    U0=0;
    V0=0;
LIQUID END;
```




```
TIME STEP 0.05, END 4.0;
```

Zeitschritt und Endzeit der Rechnung

```
PLOT DEVICE PRINTER, PLOTTER;
PLOT EVERY 0.25;
```

Ausgabesteuerung

```
SOLVE;
```

Starte Rechnung

```
END REMAC;
```

Es wird ersichtlich, daß REMAC dem Idealzustand:

Modellbeschreibung=maschinenlesbare Darstellung

sehr nahe kommt. Die Eingabe ist durch Verwendung einer POL gegenüber der FORTRAN-Eingabe quantitativ umfangreicher geworden, dies wird jedoch durch die Vorteile mehr als ausgeglichen:

- Die Eingabe läßt sich in dieser Form lesen und ohne große Mühe verstehen. Dadurch ist auch eine bessere Überprüfbarkeit gegeben.

- Durch das Fehlen von Spaltenkonventionen und der leichten Merkbarkeit von Anweisungen gegenüber Reihenfolge und Bedeutung von Werten auf Kartenspalten ist die Erstellung der Eingabe leichter, schneller und sicherer geworden.

- Da die Eingabe erst syntaktisch überprüft wird und erst bei richtiger Syntax gerechnet wird, kann die Anwendung der Steuersprache auch Rechenkosten sparen.

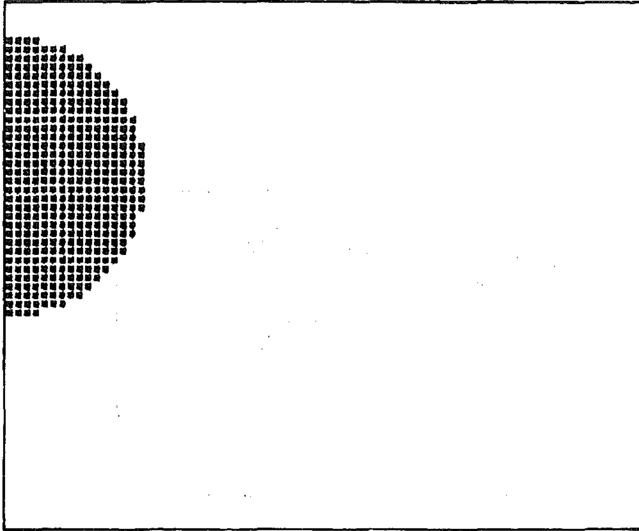


ABB : 01 FALLING DROP IN CYLINDER COORDINATES
CYCLE = 00000 TIME = 000.00

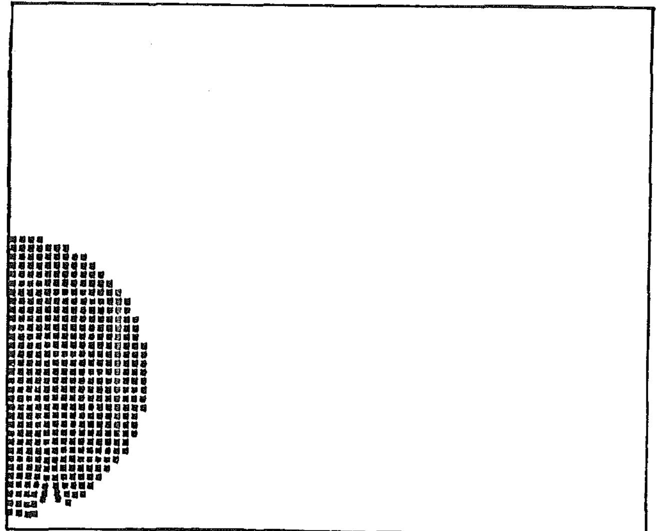


ABB : 10 FALLING DROP IN CYLINDER COORDINATES
CYCLE = 02000 TIME = 002.29

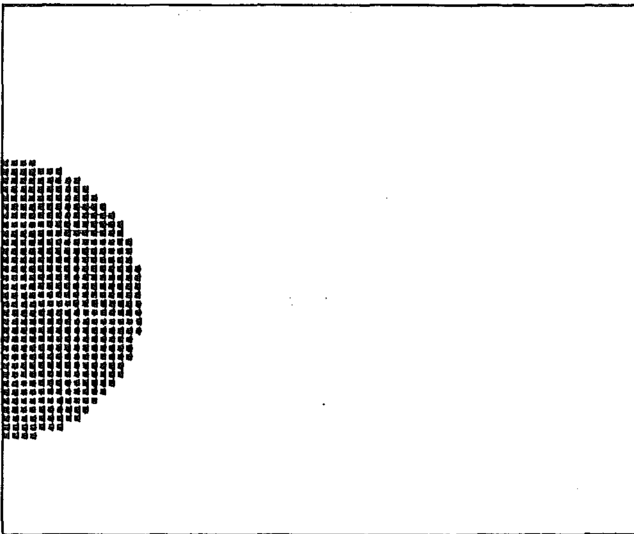


ABB : 08 FALLING DROP IN CYLINDER COORDINATES
CYCLE = 01300 TIME = 001.74

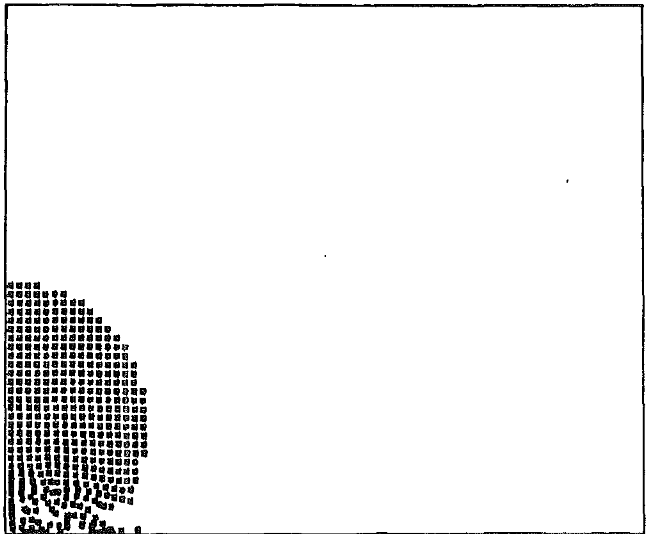


ABB : 11 FALLING DROP IN CYLINDER COORDINATES
CYCLE = 03200 TIME = 002.54

Abb. 20: Zeichenausgabe für das Anwendungsbeispiel "fallender Tropfen" (Teil 1)

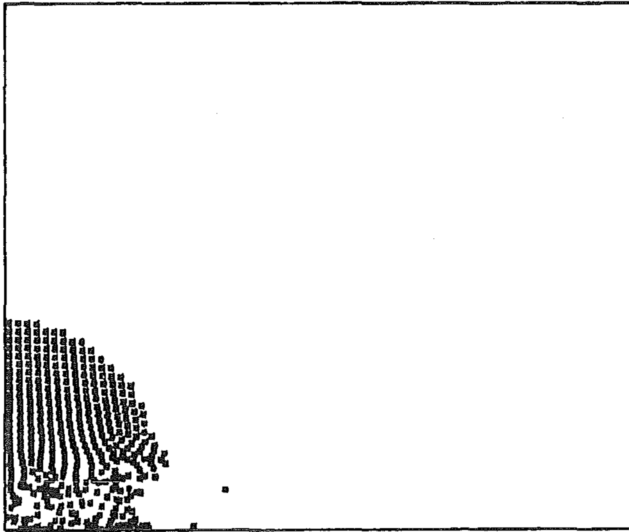


ABB : 12 FALLING DROP IN CYLINDER COORDINATES
CYCLE = 05300 TIME = 002.79

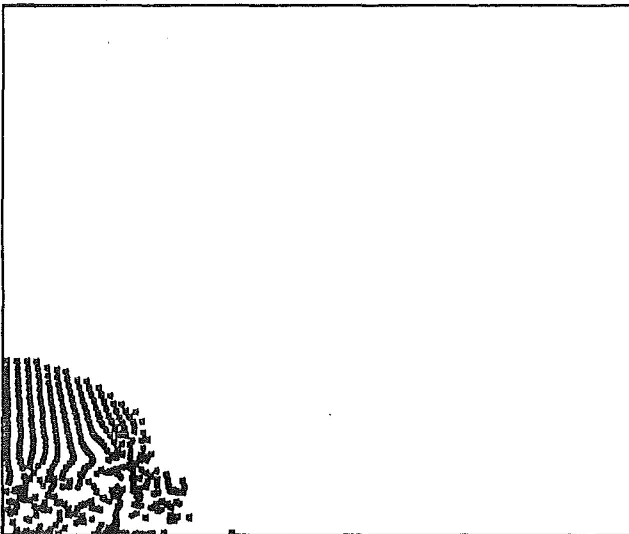


ABB : 13 FALLING DROP IN CYLINDER COORDINATES
CYCLE = 54500 TIME = 003.05

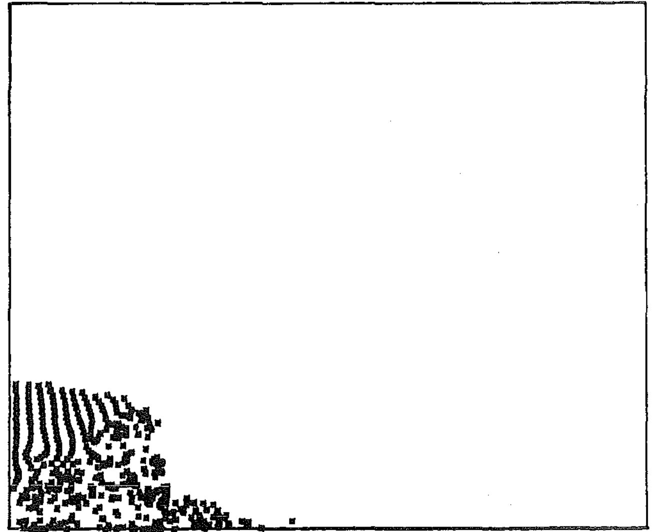


ABB : 14 FALLING DROP IN CYLINDER COORDINATES
CYCLE = 18600 TIME = 003.30

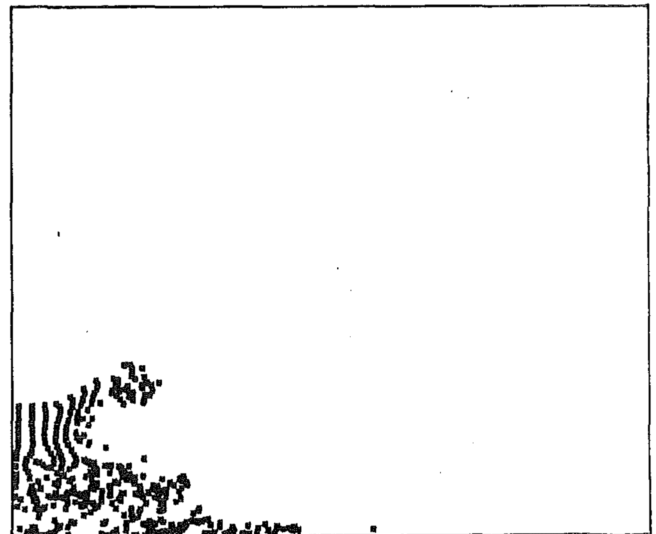


ABB : 15 FALLING DROP IN CYLINDER COORDINATES
CYCLE = 82700 TIME = 003.55

Abb. 20 (Teil 2)

Der letzte Vorteil gilt auch gegenüber interpretierenden Systemen wie ICES oder SEDAP, wo unter Umständen nach minutenlanger Rechnung ein Programm wegen eines einfachen Syntaxfehlers abgebrochen wird. Die Ausgabe des REMAC-Programms zeigt Abb. 20, diese Ausgabe ist von der Ausgabe des FORTRAN-Programmes nicht wesentlich verschieden, da wohl die Eingabeschnittstelle, aber nicht (oder kaum) das Programm selbst und die Ausgabeschnittstelle geändert wurde.

Wie alle REGENT-Subsystemsprachen bietet auch REMAC die Vorteile der Grundsprache PL/1: Eingabewerte für das Subsystem können innerhalb des POL-Programmes von beliebigen Einheiten eingelesen oder errechnet werden. Parametervariationen werden dadurch vereinfacht, daß die Problembeschreibung in ein Unterprogramm verlegt wird, das mit wechselnden Parametern aufgerufen werden kann. Durch die Fähigkeit, arithmetische und logische Ausdrücke zu verarbeiten, wurde es leicht möglich, beliebige Flüssigkeitsverteilungen zur Zeit t=0 zuzulassen, dagegen waren im ursprünglichen Programm nur kreisförmige oder rechteckige Flüssigkeitsverteilungen und Überlagerungen möglich.

Zwei Beispiele sollen die besonderen Eigenschaften der REMAC-Sprache noch veranschaulichen - die Möglichkeit beliebiger Flüssigkeitsverteilungen und die leichte Anwendung der Parametervariation.

Beispiel zur nichtrechteckigen Flüssigkeitsverteilung

Abb. 21 zeigt Anfangsverteilung und Anfangsgeschwindigkeit der Flüssigkeit.

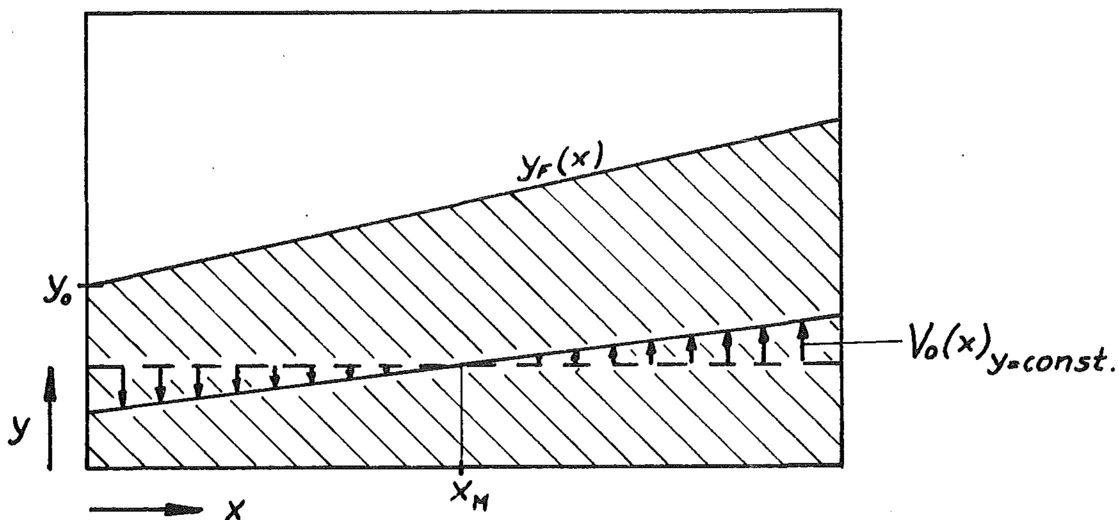


Abb. 21: Verteilung und Geschwindigkeit der Flüssigkeit bei t=0

=0 X-Komponente der Geschwindigkeit bei t=0. (1)

$_{\circ}=V_{o}(x,y)= {}_{\mid}(x-x_{M})y$ (2)

Die y-Komponente der Geschwindigkeit soll linear von x und y abhängen.

$_{F}=y_{o}+k_{2}x$ (3)

Die Flüssigkeit soll eine ebene, zur x-Achse geneigte Oberfläche haben

Sol .eses Pro em mit Hilfe der FORTRAN-Eingabe gelöst werden, erg ı sich fo ;ende Probleme:

- ·ınnen nur rechteckige Flüssigkeitsverteilungen angegeben werden.
- ·edem Flüssigkeitsbereich muß eine konstante Geschwindigkeit geben werden.
- ; Daten müssen als numerische Konstante spezifiziert werden.

Es ;t hervorzuheben, daß diese Einschränkungen nicht durch den Algori ıus oder das Problemlöseprogramm gegeben sind, sondern lediglich dı ı die primitive Eingabesprache. Um das Beispiel rechnen zu können, mı lie Flüssigkeit in kleine Bereiche konstanter Anfangsgeschwindigkı , aufgeteilt werden (Abb. 22).

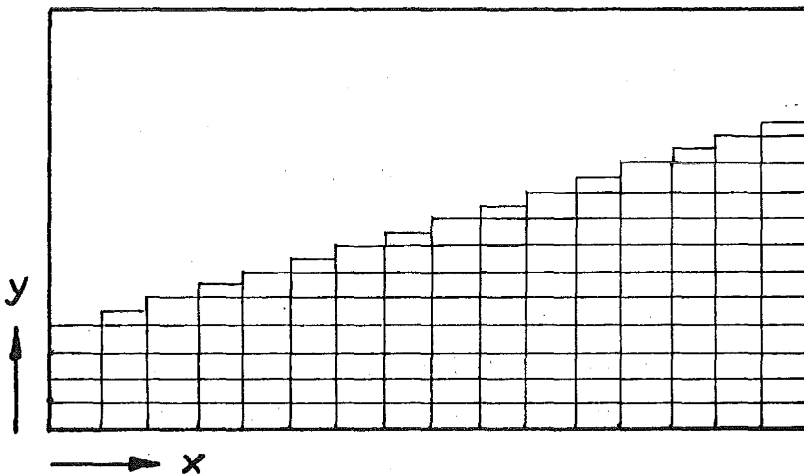


Abb. 22: Aufteilung in Einzelbereiche

Für jeden Bereich muß nun von Hand oder durch ein eigens vor der Rechnung zu erstellendes, zu testendes und auszuführendes Programm die mittlere Geschwindigkeit errechnet werden. Dabei sind die

Konstanten k_1, x_M, y_o, k_2 in ihrem Wert vor der Rechnung festzulegen, eine Änderung einer Konstante bedingt die Neuerrechnung der Geschwindigkeiten. Für jeden Flüssigkeitsbereich ist eine FORTRAN-Eingabekarte zu erstellen, z.B:
(k_1=1, x_M=10, Flüssigkeitsbereichsgröße 2*2)

x_{links}	y_{unten}	x_{rechts}	y_{oben}	U_o	V_o
0	0	2	2	0	-10
0	2	2	4	0	-30
0	4	2	6	0	-50
0	6	2	8	0	-70

etc., für jeden Bereich eine Karte, insgesamt also 128 Karten.

Bei Verwendung von REMAC können die Formeln (1) bis (3) praktisch unverändert übernommen werden, eine Aufteilung in Teilbereiche ist nicht erforderlich, eine Vorausberechnung entfällt, die Konstanten k_1, x_M, y_o, k_2 können leicht geändert werden, weil sie unmittelbar in der Eingabe erscheinen:

```
K1=1;                  }
XM=10;                 }
Y0=6;                  }    leicht änderbar
K2=0.5;                }
LIQUID DOMAIN Y<=Y0+K2*X;   vergleiche Gleichung (3)
U0=0;                       vergleiche Gleichung (1)
V0=K1*(X-XM)*Y;             vergleiche Gleichung (2)
LIQUID END;
```

Die direkte Entsprechung von Modellbeschreibung und maschinenlesbarer Darstellung wird hier wieder deutlich, der Zeitaufwand für das Erstellen der Eingabe dürfte hier um eine Größenordnung geringer sein als bei der Benutzung der FORTRAN-Eingabe. Soll dieses Problem mit wechselnden Konstanten mehrfach gerechnet werden, so brauchen bei Anwendung der REMAC-Sprache nur die ersten vier der obigen Anweisungen

ersetzt werden durch:

```
GET LIST(K1,XM,Y0,K2);
```

Das POL-Programm wird dann nicht mehr verändert, nur noch die vier Eingabeparameter ändern sich von Anwendung zu Anwendung ("parametric use").

Parametervariation

Am Beispiel eines brechenden Dammes soll die Verwendung eines Unterprogrammes zur Parametervariation gezeigt werden. Zum Beginn der Rechnung befindet sich Flüssigkeit in rechteckiger Verteilung zwischen $0 \leq X \leq Xmax$ und $0 \leq Y \leq Ymax$. Es herrscht eine Schwerkraft in -Z-Richtung, so daß sich die Flüssigkeit bei Zeiten > 0 in den Kontrollraum ergießt wie wenn ein zuvor vorhandener Damm plötzlich entfernt würde. Ein Hindernis bei $Xmax \leq X \leq Xmax + b$ und $0 \leq Y \leq h < Ymax$ hält jedoch einen Teil der Flüssigkeit zurück (es wird also nur der obere Teil des Dammes entfernt). Variiert wird in diesem Beispiel die Höhe h dieses Hindernisses (Abb. 23).

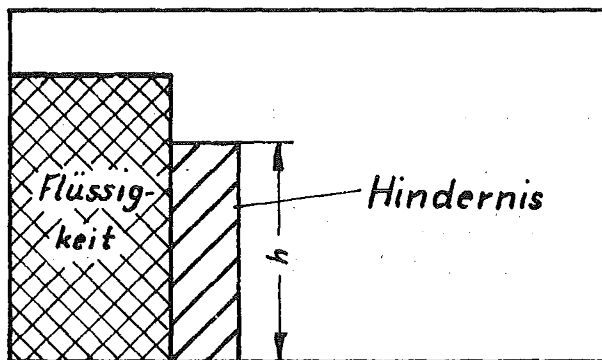


Abb. 23: Beispiel zur Parametervariation

Mit der FORTRAN-Methode muß für jede Hindernishöhe die gesamte Eingabe wiederholt werden. Obwohl sich nur der eine Parameter - Höhe des Hindernisses - ändert, müssen alle Angaben so oft angegeben werden, wie gerechnet werden soll. Abgesehen von der schlechten Übersichtlichkeit wird durch diese Redundanz auch die Fehleranfälligkeit erhöht. Im Gegensatz dazu kann bei REMAC die Rechnung durch <u>ein</u> Unterprogramm durchgeführt werden, das n-mal mit wechselndem Parameter für die Höhe h aufgerufen wird:

```
ENTER REMAC;
   DO I= 1 TO 10 BY(3);                  I nimmt die Werte 1,4,7 und
      CALL BROKEN_DAM(I);                10 an, Aufruf des Unterprogramms
   END;

BROKEN_DAM: PROCEDURE(H);              ⎫ H ist der Parameter des
   DECLARE H BIN FIXED(15);            ⎪ Unterprogramms
   NAME='BROKEN DAM, H=' ||H;          ⎪
   CONTROLROOM PLANE;                  ⎪
   CELL SIZE 0.5 0.5, NUMBER 30 15;    ⎬ POL-Unterprogramm
   PLOT EVERY 0.5;                     ⎪
   LIQUID DOMAIN X<3.5&Y<7.0;          ⎪
      UO,VO=0;                         ⎪
   LIQUID END, VISCOSITY 0.2;          ⎪
   OBSTACLE FROM 7 TO 8 HEIGHT H;      ⎪ Hindernishöhe=H
   TIME END 3.0;                       ⎪
   SOLVE;                              ⎪
   RESET;                              ⎪
END BROKEN_DAM;                        ⎭

END REMAC;
```

Für die Werte H = 1,4,7 und 10 wird jeweils das Unterprogramm BROKEN_DAM aufgerufen, das die Rechnung des Beispiels mit der jeweiligen Hindernishöhe steuert. Abb. 24 zeigt für die vier Varianten jeweils die Ausgangssituation und die Flüssigkeitsverteilung zum Zeitpunkt t = 2 und t = 4.

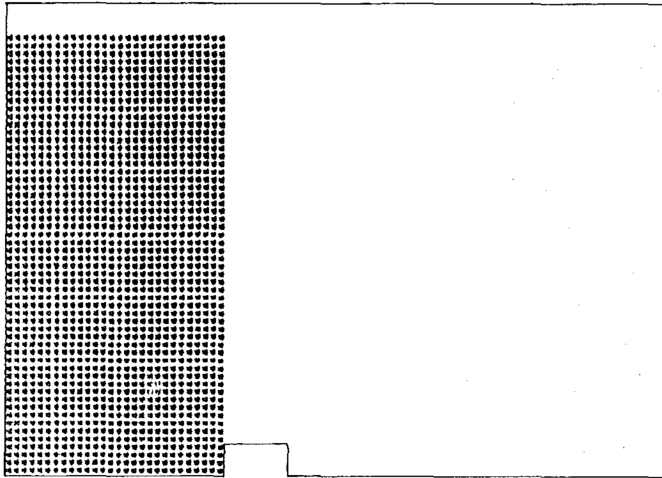


FIG. 1 BROKEN DAM, H= 1
CYCLE = 0 TIME = 0.00

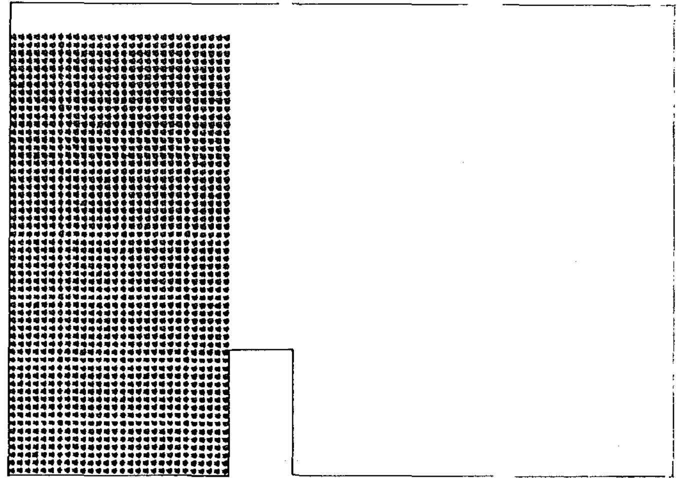


FIG. 1 BROKEN DAM, H= 4
CYCLE = 0 TIME = 0.00

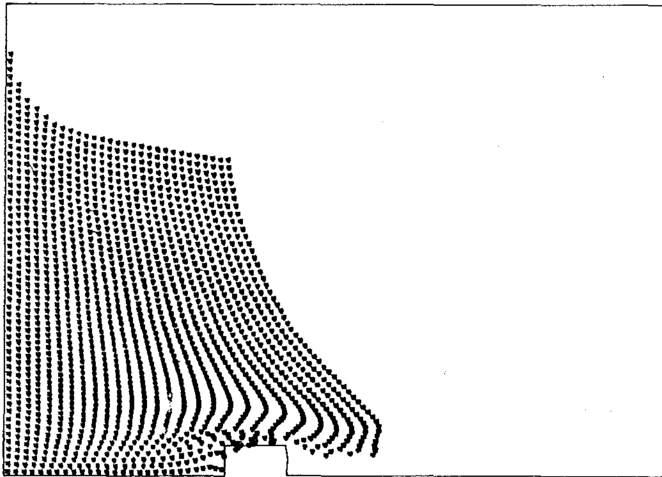


FIG. 5 BROKEN DAM, H= 1
CYCLE = 40 TIME = 2.00

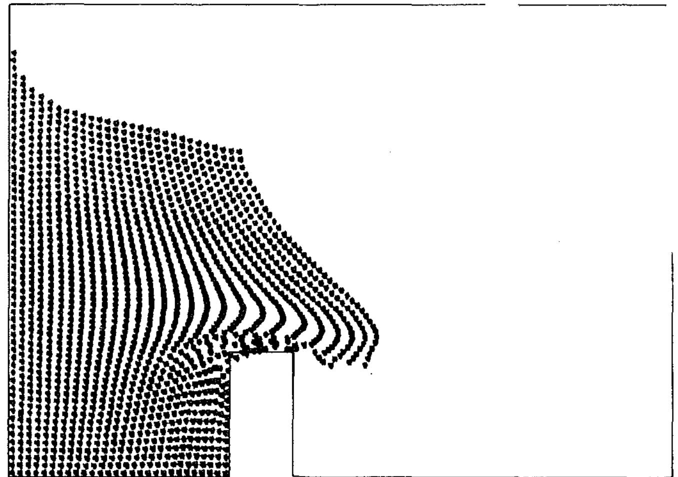


FIG. 5 BROKEN DAM, H= 4
CYCLE = 40 TIME = 2.00

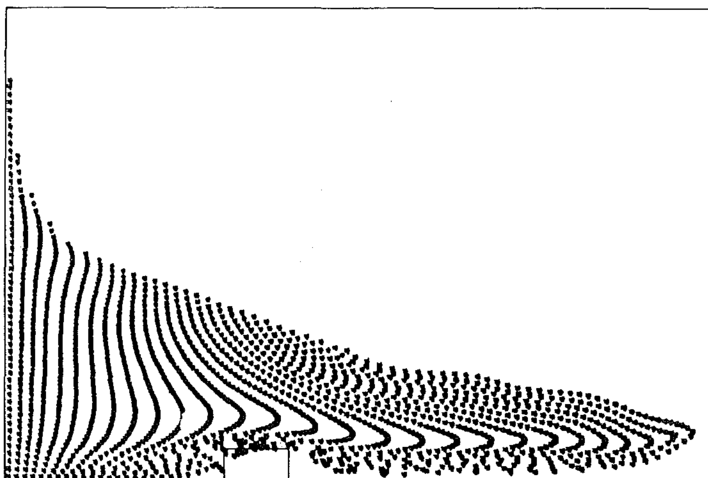


FIG. 9 BROKEN DAM, H= 1
CYCLE = 80 TIME = 4.00

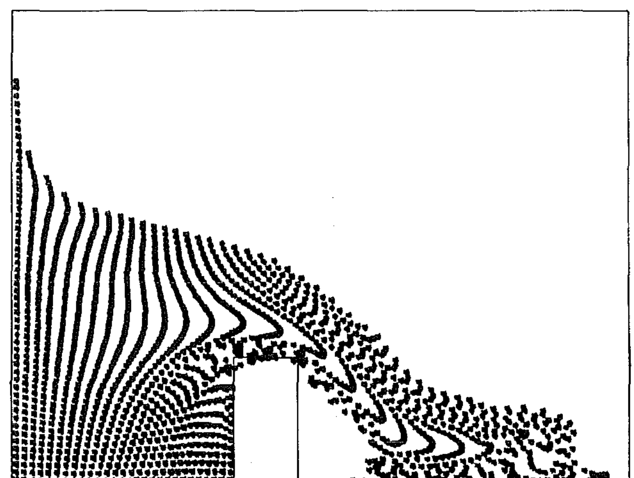


FIG. 9 BROKEN DAM, H= 4
CYCLE = 80 TIME = 4.00

Abb.24: Zeichnungen für das REMAC-Beispiel "brechender Damm" (Teil 1)

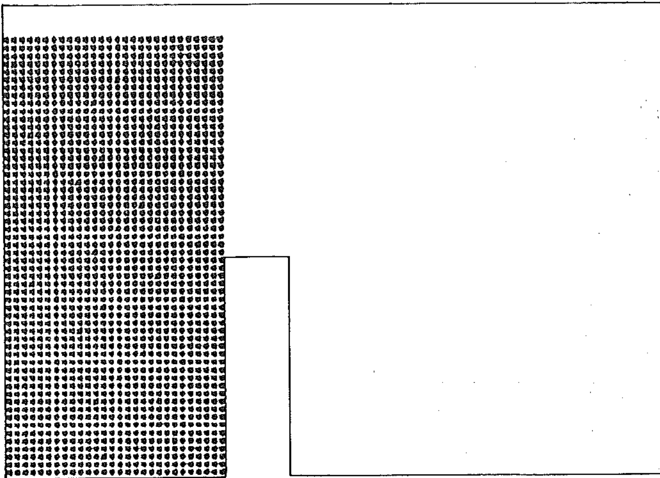


FIG. 1 BROKEN DAM, H= 7
CYCLE = 0 TIME = 0.00

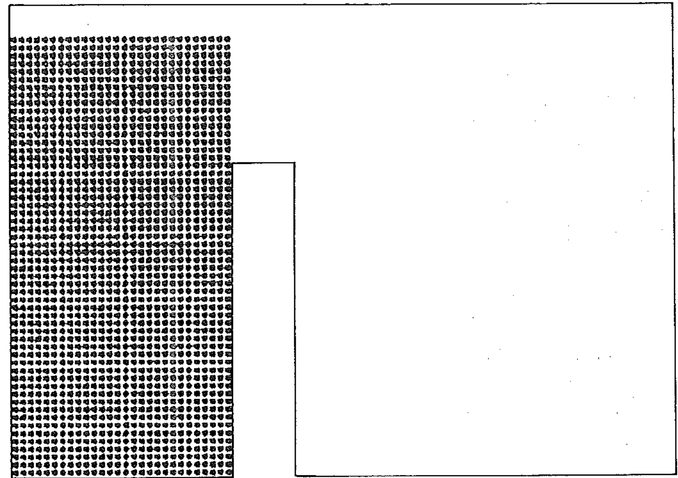


FIG. 1 BROKEN DAM, H= 10
CYCLE = 0 TIME = 0.00

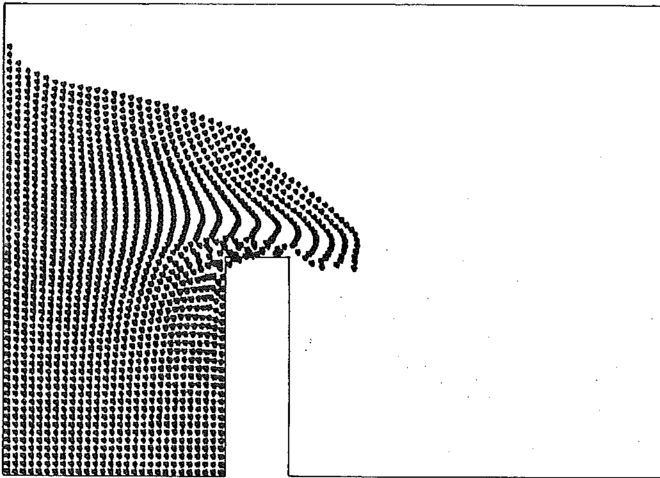


FIG. 5 BROKEN DAM, H= 7
CYCLE = 40 TIME = 2.00

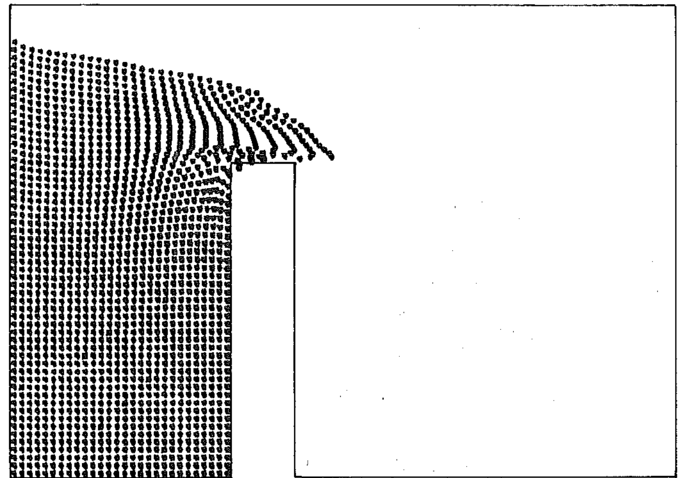


FIG. 5 BROKEN DAM, H= 10
CYCLE = 40 TIME = 2.00

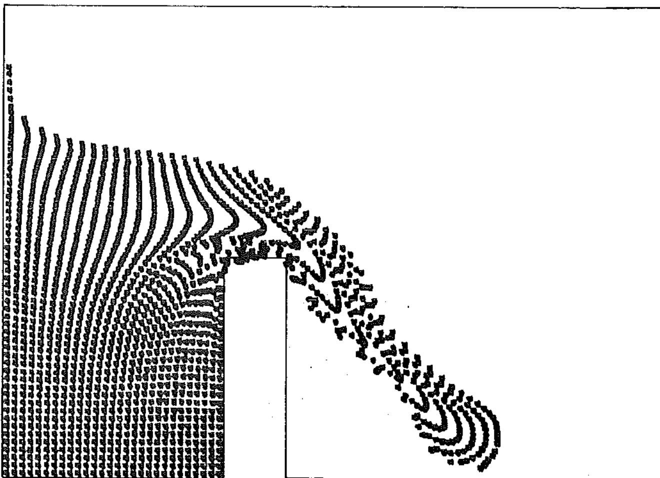


FIG. 9 BROKEN DAM, H= 7
CYCLE = 80 TIME = 4.00

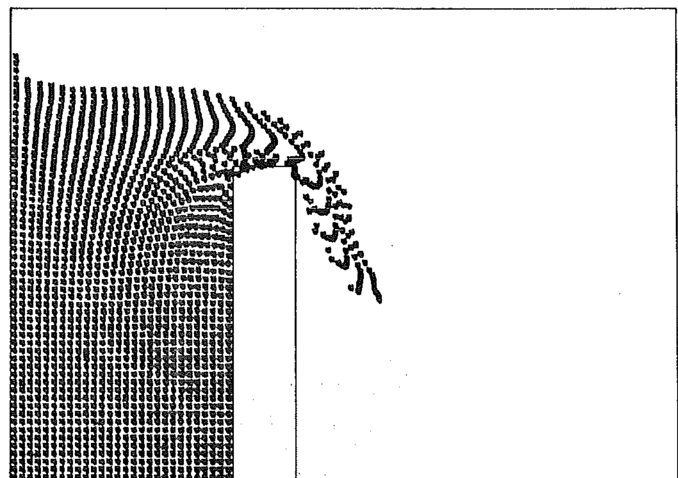


FIG. 9 BROKEN DAM, H= 10
CYCLE = 80 TIME = 4.00

Abb.24(Teil 2)

5.5 Andere REGENT-Subsysteme

Die umfangreichste Anwendung von PLS war bisher die Definition der Sprache für das REGENT-Subsystem GIPSY (Graphical Information Processing System, / 42 , 69 /). Mit GIPSY können zwei- und dreidimensionale grafische Objekte erzeugt, manipuliert und gezeichnet werden. Zu den zweidimensionalen Objekten gehören Punkte, Polygone, Zeichenketten, Koordinatenachsen und Diagramme (Funktionen, Approxemations- und Interpolationskurven). Zu den dreidimensionalen Objekten gehören Punkte, Polygone, Flächen und Körper. Die grafischen Objekte werden wie PL/1-Datentypen deklariert, mit "DCL P1(20) POLYGON (100);" wird z.B. ein Feld von 20 Polygonzügen mit je 100 Punkten deklariert. Die grafischen Objekte können in grafischen Anweisungen angesprochen werden, einem Polygonzug können z. B. Werte zugewiesen werden (Koordinaten der Stützstellen) oder Lineartransformationen können vorgenommen werden. Die Anweisung "SET P1(1) = ROTATE (P1(2), TETA,PSI, PHI);" weist dem Polygonzug P1(1) den Wert des um die eulerschen Winkel TETA, PSI und PHI gedrehten Polygonzuges P1(2) zu, wobei der Drehpunkt der Nullpunkt des Koordinatensystems ist. Durch eine PLOT-Anweisung kann die grafische Information ausgegeben werden, z.B.:

```
DO I = 1      TO   20;
   PLOT     ( P1(I) );
END;
```

Die Syntax der meisten GIPSY-Anweisungen ist rekursiv, da ein angesprochenes grafisches Objekt (Objektreferenz) eine grafische Funktion (wie ROTATE) sein kann, die ihrerseits als Argument ein grafisches Objekt erwartet. Die rekursive POL-Syntax wurde durch rekursive PL/1-Routinen in den STATEMENT-Definitionen realisiert.
Vor Fertigstellung von PLS war die GIPSY-Steuersprache unabhängig von REGENT in einer vereinfachten Form mit Hilfe des PL/1-Makroprozessors realisiert worden. Die Umstellung auf die POL-Definition mit PLS zeigte folgende Vorteile:

- Die Syntax ist flexibler geworden, statt DCL_POINT (A); DCL_POINT (B);" kann jetzt z.B. geschrieben werden: "DCL (A,B) POINT;", was eher der in PL/1 allgemein verwendeten Syntax von Deklarationen entspricht.

- Die Ausgabe der Vorübersetzung ist leserlicher. Aus den grafischen Deklarationen werden PL/1-Strukturdeklarationen erzeugt. Der PL/1-Makroprozessor ordnet die erzeugten PL/1-Anweisungen hintereinander ohne Absatz an, während es mit PLS möglich ist, den generierten PL/1-Text übersichtlich anzuordnen. Die folgende Gegenüberstellung zeigt jeweils eine Polygonzugdeklaration und den generierten PL/1-Text bei Verwendung von Preprozessor und PLS.

PL/1-Preprozessor:

```
DCL_POLY(P,20);



 DCL 1 POLYGON_P,2 P POINTER,2 TYPE BIN FIXED(15),2 SYMBOL BIN FIXED
(15),2 P_UP POINTER,2 LINETYPE BIN FIXED(15),2 NJEDER  BIN FIXED(15),2
 HEIGHT    DEC FLOAT(6),2 LENGTH    DEC FLOAT(6),2 OPEN_CLOSED BIT(1),2
PADDING BIT(15),2 N_MAX BIN FIXED(15) INIT(20),2 X(20,3) DEC FLOAT(6)
INIT CALL GROBINI(ADDR(POLYGON_P),3,1,1);
```

PLS:

```
DCL P POLYGON(20);

DECLARE   1 POLYGON_P ,
      2 P PTR,
      2 TYPE BIN FIXED(15),
      2 SYMBOL BIN FIXED(15),
      2 P_UP PTR,
      2 LINETYPE BIN FIXED(15),
      2 NJEDER BIN FIXED(15),
      2 HEIGHT DEC FLOAT(6),
      2 LENGTH DEC FLOAT(6),
      2 OPEN_CLOSED BIT(1),
      2 PADDING BIT(15),
      2 N_MAX BIN FIXED(15)INIT( 20 ),
      2 X( 20 ,3)DEC FLOAT(6) INIT
      CALL GROBINI(ADDR( POLYGON_P ),   3 , 1 ,1) ;
```

- Das Testen der PLS-STATEMENT-Definitionen ist wesentlich leichter als das Testen von PL/1-Preprozessor-Funktionen. Wegen des Fehlens jeglicher Ein/ Ausgabemöglichkeit können in Preprozessor-Funktionen bei Fehlern keine Test-Druckausgabe-Anweisungen verwendet werden. Aus demselben Grund können in Preprozessor-Funktionen bei fehlerhafter Syntax der POL-Anweisung keine Fehlermeldungen erzeugt werden.

- Die Übersetzung mittels PLS ist wesentlich schneller und billiger als mit dem Preprozessor.

Abb.25 zeigt die Ein- und Ausgabe eines GIPSY-Programmes. Dabei wird ein Polygonzug mit 5 Stützwerten deklariert und mit den Koordinaten der Ecken eines Quadrates gefüllt. Durch 270-maliges Zeichnen, Verdrehen und Verkleinern des Quadrates entsteht die gezeichnete Figur.

Weitere REGENT-Subsysteme, deren Subsystemsprachen mit PLS entwickelt wurden, sind FLODRA und YAQUIR. FLODRA/70/ erlaubt das Drucken und Zeichnen von Flußdiagrammen, die Sprach-Syntax wurde weitgehend von dem IBM-Programm FLOWCHAR/71/ übernommen. Abb.12 wurde von mit Hilfe von FLODRA erzeugt. YAQUIR ist ein Fluiddynamik-Subsystem zur Berechnung von Strömungsfeldern mit verschiedenen inkompressiblen Flüssigkeiten nach der Euler-Lagrange-Methode /72/. Für das Meßwertauswertesystem SEDAP /4/ wird mit Hilfe von PLS eine flexiblere Steuersprache entwickelt, um die Nachteile der interpretativen Verarbeitung formatgebundener Kommandos zu vermeiden /73/.

```
ENTER GIPSY;
  DCL A POLY2(5),                          /* QUADRAT 2-DIMENS.  */
      P POINT2;                            /* PUNKT 2-DIMENS.    */
  X1=0.03; X2=0.17; /* METER */
  SET A=POLY(COLL(POINT2(X1,X1)            /* KOORDINATEN        */
            +POINT2(X1,X2)                 /* DER                */
            +POINT2(X2,X2)                 /* QUADRATECKEN       */
            +POINT2(X2,X1)
            +POINT2(X1,X1)));
  SET P=POINT2(0.10,0.10);                 /* DREHPUNKTKOORD.    */
  OPEN PLOT DIN A 4;                       /* ZEICHENGROESSE     */
  DO I=1 TO 270;
      PLOT (A);                            /* ZEICHNEN           */
      SET A=ROTATE2(SCALE2(                /* SKALIERE MIT FAKT.*/
      A,0.99,0.99,P),1,P);                 /* 0.99 MIT ZENTRUM P*/
                                           /* UND DREHE 1 GRAD  */
  END;                                     /* UM PUNKT P         */
END GIPSY;
```

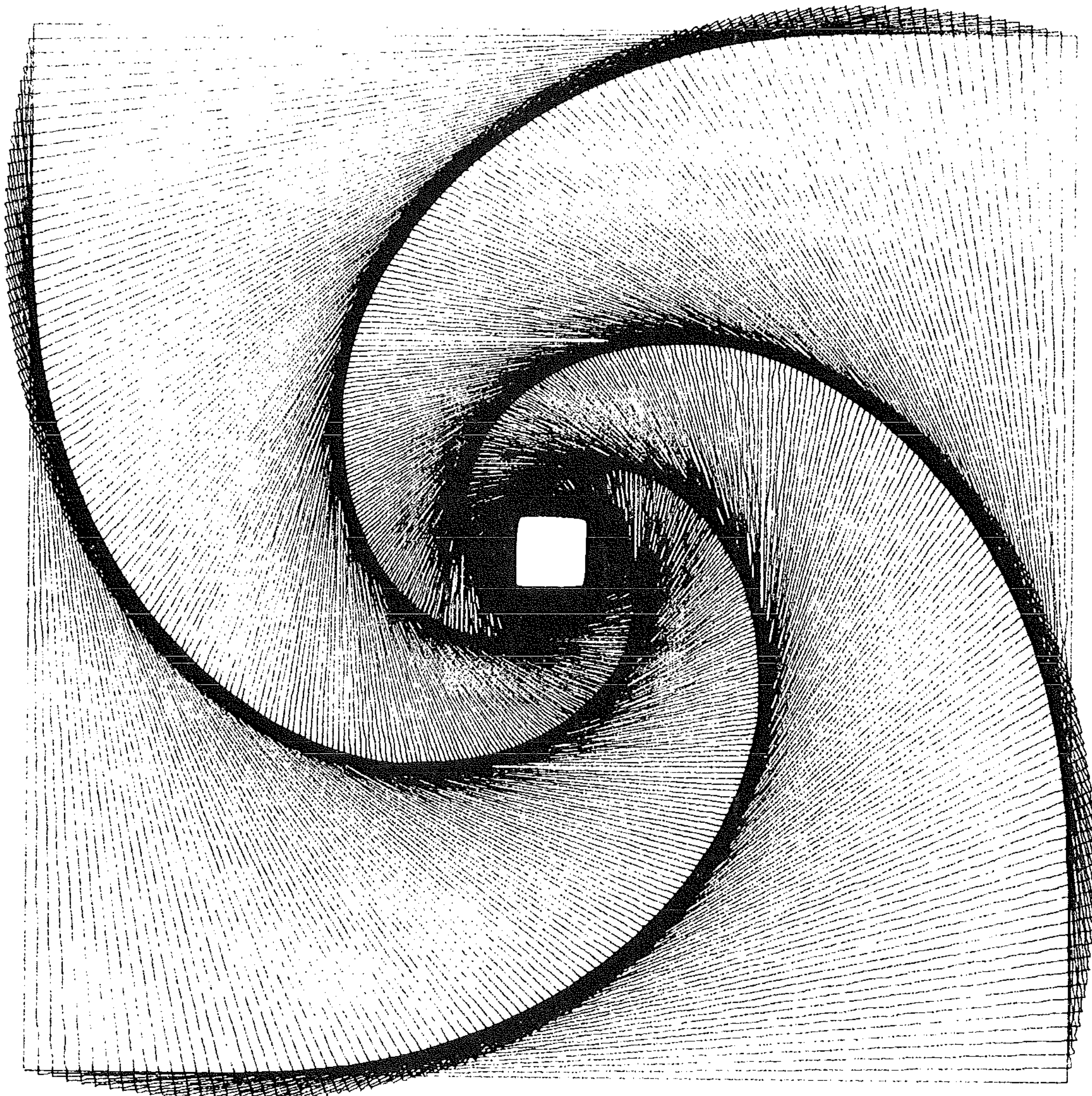


Abb. 25: Ein- und Ausgabe eines GIPSY-Programmes

6. Effektivitätsbetrachtung

Hauptaspekte der Effektivität bei der Anwendung von PLS sind benötigte Rechenzeiten, Arbeitsspeicherbedarf sowie der Zeitaufwand zum Programmieren und Testen. Dies gilt sowohl für die Sprachdefinition als auch für die Sprachübersetzung. Aus der Rechenzeit, dem Arbeitsspeicherbedarf und der Belegung von externen Einheiten werden nach einem in /74/ beschriebenen Abrechnungsalgorithmus die Kosten für einen Programmlauf berechnet. Die im folgenden angegebenen Rechenkosten sind nach diesem Algorithmus bestimmt, die Rechenzeiten und Arbeitsspeicheranforderungen beziehen sich auf die DVA IBM 370/168. Bei der Übersetzung von problemorientierten Sprachen muß nacheinander der PLS-Übersetzer und der PL/1-Compiler aufgerufen werden. Der Übersetzer wurde mit dem IBM-PL/1-Optimising Compiler unter Verwendung der Optimierungs-Option übersetzt. Der Arbeitsspeicherbedarf setzt sich wie folgt zusammen:

Programmcode des Monitors:	35 KB
Statischer Datenbereich des Monitors:	4 KB
Dynamischer Datenbereich des Monitors:	10 KB
Puffer für Ein-/Ausgabe	16 KB
Treibermodule für das PLS-Subsystem:	je nach POL-Anweisung, 1 - 30 KB

(1 KB = 1Kilobyte = 1024 Byte)

Im allgemeinen ist für die Übersetzung von POL-Programmen ein Arbeitsspeicherbereich von 120 KB ausreichend.

Bei der Übersetzung von PL/1-Anweisungen werden etwa 75 bis 100 Anweisungen/sec verarbeitet. Bei Programmen, die ausschließlich aus POL-Anweisungen bestehen, hängt die Übersetzungsrate von den verwendeten Anweisungstreibern ab. Für das REGENT-Subsystem REMAC wurden im Mittel 30 Anweisungen/sec verarbeitet. Zum Vergleich: der PL/1-Optimising Compiler kompiliert etwa 30 bis 120 Anweisungen/sec, je nach der Komplexität des Programms, dem zur Verfügung stehenden Arbeitsspeicherbereich und den verwendeten Compiler-Optionen. Der ICES-Interpretierer verarbeitet etwa 30 bis 50 Anweisungen/sec.

Um bezüglich der Rechengeschwindigkeit und der Kosten einen direkten Vergleich mit einem bestehenden Makroprozessor durchführen zu können, wurden Anweisungen zur Erzeugung von Zeichnungen einmal mit Hilfe von

PLS und einmal mit dem PL/1-Makroprozessor übersetzt. Da die Sprache für das Zeichensystem GIPSY zunächst mit dem PL/1-Preprozessor realisiert worden war, bot es sich für einen direkten Kostenvergleich an. Eine Folge von fünf Testprogrammen mit GIPSY-Anweisungen wurde mit dem Preprozessor und mit PLS übersetzt, dabei entstanden folgende Kosten und Rechenzeiten:

	Preprozessor		PLS	
	Zeit	Kosten	Zeit	Kosten
Vorübersetzen	117 sec	57.90 DM	10 sec	6.00 DM
Kompilieren	15 sec	8.70 DM	15 sec	8.70 DM
Insgesamt	132 sec	66.60 DM	25 sec	14.70 DM

Die hohen Werte für den PL/1-Preprozessor rühren daher, daß für jede Anwendung die Preprozessor-Funktionen neu übersetzt werden müssen.

Für das REMAC-Anwendungsbeispiel "fallender Tropfen" betrugen die Rechenzeiten und -kosten:

	t_{CPU}	Kosten
Vorübersetzen	0.92 sec	0.56 DM
Kompilieren	1.01 sec	0.57 DM
Ausführung	5 min 55 sec	160.00 DM

In diesem Fall sind also die Zeiten und Kosten für die Sprachübersetzung vernachlässigbar gegenüber den Rechenzeiten und -kosten der Programmausführung. Dies gilt für alle REMAC-Anwendungen, da hier die Steuersprache relativ einfach und der Rechenalgorithmus sehr zeitaufwendig ist.

Um über den Aufwand für eine Sprachdefinition genauere Aussagen machen zu können, wurde über die Zeiten, die benötigt wurden, um die REMAC-Sprache zu entwerfen, zu kodieren und zu testen, Buch geführt.

Die folgende Aufstellung zeigt die Zeiten, die für die verschiedenen Schritte der Sprachentwicklung mittels PLS benötigt wurden und die zum Testen benötigten Rechenkosten.

Sprachspezifikation		6.7 Mannstunden
Kodierung		4.0 Mannstunden
Testen	Zeitaufwand	3.6 Mannstunden
	Rechenkosten	525.60 DM

Bei einem angenommenen Arbeitsstundenpreis von DM 50.00 kostete die REMAC-Sprachdefinition insgesamt DM 1240.60. Dieser Aufwand macht sich dann bezahlt, wenn bei jeder REMAC-Anwendung Arbeitszeit und/oder Rechenkosten gespart werden. Nimmt man etwa an, daß für die Vorbereitung der Eingabe und die Fehlerkorrektur für eine Anwendung mit PLS insgesamt eine Stunde benötigt wird und die gleiche Anwendung mittels Lochkarteneingabe zwei Stunden Zeit erforderlich wäre, hätte sich die Investition für die Sprachdefinition nach 25 Anwendungen gelohnt. Allerdings wurden Vergleichsmessungen darüber, wie sich der Einsatz einer POL im Zeitaufwand niederschlägt, nicht angestellt, so daß obige Annahme (50% weniger Aufwand) nur eine grobe Schätzung bleibt.

7. Zusammenfassung der wichtigsten Ergebnisse

Innerhalb des REGENT-Systems für das rechnerunterstützte Entwickeln und Konstruieren wurde das Teilsystem PLS zum Definieren neuer problemorientierter Sprachen in einer flexiblen Syntax und zum Übersetzen solcher Sprachen entwickelt.

Die Definition problemorientierter Sprachen, die der in einem Fachgebiet verwendeten Fachsprache nahekommen, ist mit überschaubarem Aufwand möglich. Die Sprachen sind formatfrei und in ihrer Syntax nicht an starre Regeln gebunden. Durch die Möglichkeit, PL/1-Anweisungen innerhalb von POLs zu verwenden, sind Schleifen, bedingte Anweisungen und Unterprogramme zur Ablaufsteuerung verfügbar. Ein-/Ausgabeanweisungen ermöglichen das Lesen und Schreiben von Daten und das Drucken von Ergebnissen in allen problemorientierten Sprachen. Die anwendungsbezogenen Anweisungen können anstelle von numerischen Konstanten auch Variable oder arithmetische Ausdrücke verarbeiten. Das Ziel, daß auch ein Ingenieur ohne EDV-Kenntnisse die Definition problemorientierter Sprachen durchführen können sollte, wurde nicht erreicht. Mindestens die Kenntnis der Programmiersprache PL/1 ist erforderlich, Grundkenntnisse in Programmiertechniken (Listenverarbeitung, Datenstrukturen, Makroanwendung) sind von Vorteil. Dagegen ist die Anwendung eines REGENT-Subsystems mit Hilfe einer problemorientierten Sprache ohne EDV-Kenntnisse möglich.

Der Übersetzer ist bezüglich des Rechenzeitbedarfs anderen Makroprozessoren oder Interpretierern vergleichbar oder überlegen. Trotzdem der gesamte Übersetzer und auch das Definitionssystem in der höheren Programmiersprache PL/1 geschrieben wurde, reicht ein Arbeitsspeicherbereich von 120 KByte aus, um problemorientierte Sprachen zu übersetzen und zu definieren. Diese Speichergröße ist zur Anwendung integrierter Systeme mindestens erforderlich. Stellt der Übersetzer Fehler in POL-Programmen fest, werden ausreichende und lesbare Fehlermeldungen erzeugt.

Sowohl die Definition als auch die Übersetzung von POLs kann interaktiv von einem Terminal aus erfolgen. Die direkte Interpretation von POL-Anweisungen ist jedoch nicht möglich. Ein interaktiver Compiler für diese Aufgabe wurde wegen des dafür erforderlichen erheblichen Aufwandes nicht erstellt.

Die Anwendung problemorientierter Sprachen zur Steuerung von Subsystemanwendungen im REGENT-System hat gegenüber anderen Techniken eine Reihe von Vorteilen: Die Programme sind leicht lesbar und somit ohne großen Aufwand überprüfbar. Die Anwendung wird dadurch sicherer. Da eine POL-Anweisung leicht merkbar ist, wird auch die Erstellung der Eingabe für einen Anwendungsfall leichter, schneller und sicherer. Das gleiche gilt für Änderungen an der Eingabe bei Parameterstudien. Die erzwungene syntaktische Überprüfung der Eingabe erkennt Fehler bevor die Ausführung der Rechnung begonnen wird. Dies kann auch zu Kosteneinsparungen führen.

Das System PLS baut auf den positiven Errungenschaften anderer integrierter Systeme (ICES, IST, GENESYS) auf und realisiert die gewünschten Sprachverarbeitungs-Funktionen unter Berücksichtigung neuerer Forschungsergebnisse aus der Informatik (Compilerbau und Entwicklung von Makroprozessoren). Obwohl die benutzten Verfahren zum großen Teil schon bekannt waren, ist die Kombination:

- Problemorientierte Sprachen mit hochstehenden Eigenschaften,
- Leichte Definierbarkeit mit Verfügbarkeit des vollen PL/1-Sprachumfangs auf allen Sprachebenen,
- Langfristige Speicherung der Sprachdefinitionen als ausführbare Sprach-Übersetzungsmodule,
- Umschaltbarkeit von einer Sprache zur anderen durch eine hierarchische Blockstruktur von Subsystemen

neu und ohne Vorbild.

Die Arbeiten zur Erstellung von PLS sind abgeschlossen, das PLS-Teilsystem ist in das Gesamtsystem REGENT integriert. Außer Verbesserungen der Effektivität durch Neuprogrammierung kritischer Teile (wenn erforderlich in Assembler) müssen vor allem die interaktiven Fähigkeiten noch besser unterstützt werden. In der gegenwärtigen

Version werden lediglich die Fähigkeiten des TSO zur interaktiven POL-Programm-Erstellung ausgenutzt, die einige sehr unangenehme Eigenschaften haben. So wird bei Auftreten von Fehlern die gesamte Programmanwendung beendet und der Benutzer muß das Programm vom Beginn an neu starten. Auch die Erleichterung der Erstellung von Anwendersprachen, die die Menü-Technik benutzen, wäre für die Arbeit an Bildschirmgeräten von Vorteil. Erst die Anwendung von PLS für eine Reihe weiterer REGENT-Subsysteme wird nicht nur die Vorteile, sondern auch die Mängel dieses Systems aufzeigen und die einzuschlagende Richtung einer Weiterentwicklung weisen.

Anhang A

Die in dieser Arbeit verwendete Syntaxnotation

1. Großbuchstabige Texte und Sonderzeichen

Alle in den Syntaxbeschreibungen vorkommenden großbuchstabigen Texte und alle Sonderzeichen, die nicht Teile der Syntaxnotations-Sprache sind, sind konstante Elemente der beschriebenen Anweisung (oder eines Teiles einer Anweisung). Sie müssen so kodiert werden, wie sie in der Syntaxbeschreibung angegeben sind.

Beispiel: FILE ALLE;

Diese Anweisung muß genau so kodiert werden, wie sie hier steht.

Bezüglich der Leerzeichen gelten in PLS die PL/1-Regeln: Zwischen verschiedenen Sprachelementen (Benennungen, Operatoren und andere Begrenzungzeichen wie ,),(; und Konstanten) sind beliebig viele Leerzeichen erlaubt. Ein oder mehrere Leerzeichen müssen benutzt werden, um Benennungen oder Konstanten zu trennen, die nicht durch einen anderen Begrenzer getrennt sind. Im obigen Beispiel muß also zwischen FILE und ALLE ein oder mehrere Leerzeichen stehen, zwischen ALLE und ; darf ein oder mehrere Leerzeichen stehen. Innerhalb von Benennungen und Konstanten sind keine Leerzeichen erlaubt.

2. Unterstreichungen von großbuchstabigen Texten

Viele Schlüsselwörter in PLS können abgekürzt werden, d.h. nur die ersten i Buchstaben sind signifikant, $i \geq 1$. Danach dürfen beliebige Kombinationen von Buchstaben, Ziffern und der Zeichen: #, $, ∂, _ stehen. Es gelten also für PLS-Schlüsselwörter die Regeln wie für PL/1-Benennungen.

Beispiel: SUBSYSTEM

Dieses Schlüsselwort wird an den ersten drei Buchstaben SUB erkannt, die folgenden Zeichen sind ohne Bedeutung.
Gültig z.B.: SUB, SUBS, SUBI, SUBSYSTEM, SUBSYSTEMNAME.
Nicht gültig: SU (zu wenige Zeichen), SUB?(ungültiges Zeichen), SUB SYSTEM (enthält Leerzeichen).

3. Kleinbuchstabige Texte

Alle mit kleinen Buchstaben geschriebene Worte bezeichnen ein variables Sprachelement. Anstelle des Wortes können Mitglieder einer Gruppe von Sprachelementen stehen, z.B. Benennungen oder Konstanten. Die Bedeutung der Worte ist jeweils nach der Syntaxbeschreibung erläutert.

Beispiel: STATEMENT name;
"name" ist hier ein variables Sprachelement, an seiner Stelle können beliebige Benennungen mit maximal 32 Zeichen stehen.

Kleinbuchstabige Worte können auch als Platzhalter für eine genauere Syntaxbeschreibung stehen. Sie werden dann weiter beschrieben, indem nach dem Zeichen "::=" die für sie zulässigen Syntaxkonstruktionen aufgeführt sind.

Beispiel: FILE gruppe;
gruppe ::= (name, name)
Für "gruppe" ist also hier "(name, name)" einzusetzen.

4. Geschweifte Klammern { }

In geschweiften Klammern werden Alternativen aufgeführt, von denen eine ausgewählt werden muß.

Beispiel:

```
DESTROY { SUBSYSTEM } ;
        { STATEMENT }
```

Zulässige Alternativen sind:
DESTROY SUBSYSTEM; und
DESTROY STATEMENT;

5. Eckige Klammern [ ]

Eckige Klammern umschließen eine Gruppe von Sprachelementen die vorhanden sein können, aber nicht müssen.

Beispiel: LIST STATEMENTS [OF]
[SUBSYSTEM] name;
Die Worte OF und SUBSYSTEM können entfallen.

Eckige Klammern gefolgt von einem Stern bezeichnen eine Gruppe von Sprachelementen, die nicht, einmal oder beliebig oft stehen können.

Beispiel: ACTIVE aname [,aname]*;

Gültig ist:

ACTIVE aname;

ACTIVE aname, aname;

ACTIVE aname, aname, aname;

etc.

Steht nach dem Stern eine ganze Zahl, so bezeichnet sie die maximal erlaubte Anzahl von Wiederholungen der in eckigen Klammern stehenden Gruppe. Steht eine Alternativklammer in eckigen Klammern und ist vor eine der Alternativen ein waagrechter Pfeil gesetzt, so bezeichnet dies den Standardwert für den Fall, daß keine der Alternativen aufgeführt ist.

Beispiel:

```
 ┌ ┌→ SKIP   ┐ ┐
[{          }]
 └ └  NOSKIP ┘ ┘
```

Im Falle, daß weder SKIP noch NOSKIP steht, ist SKIP der Defaultwert.

Anhang B PLS-Syntaxbeschreibung

B1. Systemanweisungen

B1.1 Die ENTER-Anweisung

Syntax: ENTER subsystemname parameter;

Bedeutung: Das Subsystem mit dem Namen "subsystemname" wird gestartet. Die Bedeutung von "parameter" ist von Subsystem zu Subsystem unterschiedlich, hier können Subsystemparameter gesetzt werden. ENTER ist keine ausführbare Anweisung, sondern der Beginn eines Subsystemblockes. Von ENTER bis zum zugehörigen END ist das "Enviroment" des Subsystems vorhanden, d.h. alle subsystemspezifischen Daten sind zugänglich, außer Systemanweisungen sind auch die zum Subsystem gehörigen POL-Anweisungen gültig. ENTER darf nur stehen, wo auch eine BEGIN- oder PROC-Anweisung stehen dürfte.

B1.2 Die END-Subsystem-Anweisung

Syntax:

END subsystemname;

Bedeutung:

Das Subsystem mit dem Namen "subsystemname" wird abgeschlossen.

B1.3 Die POOL-Anweisung

Syntax: POOLSIZE { MODULES | DYARRAYS } poolsize;

Bedeutung:

Die Modulpoolsize, die von RMM für die dynamische Modulverwaltung benutzt wird, erhält den neuen Wert "poolsize", oder die Dynamic-Array-poolsize erhält den neuen Wert "poolsize".

B1.4 Die TRACE-Anweisung

Syntax: TRACE [{ØN / ØFF}] ;

Bedeutung:

Die TRACE-Anweisung schaltet den Lademodultrace ein oder aus. Dieser Trace wird von der REGENT-Modulverwaltung erzeugt.

B1.5 Die FINISH-Anweisung

Syntax: FINISH;

Bedeutung:

FINISH schließt die Modulverwaltung, die Dynamic-Array-Verwaltung und die Datenbankverwaltung ab und druckt abschließende Statistiken.

B1.6 Die PRINT-Anweisung

Syntax:

```
        { POOLTABLE [IDENT text ]                           }
  PRINT { DISKDUMP  [IDENT text ] [[WITH]   LEAVES ]        }
        { POOLDUMP  [IDENT text ] [[WITH]   LEAVES ]        }
          [[ON] FILE ddname];
```

Bedeutung:

Es wird Dynamic-Array-Information gedruckt. Bei POOLTABLE Verwaltungsinformation, bei DISKDUMP die Dynamic-Arrays auf der Platte, bei POOLDUMP die im Arbeitsspeicher. "text" ist eine Kennzeichnung des Ausdruckes, "WITH LEAVES" bedeutet Ausdrucken der DA-Blätter, "ddname" ist der Name eines Printfiles (Standard:SYSPRINT). Das Ausdrucken wird von der REGENT-Dynamic-Array-Verwaltung vorgenommen.

B1.7 Die REGENT - Option

Die REGENT-Option steht auf der PROCEDURE-Anweisung und kennzeichnet ein POL-Programm.

Syntax:

```
REGENT | REGENT  (option [,option]* )
option::=  { INIT    } | { SUBSYSTEM      =  subname }
           { NOINIT  }   { NOSUBSYSTEM               }

         | { MOD =  {1 | 2 | 3} } | { DARRAYS }
           { NOMOD              }   { NODA    }

         |   POOL =  poolsize     | { TRACE    } | { LIST [ = ddname ] }
                                    { NOTRACE  }   { NOLIST            }
         |  DPOOL = dpoolsize

         | { PLOT   }            | { BANK   }
           { NOPLOT }              { NOBANK }
```

Bedeutung:

INIT: Die REGENT-Datenstruktur wird initialisiert, die REGENT-Kernroutinen werden in den Modul integriert, die Modul-Verwaltung und Dynamic-Arrayverwaltung (jeweils falls benutzt) werden initialisiert. Die INIT-Option muß also für den Main-Modul angegeben werden.

NOINIT: Es werden keine Initialisierungen durchgeführt. Die REGENT-Kernroutinen werden nicht in den Modul integriert. NOINIT muß für externe POL-Routinen benutzt werden, die aus einem Main-Modul gerufen werden sollen.

SUBSYSTEM = subname:

Nur gültig für NOINIT. Es werden neben den fehlenden REGENT-Initialisierungen auch keine Subsystem-Initialisierungen vorgenommen. Das Subsystem-Environment des Subsystems "subname" wird der externen POL-Prozedur vom rufenden Modul übergeben.

NOSUBSYSTEM: Nur gültig für NOINIT. Die externe POL-Prozedur ist nicht an ein spezielles Subsystem-Environment gebunden, das vom rufenden Modul übergeben wird.

MOD = {1|2|3}: MOD gibt die Art der Strategie der Modulverwaltung im Kernspeicher an.

NOMOD: Es wird keine dynamische Modulverwaltung benutzt, die entsprechenden Kernroutinen werden nicht in den Modul integriert, die Initialisierung für die Modulverwaltung entfällt.

DARRAYS: Es wird die REGENT-Dynamic-Array-Verwaltung benutzt.

NODA: Die Dynamic-Array-Verwaltung wird nicht benötigt. Die entsprechenden REGENT-Kernroutinen fehlen im Modul, die Initialisierung für die Dynamic Arrays entfällt.

POOL=poolsize: POOL gibt den Anfangswert für den von RMM verwalteten Modulpool an. Nur von Bedeutung bei MOD.

TRACE: Die Modulverwaltung druckt auf SYSPRINT einen Trace der von ihr durchgeführten Aufträge aus. (LINKS, LOADS, DELETEs, Reorganisationen). Nur sinnvoll bei MOD=2 oder 3.

NOTRACE: Es wird kein Modul-Trace gedruckt.

LIST: PLS druckt auf SYSPRINT eine mit Statement-Nummern und Fehlermeldungen versehene Liste des POL-Programmes aus. Ist "ddname"angegeben,erfolgt die Ausgabe der Programmliste auf die Datei dieses DD-Namens.

NOLIST: Es werden nur die PROC-Anweisungen mit der REGENT-Option und die Fehlermeldungen gedruckt.

DPOOL: Gibt die Größe des Dynamic-Array-Bereiches an.

PLOT: Die Calcomp-Zeichenroutinen /75/ werden zur Ausgabe von Zeichnungen benötigt. Initialisierungs- und Endaufruf werden beim Systemstart und beim Systemabschluß vorgenommen.

NOPLOT: Es werden keine Zeichnungen erstellt.

BANK: Es wird die REGENT-Datenbankverwaltung benötigt.

NOBANK: Die REGENT-Datenbankverwaltung wird nicht benutzt.

Standardwerte:

REGENT (INIT, MOD=2, DA, POOL= 20000, NOTRACE, LIST,
DPOOL= 20000, NOPLOT, NOBANK)

Fehlt eine gültige Procedure-Anweisung als erste Anweisung eines Programms, wird die Anweisung:

REGENT: PROC OPTIONS(MAIN) REORDER;

eingefügt.

B2. PLS-Funktionen

B2.1 TYPE

RETURNS (BIN FIXED(15)), liefert den Typ des nächsten in der Eingabe stehenden Elementes zurück.

1 - Integerzahl, Binary oder Decimal Fixed-Konstante ohne Vorzeichen

2 - Realzahl, Binary oder Decimal-Float-Konstante ohne Vorzeichen

3 - Zeichenkette

4 - Bitkette

5 - Benennung (Identifier)

6 - Imaginärkonstante

7 - Operator: +, -, *, /, &, | , ¬, ||, =, ¬=, <, <=, ¬<, > , >=, ¬>, **

8 - Klammer auf: (

9 - Klammer zu:)

10 - Semikolon: ;

11 - Wie 1, jedoch mit Vorzeichen + oder -

12 - Wie 2, jedoch mit Vorzeichen + oder -

13 - Anderes PL/1-Zeichen: ,, %,

14 - Nicht - PL/1 - Zeichen, z.B. !

15 - End of File, Ende der Eingabe (EOF)

B2.2 NEXT_ITEM (typ [, { →SKIP | NOSKIP }])

RETURNS(CHAR(250) VARYING), liefert das nächste Element, das in der Eingabe vorhanden ist, der Typ des Elementes wird in der BIN FIXED(15)-Variablen "typ" zurückgeliefert, die Werte haben die gleiche Bedeutung wie bei der PLS-Funktion "TYPE".

SKIP bedeutet, daß das Element in der Eingabe übergangen wird. Bei NOSKIP wird der Eingabezeiger nicht verändert. Der Defaultwert ist SKIP.

B2.3 NEXT_REAL
NEXT_STRING
NEXT_BITSTRING
NEXT_OPERATOR
NEXT_IDENTIFIER
NEXT_WORD

[({ →SKIP | NOSKIP })]

RETURNS (CHAR(250)Varying), liefert ein Element aus der Eingabe,
NEXT_REAL liefert eine Decimal Float oder Binary Float-Konstante mit oder ohne Vorzeichen,
NEXT_INTEGER liefert eine Decimal oder Binary Fixed - Konstante mit oder ohne Vorzeichen,
NEXT_STRING liefert eine Zeichenketten-Konstante einschließlich der Apostrophe,
NEXT_BITSTRING liefert eine Bitketten-Konstante einschließlich der Apostrophe und des "B",
NEXT_OPERATOR liefert einen PL/1-Operator,
NEXT_IDENTIFIER liefert eine Benennung,
NEXT_WORD liefert alle Zeichen bis zum nächsten Leerzeichen in der Eingabe oder bis zum nächsten Semikolon.

Falls das gewünschte Element nicht vorhanden ist, d.h. wenn der Typ des nächsten Elementes in der Eingabe nicht gleich dem angeforderten Typ ist, erfolgt eine Fehlermeldung. Es wird dann Null oder der Nullstring zurückgegeben.

SKIP: Das gelieferte Element in der Eingabe wird übergangen.
NOSKIP: Der Eingabezeiger bleibt auf dem gelieferten Element stehen.

Ist bei NEXT_WORD das nächste Zeichen in der Eingabe ein Semikolon, so wird ein Semikolon zurückgeliefert, gleichzeitig erfolgt eine Fehlermeldung.

B2.4 NEXT_N(n [, {←SKIP | NOSKIP}])

RETURNS(CHAR(250)VARYING), liefert die nächsten n Zeichen die in der Eingabe stehen. "n" ist BIN FIXED(15) und muß ≤ 250 sein. Sind bis zum Ende der Eingabedatei weniger als n Zeichen vorhanden, erfolgt eine Fehlermeldung, es werden dann nur die vorhandenen Zeichen geliefert.
SKIP-NOSKIP haben die Bedeutung wie bei den anderen PLS-Funktionen.

B2.5 NEXT_EXPRESSION [({←SKIP | NOSKIP})]

RETURNS(CHAR(250)VARYING, liefert den nächsten arithmetischen, logischen oder Zeichenketten-Ausdruck, der in der Eingabe steht.

B2.6 THIS_STATEMENT

RETURNS(CHAR(3500)VARYING, liefert die aktuelle Anweisung vom Anfang bis zum Semikolon, ohne Labels und Prefixes.

B2.7 WORD(xyz [, {←SKIP | NOSKIP}])

BWORD(xyz [, {←SKIP | NOSKIP}])

RETURNS (BIT(1)), stellt fest, ob die Zeichen "xyz" gleich dem nächsten Wort in der Eingabe sind (bei WORD) oder ob "xyz" der Anfang des nächsten Wortes in der Eingabe ist (bei BWORD, B steht für Begin). Ein Wort ist in diesem Sinne eine Folge von Zeichen außer Leerzeichen und Semikolon, abgeschlossen durch Leerzeichen oder Semikolon. Wird das Wort (oder der Wortanfang)gefunden, wird '1'B zurückgegeben, sonst '0'B.

B2.8 IDENTIFIER(xyz [,{→SKIP | NOSKIP}])

BIDENTIFIER(xyz[,{→SKIP | NOSKIP}])

RETURNS(BIT(1)), stellt fest, ob die Zeichen "xyz" die nächste Benennung (Identifier) bzw. der Anfang der nächsten Benennung in der Eingabe sind. Eine Benennung ist eine Folge von Zeichen, bestehend aus Buchstaben, Ziffern und den Zeichen $, #, ∂ , -
Die Zeichenfolge muß mit einem Buchstaben oder #, $, ∂ beginnen. Benennungen oder Identifier sind also gültige PL/1-Variablennamen. Wird die Benennung gefunden, wird '1'B zurückgeliefert und falls "SKIP" angegeben wurde, übergangen. Ist die Benennung (oder deren Anfang) nicht in der Eingabe vorhanden, wird '0'B zurückgegeben, der Eingabezeiger wird nicht verändert.

B2.9 ISEXPRESSION

RETURNS(BIT(1)), liefert '1'B falls als nächstes in der Eingabe ein arithmetischer, logischer oder Zeichenketten-Ausdruck steht. Sonst ist das Ergebnis '0'B.

B2.10 FIND(xyz)

RETURNS(BIT(1)), liefert '1'B, falls in der behandelten POL-Anweisung bis zum Semikolon die Zeichen "xyz" der Beginn eines Elementes sind. Die Zeichenfolge xyz wird also nur am Beginn einer Benennung einer Konstanten, eines Operators oder eines Begrenzers wie , ,), (,; gesucht.
Wird die Zeichenfolge "xyz" in der Anweisung nicht gefunden, wird '0'B zurückgeliefert.

B3. PLS-Anweisungen

B3.1 INITIATE - initialisiere ein Subsystem

Die INITIATE-Anweisung dient dazu, ein neues Subsystem zu initialisieren.Es wird der Name des Subsystems und ein Schlüsselwort angegeben. Dieses Schlüsselwort muß stets angegeben werden, wenn das Subsystem erweitert, geändert oder gelöscht werden soll.

Syntax:

INITIATE [SUBSYSTEM] name [[KEY] key];

Bedeutung:

"name" ist eine Zeichenkette mit maximal 32 Zeichen, die den Namen des Subsystems darstellt. "name" darf kein Leerzeichen enthalten. Soll der Name des Subsystems abkürzbar sein, ist nach den signifikanten ersten Buchstaben ein "." in den Namen einzufügen. Soll ein Punkt Bestandteil des Namens sein, müssen 2 Punkte geschrieben werden. Beispiel: 'NAM.E', das Subsystem heißt NAME, es kann zu NAM abgekürzt werden. 'S..1.22', das Subsystem heißt S.122 und kann S.1 abgekürzt werden.

"key" ist eine Zeichenkette von maximal 32 Zeichen und bezeichnet den Schlüssel für das Subsystem. "key" darf keine Leerzeichen enthalten. Defaultwert für "key", falls nicht angegeben, ist der Nullstring.''

Ist ein Subsystem mit gleichem Namen und gleichem Schlüssel schon vorhanden,so wird zuerst das alte Subsystem zerstört und anschließend neu initialisiert. Stimmt dagegen nur der Name und nicht der Schlüssel, erfolgt eine Fehlermeldung und keine Initialisierung.

B3.2 SUBSYSTEM - Identifiziere ein Subsystem

Die Subsystem-Anweisung dient dazu, anzugeben, zu welchem REGENT-Subsystem die folgenden Definitionen gehören sollen. Alle POL-und Datenstruktur-Definitionen bis zur nächsten SUBSYSTEM-Anweisung beziehen sich auf das angegebene Subsystem.

Syntax:

```
SUBSYSTEM   name   [ [ KEY ] key ];
```

Bedeutung:

"name" Subsystemname, ausgeschrieben oder falls abkürzbar, auch abgekürzt.

"key" Schlüssel für das Subsystem.
(Nullstring falls nicht angegeben).

B3.3 DESTROY - Lösche ein Subsystem, eine Datenstruktur oder eine Anweisung

Mit der DESTROY-Anweisung werden Teile eines Subsystems oder ein ganzes Subsystem gelöscht.

Syntax:

```
          ⎧ DATASTRUCTURE dname [dname]*                                     ⎫
          ⎪ CLAUSE   cname  [cname]*                                         ⎪
          ⎪ [SUBSYSTEM] subname  [[KEY] subkey]                              ⎪
DESTROY   ⎨ MACROTIME [DATASTRUCTURE]                                        ⎬ ;
          ⎪ INITIAL [CLAUSE]                                                 ⎪
          ⎪                                                                  ⎪
          ⎪ STATEMENT  ⎧ [NAME] stname  ⎫ [⎧ [NAME] stname  ⎫]*              ⎪
          ⎩            ⎩ DATATYPE stype ⎭  ⎩ DATATYPE stype ⎭                ⎭
```

Bedeutung:

DESTROY DATASTRUCTURE löscht die aufgeführten Subsysteme-Datenstrukturen mit dem Namen "dname".

DESTROY CLAUSE zerstört die aufgeführten Clauses mit dem Namen"cname".

DESTROY SUBSYSTEM löscht ein Subsystem mit dem Namen "subname", falls der richtige Schlüssel "subkey" angegeben wird.

DESTROY MACROTIME DATASTRUCTURE löscht die Makrozeit-Datenstruktur des gerade behandelten Subsystems.

DESTROY INITIAL CLAUSE löscht die Initialisierungsroutine des gerade behandelten Subsystems.

DESTROY STATEMENT löscht die angeführten POL-Anweisungen. "stname" ist das Schlüsselwort der POL-Anweisungen, die mit einem festen Schlüsselwort beginnen. "stype" ist der Typ der Datentyp-Anweisungen, die mit einem bestimmten Datentyp beginnen "stype" kann ASSIGNMENT, PASSIGNMENT, REAL, INTEGER IDENTIFIER, STRING, BITSTRING oder OPERATOR sein.

B3.4 LIST - Erzeuge eine Liste der Subsysteme, POL-Anweisungen und Subsystem-Datenstrukturen.

Mit der LIST-Anweisung lassen sich die REGENT-Subsysteme und die gültigen Anweisungen eines Subsystems listen.

Syntax:

```
       { SUBSYSTEMS                                  }
LIST   {                                             } ;
       { STATEMENTS [OF][SUBSYSTEM]  subname         }
```

Bedeutung:

Entweder werden alle REGENT-Subsysteme oder die gültigen PL/1- Anweisungen, die POL-Anweisungen und die Datenstrukturen eines Subsystems gelistet (Siehe Abb. 24 und 25, Seiten 116 bis 118).

Bedeutung der Spalten der Subsysteme-Liste:

NAME OF SUBSYSTEM - Subsystemname in voller Länge; wenn der Name abkürzbar ist, sind die signifikanten Zeichen unterstrichen.

PREFIX - Subsystem-Prefix. Dieser aus zwei Zeichen bestehende Prefix wird benutzt, um die Treibermodule, die Datenstrukturen und die von PLS verwalteten Listen, die zu einem Subsystem gehören, eindeutig zu kennzeichnen. Die Namen aller zu einem Subsystem gehörenden Member in den PLS-Bibliotheken fangen mit dem Prefix an. Auch die Namen der PLR-Module beginnen mit dem Subsystem-Prefix.

CREATION DATE - Datum der Initialisierung des Subsystems.

SUBSYSTEM LIST - Name der Tabelle der POL-Anweisungen und Datenstrukturen des Subsystems in der Bibliothek REGENT.PLSTRAN.DATA

ABBREV.? - YES oder NO, je nachdem ob der Subsystemname abkürzbar ist. Bei YES sind die signifikanten Buchstaben des Namens unterstrichen.

NUMBER OF STATEMENTS - Anzahl der POL und PL/1 Anweisungen, die zum Subsystem gehören, die Anzahl ist $\geq$ 61.

NUMBER OF CLAUSES - Anzahl der Clauses des Subsystems.

NUMBER OF DATA-DECLARATIONS - Anzahl der Datenstruktur-Definitionen.

Bedeutung der Spalten der Anweisungsliste:

Der Kopf der Liste besteht aus den Angaben, die das betreffende Subsystem beschreiben.

NAME OF STATEMENT - Name der POL- oder PL/1-Anweisung oder der Datendefinition. Die Reihenfolge der Liste ist:

- Preprocessorzuweisung und PL/1-Zuweisung (%AS und ASS)
- Datentyp - POL-Anweisungen, falls definiert (REAL, INT,etc)
- PL/1-Preprocessor-Anweisungen
- PL/1- und POL-Anweisungen

Sind die Namen abkürzbar, so sind die signifikaten Buchstaben unterstrichen.

CHARACTERISTIC - beschreibt die Art der Anweisungen

MODUL - Name des zugehörigen Members in einer PLS-Bibliothek. Bei POL-Anweisungen und Systemanweisungen ist dies der Name des Treiberroutinen-Moduls. Bei Datendefinitionen ist es der Membername in der PLS-Databibliothek.

B3.5 DATASTRUCTURE - Definiere eine Subsystem-Datenstruktur

Die DATASTRUCTURE-Anweisung dient dazu, Subsystem-Datenstrukturen zu definieren. Die PL/1-Deklaration der gewünschten Datenstrukturen werden angegeben. Diese

Syntax:

```
DATASTRUCTURE  { COMMON } ;
               { dname  }
      :
      :
END DATASTRUCTURE;
```

Bedeutung:

DATASTRUCTURE COMMON; Beginn der Subsystem-Common-Definition. Es muß anschließend eine einzige Level-1 "structure"-Deklaration folgen.

DATASTRUCTURE dname: Es wird eine Subsystem-Datendeklaration definiert mit dem Namen "dname"(maximal 32 Zeichen). Es folgen anschließend eine Reihe von PL/1-Deklarationen und ausführbare Anweisungen, die beim Subsystemstart ausgeführt werden sollen.

B3.6 MACROTIME DATASTRUCTURE - Deklariere eine PLS-Makrozeit-Datenstruktur

Syntax:

```
MACROTIME [DATASTRUCTURE]  [LENGTH l];

DCL  1  name,
        2 ..........
        .
        .
        .
END MACROTIME [DATASTRUCTURE];
```

Bedeutung:

LENGTH l : Die Länge, die die Datenstruktur im Arbeitsspeicher beansprucht, in Bytes."l" muß größer oder gleich der tatsächlichen Strukturlänge sein. Wenn die Angabe fehlt, wird der Defaultwert l=1024 genommen. (Es ist vorgesehen, l aus der Strukturdeklaration zu errechnen). Es kann nur eine einzige solche PL/1 - Structure-Deklaration angegeben werden, Dimensionen und Längen müssen fest sein.

B3.7 STATEMENT

Syntax:

```
STATEMENT { [NAME] name    } [ALIAS  aname [aname]* ]
          { DATATYPE type  }

     [LABELS [TO] labelvar ] [PREFIXES [TO] prefixvar ];
     .
     .
     .
     Definition der Anweisungs-Abarbeitung
     .
     .
     .
END  STATEMENT;
```

Bedeutung:

NAME-DATATYPE: Im Normalfall beginnt eine POL-Anweisung mit einem Schlüsselwort, so wie auch z.B. alle PL/1-Anweisungen außer der Zuweisung und der Nullanweisung mit einem Schlüsselwort beginnen. Dieses Schlüsselwort, der Name der Anweisung, wird nach "NAME" angegeben. "name" ist eine Zeichenkette mit maximal 32 Zeichen, eingeschlossen in Apostrophe. Soll der Name der Anweisung abkürzbar sein, so muß nach den signifikanten Zeichen ein Punkt stehen. Soll der Name der Anweisung einen Punkt enthalten, müssen zwei Punkte angegeben werden.

Es ist auch möglich, Anweisungen zu definieren, die nicht mit einem Schlüssel, wort, sondern mit einem bestimmten Datentyp beginnen. Die möglichen Datentypen sind:

Bitstring	-	"type"	=	BITSTRING	Bitkettenkonstante
Identifier	-	"type"	=	IDENTIFIER,	dies sind Benennungen (Namen)
Integerzahl	-	"type"	=	INTEGER,	alle BINARY oder DECIMAL FIXED-Konstanten
Operator	-	"type"	=	OPERATOR,	alle gültigen PL/1-Operatoren +, -, /, *, ||, |, &, >, usw.
Realzahl	-	"type"	=	REAL,	alle BINARY oder DECIMAL FLOAT-Konstanten
Characterstring	-	"type"	=	STRING,	alle Zeichenketten-Konstanten

Alias : Eine Anweisung kann bis zu 21 verschiedene Namen haben, also außer dem Hauptnamen auch bis zu 20 Aliasnamen. Für "aname" gelten die gleichen Regeln wie für den Namen der Anweisung (max.32 Zeichen lange Zeichenkette in Apostrophen, Abkürzung wird durch Punkt bezeichnet).

LABELS: PREFIXES: Jede PL/1-Anweisung und jede POL-Anweisung kann mit Labels und Prefixes versehen sein. Im Normalfall, wenn die Angaben von "LABELS" und "PREFIXES" in der Anweisungs-Definition fehlt, werden die Prefixes und Labels vor die expandierte(n) Anweisung(en) gesetzt. Sollten jedoch die Treiberroutine, die die POL-Anweisung abarbeitet, die Labels und/oder Prefixes zur Verfügung gestellt werden, können mit "labelvar" und "prefixvar" zwei Variablennamen angegeben werden. In diese Variable speichert der PLS-Übersetzer dann die Labels bzw. Prefixes hintereinander, einschließlich aller Klammern und Doppelpunkte.

B3.8 CLAUSE

Syntax:

```
CLAUSE  { cname   } ;
        { INITIAL }
   :
Definition der Abarbeitung
   :
END CLAUSE;
```

Bedeutung:

"cname" ist der Name der CLAUSE. Er dient zur Identifikation, z.B. beim Löschen einer CLAUSE und zum Aufruf. "cname" ist eine Zeichenkette, eingeschlossen in Apostrophe, maximal 32 Zeichen lang.

INITIAL definiert eine Initialisierungs-CLAUSE, die nur beim Subsystemstart aktiv wird. Sie kann zum Initialisieren der Subsystem-Datenstrukturen, auch zum Initialisieren der Übersetzungszeit-Datenstruktur und zum Abarbeiten des Restes der "ENTER subsystemname"-Anweisung dienen.

B3.9 FILE - speichere STATEMENT und CLAUSE-Definitionen permanent

Die PLS-Anweisungen STATEMENT und CLAUSE erzeugen Treiberroutinen auf temporären Dateien (es sind dies Members auf einer Objektmodul-Bibliothek). Die FILE-Anweisung dient zum permanenten Speichern der Treibermodule (als Lademodule). Durch FILE werden also die Treiberroutinen zu dynamisch aufrufbaren Moduln zusammengebunden. Die einfachste und normalerweise angewandte Form der Anweisung ist : FILE;. Es werden alle bis dahin seit der letzten FILE-Anweisung definierten STATEMENT- und CLAUSE-Routinen jede für sich zu einem Modul gebunden und in der Datei REGENT. PLSTRAN.MODS angelegt. Die FILE-Anweisung ermöglicht es aber auch, verschiedene Statement- und Clause-Treiberroutinen und schon fertig gebundene Module zu einem einzigen Modul zusammenzubinden. Dies ist dann effektiver, wenn die Routinen meist zusammen benutzt werden, da dann nur einmal der Modul von der Platte geladen werden muß.

<u>Syntax:</u>

```
              ⎧ [ALL]                    ⎫
<u>FILE</u>    [  ⎨                           ⎬ ] [[ON] LIBRARY ddname ];
              ⎩ (gruppe [,gruppe]*)      ⎭

gruppe::= ⎧ name              ⎫
          ⎩ (name [,name]*19  ⎭ )

          ⎧ DATATYPE  dtype   ⎫
name:: =  ⎨ [NAME] rname      ⎬
          ⎩ MODUL modulname   ⎭
```

<u>Bedeutung:</u>

ALL: Jede bisher definierte Statement- oder Clause-Routine wird in je einem Modul permanent gespeichert. ALL ist default.

gruppe: Besteht eine "gruppe" aus einem Namen, wird die Definition dieses Namens zu einem Treibermodul gebunden, besteht die "gruppe" aus einer eingeklammerten Liste von Namen, werden die in der Liste aufgeführten Definitionen zusammen in einen Modul gebunden.

name: Name einer Statement- oder Clause-Definition (Schlüsselwort: NAME, default), einer Datentyp-Anweisungs-Definition (Schlüsselwort DATATYPE) oder eines Moduls, der sich bereits in der Datei REGENT.PLSTRAN.MODS befindet (Schlüsselwort MODUL). Namen jeweils in Apostrophe eingeschlossen.

- type
- rname
- modulname

ddname: DD-Name der Bibliothek, auf die die erzeugten Treibermodule gespeichert werden sollen. Normalerweise ist die Datei REGENT.PLSTRAN.MODS über den DD-Namen 'PLSLIB' ansprechbar, dies ist auch der Defaultwert für "ddname". Der Name muß in Apostrophe eingeschlossen sein.

B3.10 COMPRESS, NOCOMPRESS - komprimiere die PLS-Bibliotheken(nicht)

Normalerweise werden die Bibliotheken REGENT.PLSTRAN.DATA und REGENT.PLSTRAN. MODS immer dann beim Abschluß des PLS-Subsystems komprimiert, wenn sie während der betreffenden PLS-Anwendung verändert wurden. Wenn nicht viel verändert wurde und es ist genug Platz auf den externen Speichern, ist das Komprimieren nach jedem Lauf jedoch nicht sehr effektiv. Durch die PLS-Anweisung NOCOMPRESS wird das Komprimieren verhindert.

Syntax: NOCOMPRESS;

Soll dagegen komprimiert werden, obwohl in dem betreffenden PLS-Lauf nichts auf den Bibliotheken verändert wurde (kein INITIATE,DESTROY, FILE, DATASTRUCTURE), kann das durch Angabe der Anweisung COMPRESS vor Subsystemabschluß geschehen.

Syntax: COMPRESS;

B3.11 CPARMS, LPARMS - verändere die Parameter für PL/1-Compiler und Linkage-Editor

Die STATEMENT-und CLAUSE-Definitionsprogramme benutzen zur Erzeugung eines Objektmoduls einer Treiberroutine den PL/1-Optimising Compiler des OS/360. Die FILE-Routine benutzt zur Erzeugung von Treibermoduln den OS/360-Linkage Editor. Mit den Anweisungen CPARMS und LPARMS können die Parameter für diese Programme geändert werden.
Default-Parameter für den Compiler:

'INCLUDE, NA, NAG, NOP, NSTG, SMSG, NX',

für den Linkage-Editor:

'REUS, SIZE=(64K, 18K), DCBS'.

Syntax:

CPARMS char-ex;

LPARMS char-ex;

char-ex: Zeichenkettenausdruck.

B4. Makrozeitanweisungen

B4.1 SKIP-Anweisungen

SKIP;
SKIPB;
Diese Anweisungen dienen zum Übergehen eines Elementes in der Eingabe, sie setzen den Eingabezeiger um ein Element nach rechts. Ob der Eingabezeiger verändert wird, kann davon abhängig gemacht werden, ob ein bestimmtes Element oder eine Elementart als nächstes in der Eingabe steht.

SKIP REAL; Übergehe das nächste Element, falls es eine Realkonstante ist.

SKIP INTEGER; Übergehe das nächste Element, falls es eine Integerkonstante ist.

SKIP OPERATOR; Übergehe das nächste Element, falls es ein Operator ist.

SKIP STRING; Übergehe das nächste Element, falls es eine Zeichenkette ist.

SKIP BITSTRING; Übergehe das nächste Element, falls es eine Bitkette ist.

SKIP EXPRESSION; Übergehe einen arithmetischen, logischen oder Zeichenkettenausdruck, falls ein solcher als nächstes in der Eingabe vorhanden ist.

SKIP WORD; Übergehe das nächste Wort in der Eingabe (alle Zeichen bis zum nächsten Semikolon oder Leerzeichen).

SKIP n ; "n" ist eine Integerkonstante. Übergehe die nächsten n Zeichen.

SKIP IDENTIFIER; Übergehe das nächste Element in der Eingabe, falls es eine Benennung ist.

SKIP IDENTIFIER(xyz); Übergehe das nächste Element in der Eingabe, falls es genau die Benennung "xyz" ist.

SKIPB IDENTIFIER(xyz); Übergehe das nächste Element, falls es eine Benennung ist, die mit "xyz" beginnt.

SKIP(xyz); Übergehe das nächste Wort in der Eingabe, falls es genau aus den Zeichen "xyz" besteht.

SKIPB(xyz); Übergehe das nächste Wort, falls es mit den Zeichen"xyz" beginnt.

SKIP; Übergehe das nächste Element in der Eingabe.

B4.2PLI; Diese Anweisung teilt dem PLS-Übersetzer mit, daß er die Anweisung als System- oder PL/1-Anweisung behandeln soll. Man kann PL/1-Anweisungen auf bestimmte Eigenschaften und Optionen untersuchen, indem man eine POL-Anweisung gleichen Namens definiert. Will man dann die Anweisung unverändert lassen, benutzt man dazu die PLI-Anweisung.

B4.3 LINK cname; Die LINK-Anweisung dient zum Aufrufen von CLAUSES. "cname" ist der in der CLAUSE-Anweisung definierte Name der Clause.

B4.4 EXECUTE; END EXECUTE; Zwischen "EXEC;" und END EXEC;" steht der Ersetzungstext, also diejenigen PL/1-Anweisungen, die an Stelle eines Teiles einer POL-Anweisung erzeugt werden sollen. Der Ersetzungstext kann auch zwischen "EXEC" und ";" stehen. Innerhalb des Ersetzungstextes werden Namen aktiver Ersetzungsvariablen durch ihren Wert ersetzt.

B4.5 EXECUTE LINK modulname [(argumente)] ;

Innerhalb des Ersetzungstextes ist außer PL/1- Anweisungen auch eine besondere Anweisung erlaubt, die LINK-Anweisung. Sie dient zum dynamischen Aufruf von Subsystemmoduln bei der Ausführung des aus dem POL-Programm erzeugten PL/1- Programmes. An den gerufenen Moduln können Argumente übergeben werden.

B4.6 ACTIVE anname [, aname]*;
UNACTIVE uname [, uname]*;

Durch die ACTIVE-Anweisung kann eine Makrozeitvariable innerhalb des PL/1-Blockes, in dem die ACTIVE - Anweisung steht, aktiviert werden, d.h. in Ersetzungstexten wird ihr Name durch ihren Wert ersetzt. UNACTIVE desaktiviert eine Makrozeitvariable. PLS-Funktionen können im äußeren Block einer STATEMENT- oder CLAUSE - Definition durch eine UNACTIVE - Anweisung desaktiviert werden.

Anhang C: Zur Implementierung

C1 Implementierung des PLS-Übersetzers

Symbolentschlüsselung

Die Symbolentschlüsselung oder lexikalische Analyse zergliedert den Eingabetext, der in Form einer sequentiell einzulesenden Zeichenkette vorliegt, in einzelne Grundelemente (Symbole). Die Grundelemente der problemorientierten Sprachen in REGENT sind die Symbole der PL/1-Sprache. Der Symbolentschlüsseler stellt einen deterministischen endlichen Automaten dar. Ein solcher Automat A läßt sich durch folgendes Quintupel beschreiben /76/:

$$A := (T, Q, \delta, S, Z) \qquad Z \supset Q, \ Z \neq \emptyset, \ S \ni Q \tag{4}$$

T ist die Menge der Eingabesignale, Q die Menge der Zustände und δ eine Abbildung, die die Produktmenge T*Q in die Zustandsmenge Q abbildet:

$$\delta := T * Q \rightarrow Q \tag{5}$$

Der innere Zustand des Automaten zum Zeitpunkt i+1 hängt also vom Zustand und der Eingabe zum Zeitpunkt i ab.S ist der Startzustand und Z die Menge der Endzustände des Automaten.

Bei der Symbolentschlüsselung ist die Menge der Eingabesignale T die Menge aller erlaubten Zeichen eines Programms. Da in PL/1-Zeichenketten für jedes Zeichen alle darstellbaren Bitkombinationen erlaubt sind, besteht das Eingabealphabet auf den DVA des Typs IBM/360-/370 aus den 256 verschiedenen Zeichen des EBCDIC-Codes /77 /. Die Übergangsfunktion δ wird durch das Programm des Symbolentschlüsselers realisiert. Zu jedem Zustand wird der Folgezustand durch das nächste gelesene Zeichen bestimmt. Wenn ein Sprachsymbol vollständig erkannt ist, ist ein Endzustand des Automaten erreicht. Abhängig vom Endzustand erfolgt eine Ausgabe. Sie besteht aus einer Zahl, die die Art des Symbols angibt und aus dem Symbol selbst (Tabelle der Symbolarten siehe Anhang D). Das Symbol ist die Verkettung der zwischen Start- und Endzustand gelesenen Zeichen. Das zuletzt gelesene Zeichen kann aber muß nicht Teil des Symbols sein. Die Zeichenfolge A+B besteht z.B. aus dem Symbol "A"(Art:"Benennung"), dem Symbol "+"(Art:"+-Operator") und dem Symbol "B"(Art:"Benennung"). Bei den Benennungen kann erst nach dem Lesen des nächsten Zeichens festgestellt werden, daß das Symbolende erreicht ist,

Dagegen kann der +-Operator immer nur aus einem Zeichen bestehen. Das Lesen des nächsten Zeichens ist daher nicht nötig zum Erkennen des Symbols.

Die Zustände und die Übergänge eines Automaten können in einem Automatendiagramm dargestellt werden, die Knoten des Graphen stellen die Zustände dar, Pfeile zwischen den Knoten die Zustandsübergänge. An den Pfeilen sind diejenigen Eingabezeichen eingetragen, die den jeweiligen Übergang bewirken. Anhang E zeigt das Automatendiagramm des PLS-Symbolentschlüsslers. Dabei ist die Entschlüsselung für numerische Konstanten ein Unterautomat, der immer dann aufgerufen wird, wenn das erste gelesene Zeichen bei der Symbolentschlüsselung eine Ziffer oder ein Dezimalpunkt ist.

Der Symbolentschlüsseler ist als Prozedur realisiert (P1SCAN), der als Ergebnis seines Aufrufes in drei globalen Variablen des Übersetzer-Monitors den Typ des Symbols (NTYP, siehe Anhang D), seine Startposition im Eingabepuffer (NB) und die Symbollänge (NL) zurückliefert.

Nachdem ein Symbol erkannt wurde, wird vor dem Rücksprung aus der Entschlüsselungsroutine der Zeiger im Eingabepuffer im Normalfall auf das erste Zeichen gesetzt, das nach dem erkannten Symbol folgt. Jedoch kann der Symbolentschlüsseler auch veranlaßt werden, den Zeiger auf den Beginn des gerade erkannten Symbols zurückzusetzen. Auf diese Weise kann ein Symbol betrachtet werden, ohne daß es im Puffer übergangen wird. In manchen Fällen ist während der Syntaxanalyse ein solcher Vorgriff auf das nächste Symbol (Lookahead" um ein Symbol) notwendig, z.B. um zu entscheiden, ob ein arithmetischer Ausdruck zu Ende ist oder ob ein weiterer Operator folgt. Damit nicht bei einem erneuten Aufruf der Entschlüsselungsroutine das Symbol noch einmal erkannt werden muß, wird es in diesem Fall solange zwischengespeichert, bis der Eingabezeiger hinter das Symbolende gesetzt wird. Mit Hilfe der COUNT-Option des PL/1-Optimising Compilers /48 /, die es erlaubt, die Häufigkeit der Ausführung jeder Programmanweisung während eines Programmlaufs zu zählen, wurde festgestellt, daß in etwa 2/3 der Aufrufe der Symbolentschlüsselung die Symbole neu erkannt werden müssen und daß in 1/3 der Fälle die bereits erkannten Symbole aus den Zwischenvariablen entnommen werden konnten. Über Steuerparameter kann die Entschlüsselungsroutine weiterhin veranlaßt werden, Leerzeichen (Blanks) und Kommentare entweder an den ein Symbol anfordernden Ort des Aufrufs zurückzugeben oder aber Leerzeichen und Kommentare zu übergehen. Dabei werden Kommentare sofort in den erzeugten PL/1-Text kopiert.

Der die Symbolentschlüsselung bewirkende endliche Automat ist durch direkte Programmierung realisiert, jedem Zustand entspricht ein Programmstück, das durch Fallunterscheidung über das Folgezeichen zum nächsten Zustand springt. Für den Unterautomaten für die numerischen Konstanten ist die Übergangsfunktion δ dargestellt durch eine Zustandsmatrix $\underline{M}$. Sie bestimmt in Abhängigkeit vom alten Zustand q_i und dem nächsten Zeichen z den Folgezustand q_{i+1}:

$$q_{i+1} = \underline{M}\,(q_i, z) \qquad (6)$$

Die q und z sind dabei als ganze Zahlen dargestellt, die direkt zur Indizierung von $\underline{M}$ verwendet werden können. Ausgezeichnete Zustände sind die Endzustände, im Fehlerfall wird ein besonderer Endzustand erreicht. Das Programm wird bei Verwendung einer Zustandsmatrix sehr kurz:

```
Q = Startzustand;
SYMBOL = '';                Leere Zeichenkette
DO WHILE Q ¬= Endzustand;
   Z = nächstes Zeichen;
   SYMBOL = SYMBOL || Z;  Hänge Z an Symbol an
   Q = M(Q,Z);
   END;
```

Zum Lesen des jeweils nächsten Zeichens wird eine Prozedur GETCHAR aufgerufen, die in einem Eingabepuffer für die zu lesenden Zeichen einen Zeiger auf das nächste Zeichen setzt. Für die Zergliederung ist es sinnvoll, Zeichen, die im Automaten gleiche Zustandsübergänge bewirken, zu Zeichenklassen class(z) zusammenzufassen. Die Buchstaben A bis Z oder die Ziffern 0 bis 9 gehören z.B. zu solchen Zeichenklassen. In einem Zeichenklassenvektor ist zu jedem der 256 EBCDIC-Zeichen die Zeichenklasse festgelegt, so daß die Routine GETCHAR durch Indizieren dieses Vektors mit der internen Representation des Zeichens die zugehörige Zeichenklasse feststellen und an die Entschlüsselungsroutine zurückliefern kann (Anhang D). Die Zustandsmatrix muß nun nicht mehr für jedes Zeichen, sondern nur noch für jede Zeichenklasse eine Spalte besitzen und wird dadurch in ihrer Größe reduziert. Gl.(6) geht über in:

$$q_{i+1} = \underline{M}\,(q_i, \mathrm{class}(z)) \qquad (7)$$

Wenn der Zeiger im Eingabepuffer das Pufferende erreicht hat, wird eine Einleseroutine zum Füllen des Puffers aufgerufen. Da bei der Übersetzung von

Subsystem-Sprachanweisungen auf die gerade behandelte Anweisung in manchen Fällen noch einmal zugegriffen werden muß, werden die Zeichen vom Anweisungsbeginn bis zum Pufferende an den Pufferanfang geschoben und anschließend der Rest des Puffers neu eingelesen. Bei Dateiende wird eine besondere Zeichenklasse "Dateiende" von GETCHAR zurückgegeben, die dazu führt, daß auch der Symbolentschlüsseler das Symbol "Dateiende" erkennt und zurückgibt.

Bei mehreren verschiedenen Anwendungen des PLS-Übersetzers wurde mit Hilfe der COUNT-Option des Optimising Compilers festgestellt, welches die am häufigsten ausgeführten und deshalb für die Effektivität am wichtigsten Anweisungen sind. Erwartungsgemäß war die Routine GETCHAR das bei weitem am häufigsten ausgeführte Programmstück des Übersetzers. An zweiter Stelle lagen die Anweisungen, die den Übergang vom Startzustand des Symbolentschlüsselers in einen Folgezustand bewirken. Die Routine GETCHAR wurde deshalb noch einmal in Assembler kodiert, um ihre Schnelligkeit zu steigern.

Die Symbolentschlüsselungsprozedur P1SCAN wird immer dann aufgerufen, wenn während der Syntaxanalyse (Zerteilung) ein neues Symbol benötigt wird; im Monitor des Übersetzers insbesondere zum Erkennen von Anweisungspräfixen, von Zuweisungen und Schlüsselwortanweisungen und von arithmetischen oder logischen Ausdrücken. Die Anweisungstreiberroutinen rufen den Entschlüsseler nicht direkt, sondern über ebenfalls zum Monitor gehörenden Interfaceroutinen auf.

Erkennen von Präfixen und Zuweisung

Der Monitor muß die Bedingungs-Präfixe und die Marken vor jeder Anweisung eines Programms erkennen können. Zuweisungen müssen von Schlüsselwortanweisungen unterschieden werden. Das Erkennen von Anweisungs-Präfixen und Zuweisungen kann ebenso wie die Symbolentschlüsselung durch deterministische endliche Automaten erfolgen. Die Automaten sind durch Zustandsmatrizen realisiert, da der Speicherplatzbedarf bei dieser Methode geringer war als bei direkter Programmierung der Zustandsübergänge.

Zerteilung von Ausdrücken

Die Zerteilung (Syntaxanalyse) erkennt die Syntax von (Teil-)Anweisungen. Die Anweisungstreiberoutinen können vom Übersetzermonitor das Erkennen von arithmetischen, logischen oder Zeichenketten-Ausdrücken fordern. Die Ausdruck-Zerteilung stellt fest, ob in der Eingabe ein Ausdruck vorhanden ist und gibt ihn ggf.an die anfordernde Treiberroutine als Ganzes zurück. Dabei wird der längste vorhandene Ausdruck erkannt. Steht in der Eingabe"A*B+C" so wird nicht nur "A*B" sondern "A*B+C" geliefert. Zum Erkennen von Ausdrücken wird die Methode des rekursiven Abstiegs verwendet. Aus Effektivitätsgründen wird sie jedoch nicht durch rekursive Prozeduren, sondern durch eine nichtrekursive Prozedur mit Hilfe eines Kellers (Stack, Last-In First-Out List) realisiert. Eine kurze Einführung in die Methode des rekursiven Abstiegs ist z.B. in /78/ enthalten.

Monitorroutinen

Außer dem Symbolentschlüsseler und den Prozeduren zur Verarbeitung von Prä - fixen, Zuweisungen und Ausdrücken enthält der Monitor des Übersetzers eine Reihe von Prozeduren, die von den Anweisungstreibern benutzt werden können zum Übersetzen von Problemsprachen-Anweisungen. Sie rufen dazu die Symbolentschlüsselung und die Ausdruckzerteilung auf. Im einzelnen gibt es Prozeduren für folgende Zwecke:

- Feststellen des Typs des nächsten Symbols in der Eingabe
- Liefern des nächsten Symbols in der Eingabe
- Liefern des nächsten Symbols, falls es einen vorgegebenen Typ hat
- Feststellen, ob eine bestimmte Benennung oder eine bestimmte Folge von Zeichen in der Eingabe als nächstes ansteht
- Feststellen, ob eine bestimmte Benennung oder eine bestimmte Folge von Zeichen in der Eingabe bis zum Anweisungsende vorkommt
- Liefern des nächsten Ausdruckes in der Eingabe
- Setzen des Eingabezeigers hinter das nächste Symbol, falls es einen bestimmten Typ hat oder eine bestimmte Folge von Zeichen in der Eingabe vorliegt (Übergehen von Symbolen)
- Liefern der gesamten gerade behandelten Anweisung als eine Zeichenkette
- Liefern der vor einer POL-Anweisung stehenden Präfixe

Außerdem enthält der Monitor Routinen zur Ausgabe einer Programmliste mit Anweisungsnumerierung, zur Ausgabe des erzeugten PL/1-Programmes und zum dynamischen Aufruf von Fehlerroutinen.

Aufruf von Treiberroutinen

Wenn der Monitor eine POL-Anweisung oder eine Systemanweisung erkannt hat, muß er die zugehörige Treiberroutine dynamisch aufrufen. Den Namen des Treibermoduls entnimmt er aus der Anweisungstabelle für das gerade aktive Subsystem. Der Aufruf wird durch eine Assembler-Interfaceroutine durchgeführt. Sie erhält außer den Argumenten für die Treiberroutine über ein zusätzliches Argument auch den Namen der zu rufenden Routine. Der Treibermodul wird mit Hilfe des LINK-Macros /79/ von der Treiberbibliothek dynamisch in den Arbeitsspeicher geladen. Die restlichen Argumente werden von der Interfaceroutine an die Treiberroutine weitergegeben. Da die Treibermodule als REUSABLE gekennzeichnet sind, werden inaktive Module, die von einem vorangegangenen Aufruf noch im Arbeitsspeicher liegen, bei einem erneuten Aufruf nicht neu von der externen Einheit geladen.

Der Zugriff von Treiberroutinen auf Fähigkeiten des Monitors (auf die Monitorroutinen) erfolgt über eine Schnittstelle in Form einer PL/1-Struktur. Der Zeiger auf diese Struktur wird an alle Treiberroutinen über ein Argument übergeben. Sie enthält Verweise auf die Monitorroutinen in PL/1-Entryvariablen. Die Treiberroutinen sprechen die in der Interfacestruktur deklarierten Entryvariablen in CALL - oder Funktionsaufrufen an, da diese Entryvariablen auf die entsprechenden Monitorroutinen verweisen, werden diese direkt aufgerufen (Abb. 26).

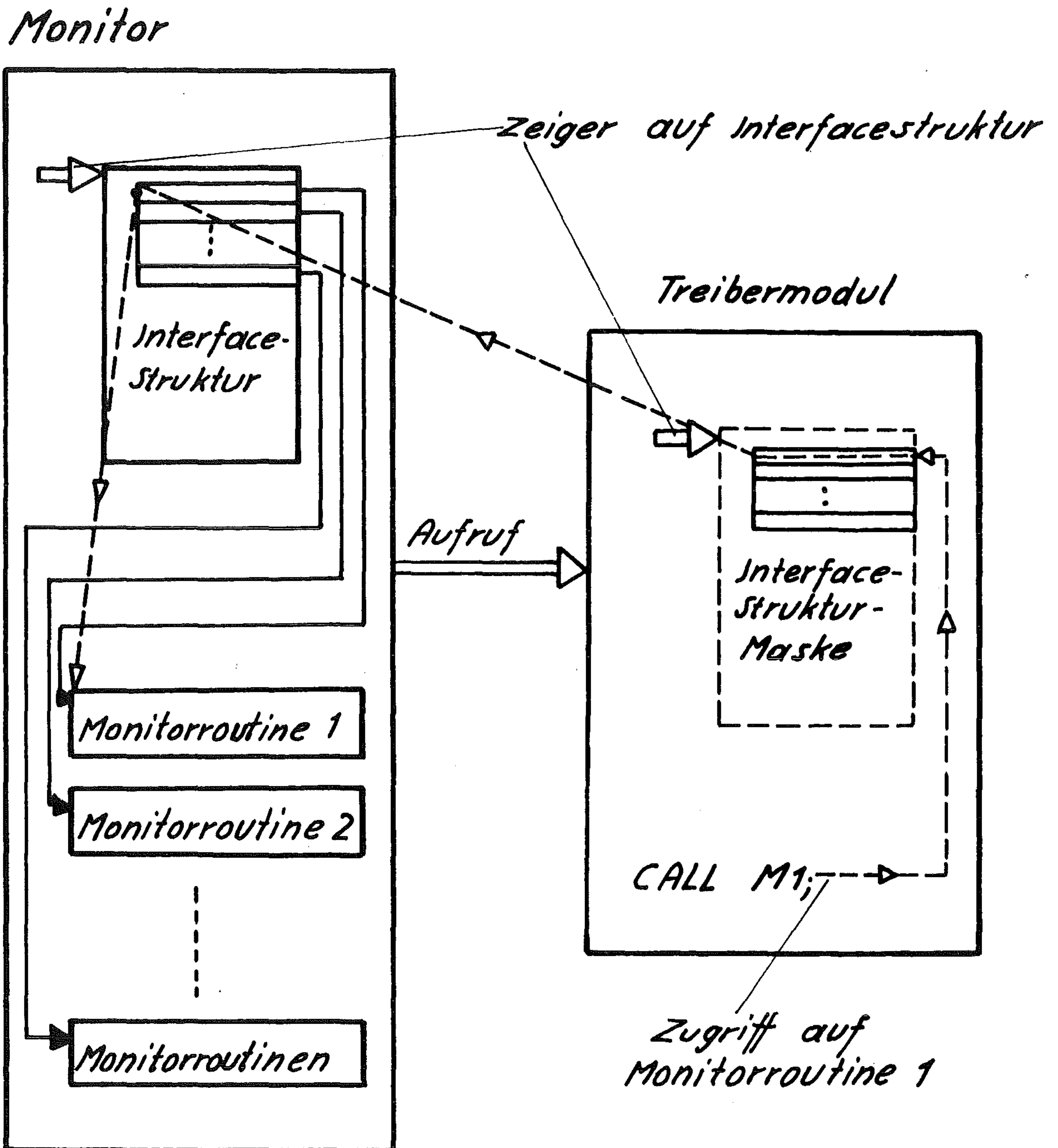


Abb. 26 Zugriff auf Monitorfähigkeiten aus den Treiberroutinen.

C2 Initialisierung und Zerstörung von Subsystemen

Durch eine Subsysteminitialisierung wird ein neues Subsystem in die PLS-Tabellen eingetragen, so daß für dieses Subsystem Anweisungen und Datenstrukturen definiert werden können und das Subsystem aufgerufen werden kann. Mittels der PLS-Anweisung INITIATE wird dem System der Name des Subsystems und ein Schlüsselwort bekanntgegeben. Dieses Schlüsselwort muß stets angegeben werden, wenn das Subsystem erweitert, geändert oder zerstört werden soll. Durch die Initialisierung werden folgende Einzelschritte durchgeführt:

- Die Subsystemtabelle wird erweitert, Name, Schlüssel, Initialisierung und Name der zugehörigen Anweisungstabelle werden eingetragen. Das Schlüsselwort wird vor dem Eintrag kodiert, um es vor unbefugtem Lesen zu schützen. Der Name der Anweisungstabelle (Membername in der Datenbibliothek) wird so erzeugt, daß er eindeutig ist. Ein ebenfalls eindeutiger Subsystem-Präfix, bestehend aus 2 Buchstaben wird generiert. Diese Buchstaben werden als Anfangsbuchstaben aller in einem Subsystem verwendeten Modulnamen, Treibermodulnamen, Datendeklarationennamen und Tabellennamen verwendet, so daß eine Namenskollision zwischen Bestandteilen verschiedener Subsysteme vermieden wird. In der Subsystemtabelle ist weiter vermerkt, ob der Subsystemname abkürzbar ist und wieviele Anweisungen und Datendeklarationen zum Subsystem gehören. Die Subsystemetabelle kann durch die PLS-Anweisung "LIST SUBSYSTEM;" ausgedruckt werden (Abb.27).

- Die Datenbibliothek wird um eine Anweisungstabelle für das Subsystem erweitert. Sie enthält zunächst lediglich die PL/1-Anweisungen und die Systemanweisungen, die in jedem Subsystem gültig sind, und eine Datenstruktur, den Subsystemcommon. Durch die PLS-Anweisung "LIST STATEMENTS OF SUBSYSTEM name;" wird der Inhalt der Anweisungstabelle für das Subsystem mit dem Namen "name" gelistet (Abb.28).

- Ein Subsystemcommon wird erzeugt und in die Datenbibliothek eingebracht, der nur den Subsystemnamen enthält.

Bei der Zerstörung von Subsystem durch die PLS-Anweisung DESTROY werden alle Treibermodule und Datendeklarationen und die Anweisungstabelle des Subsystems gelöscht, in der Subsystemtabelle wird der Name des Subsystems gestrichen.

```
************************************
*                                  *
*    LIST OF REGENT - SUBSYSTEMS   *
*                                  *
************************************
```

NAME OF SUBSYSTEM	PREFIX	CREATION DATE	SUB-SYSTEM LIST	ABB-REV. ?	NUMBER OF STATE-MENTS	NUMBER OF CLAU-SES	NUMBER OF DATA-DECL.
DABAL	DA	17.02.75	PLDABA	NO	77	0	2
FLOW	FL	20.02.75	PLFLOW	NO	89	3	1
GIPSY	GI	17.02.75	PLGIPS	NO	79	4	2
PLS	PL	01.05.73	PLPLS	NO	79	0	2
POLY	PO	17.02.75	PLPOLY	YES	69	0	2
REGENT	QQ	01.05.73	PLREG	NO	65	0	1
REMAC	RE	16.02.75	PLREMA	NO	72	0	2
SEDAP	SE	21.02.75	PLSEDA	YES	66	2	2
YAQUIR	YA	20.02.75	PLYAQU	YES	73	1	2

Abb.27 : Subsysteme-Tabelle

SUBSYSTEM - INFORMATION

NAME OF SUBSYSTEM	PREFIX	CREATION DATE	SUB-SYSTEM LIST	ABB-REV. ?	NUMBER OF STATE-MENTS	NUMBER OF CLAU-SES	NUMBER OF DATA-DECL.
PLS	PL	01.05.73	PLPLS	NO	79	0	2

NAME OF STATEMENT	CHARACTERISTIC	MODUL
%AS	PL/1-STATEMENT	
ASS	PL/1-STATEMENT	
%ACT	PL/1-STATEMENT,ALIAS	
%ACTIVATE	PL/1-STATEMENT	
%CONTROL	PL/1-STATEMENT	
%DCL	PL/1-STATEMENT,ALIAS	
%DEACT	PL/1-STATEMENT,ALIAS	
%DEACTIVATE	PL/1-STATEMENT	
%DECLARE	PL/1-STATEMENT	
%DO	PL/1-STATEMENT	
%END	PL/1-STATEMENT	
%GO	PL/1-STATEMENT	
%GOTO	PL/1-STATEMENT,ALIAS	
%IF	PL/1-STATEMENT	
%INCLUDE	PL/1-STATEMENT	
%PAGE	PL/1-STATEMENT	
%SKIP	PL/1-STATEMENT	
ALLOC	PL/1-STATEMENT,ALIAS	
ALLOCATE	PL/1-STATEMENT	
BEGIN	PL/1-STATEMENT	
CALL	PL/1-STATEMENT	
CHECK	PL/1-STATEMENT	
CLAUSE	POL-STATEMENT,ABBREV.	PLCLAU
CLOSE	PL/1-STATEMENT	
COMPRESS	POL-STATEMENT,ABBREV.	PLCOMP
CPARMS	POL-STATEMENT,ABBREV.	PLCPAR
DATASTRUCTURE	POL-STATEMENT,ABBREV.	PLDATA
DCL	PL/1-STATEMENT,ALIAS	
DECLARE	PL/1-STATEMENT	
DEFAULT	PL/1-STATEMENT	
DELAY	PL/1-STATEMENT	
DELETE	PL/1-STATEMENT	
DESTROY	POL-STATEMENT,ABBREV.	PLDEST
DFT	PL/1-STATEMENT,ALIAS	
DISPLAY	PL/1-STATEMENT	
DO	PL/1-STATEMENT	
ELSE	PL/1-STATEMENT	

Abb. 28: Anweisunstabelle (1.Teil)

NAME OF STATEMENT	CHARACTERISTIC	MODUL
END	POL-STATEMENT SUBST. FOR SYSTEM STATEMENT	PLENDB
ENTER	SYSTEM STATEMENT,ABBREV.	QCENTE
ENTRY	PL/1-STATEMENT	
EXIT	PL/1-STATEMENT	
FETCH	PL/1-STATEMENT	
FILE	POL-STATEMENT,ABBREV.	PLFILE
FINISH	SYSTEM STATEMENT,ABBREV.	QQFINI
FLOW	PL/1-STATEMENT	
FORMAT	PL/1-STATEMENT	
FREE	PL/1-STATEMENT	
GET	PL/1-STATEMENT	
GO	PL/1-STATEMENT	
GOTO	PL/1-STATEMENT,ALIAS	
IF	PL/1-STATEMENT	
INITIATE	POL-STATEMENT,ABBREV.	PLINIA
LIST	POL-STATEMENT,ABBREV.	PLLIST
LOCATE	PL/1-STATEMENT	
LPARMS	POL-STATEMENT,ABBREV.	PLLPAR
MACROTIME	POL-STATEMENT,ABBREV.	PLMACR
NOCHECK	PL/1-STATEMENT	
NOCOMPRESS	POL-STATEMENT,ABBREV.	PLNOCO
NOFLOW	PL/1-STATEMENT	
ON	PL/1-STATEMENT	
OPEN	PL/1-STATEMENT	
POOLSIZE	SYSTEM STATEMENT,ABBREV.	QQPOOL
PRINT	SYSTEM STATEMENT,ABBREV.	QQPRIN
PROC	PL/1-STATEMENT,ALIAS	
PROCEDURE	PL/1-STATEMENT	
PUT	PL/1-STATEMENT	
READ	PL/1-STATEMENT	
RELEASE	PL/1-STATEMENT	
RETURN	PL/1-STATEMENT	
REVERT	PL/1-STATEMENT	
REWRITE	PL/1-STATEMENT	
SIGNAL	PL/1-STATEMENT	
STATEMENT	POL-STATEMENT,ABBREV.	PLSTAT
STOP	PL/1-STATEMENT	
SUBSYSTEM	POL-STATEMENT,ABBREV.	PLSUBS
TRACE	SYSTEM STATEMENT,ABBREV.	QCTRAC
UNLOCK	PL/1-STATEMENT	
WAIT	PL/1-STATEMENT	
WRITE	PL/1-STATEMENT	
COMMON	DATA-DECLARATION	PLCOM1
PLS_DATA_STRUCTURE	DATA-DECLARATION	PLSYSI

Abb.28 (2.Teil)

Anhang C3: Definition von POL-Anweisungen und Datenstrukturen

Formale Syntax der POLs

Eine formale Sprache wird definiert durch die Beschreibung ihrer Syntax und ihrer Semantik. Zur formalen Syntaxbeschreibung der hier betrachteten Sprachen eignen sich kontextfreie Grammatiken, die verwendete Notation heißt Backus-Naur-Form oder Backus-Normal-Form (BNF,/62/). Eine kontext - freie Grammatik G ist gegeben durch ein Quadrupel /80/ :

$$G := (V,T,S,P) \quad \text{mit } P \subseteq V \times (V \cup T)^*,\ S \in V \qquad (8)$$

Dabei ist V die Menge der nichtterminalen Symbole der Grammatik, T die Menge der terminalen Symbole, S das Startsymbol der Grammatik und P eine Menge von Produktionen.

Es sei G_{PLI} die Grammatik der Grundsprache aller REGENT-POLs, also PL/1 erweitert um die Systemanweisungen:

$$G_{PLI} := (V_{PLT}, T_{PLI}, S_{PLI}, P_{PLI}) \qquad (9)$$

Eine formale Beschreibung der Syntax von PL/1 befindet sich in /81/. Diese Beschreibung stimmt allerdings nicht mit den durch die IBM-Optimising und Checkout-Compiler definierten PL/1-Syntax überein. In /51/ wird die Syntax und die Semantik des von der ECMA (European Computer Manufactures Association) und der ANSI(American National Standards Institute) erarbeiteten Normvorschlages für PL/1 beschrieben.

Die Grammatik einer Subsystem-POL sei

$$G_{SUB} := (V_{SUB}, T_{SUB}, S_{SUB}, P_{SUB}) \qquad (10)$$

Es gilt immer:

$$T_{SUB} = T_{PLI} \quad S_{SUB} = S_{PLI} \qquad (11)$$

Wenn die Subsystemsprache $L(G_{SUB})$ neue Anweisungen erhält, PL/1-Anweisungen jedoch weder geändert, erweitert,noch gestrichen wurden, dann gilt zusätzlich:

$$V_{SUB} \supset V_{PLI} \quad P_{SUB} \supset P_{PLI} \qquad (12)$$

Bei Erweiterung von PL/1-Anweisungen müssen nicht unbedingt neue nichtterminale Symbole in der Grammatik enthalten sein, so daß dafür gilt:

$$V_{SUB} \supseteq V_{PLI} \qquad (13)$$
$$P_{SUB} \supset P_{PLI}$$

Wenn im allgemeinsten Fall außer neuen Anweisungen und Erweiterungen von PL/1-Anweisungen auch Änderungen und Streichungen von PL/1-Anweisungen vorgenommen werden, so läßt sich über V_{SUB} und P_{SUB} keine Aussage bezüglich des Zusammenhangs zu V_{PLI} und P_{PLI} mehr machen. Bei Definition einer neuen POL-Anweisung wird die Grammatik G_{SUB} zunächst um eine Produktion erweitert, deren linke Seite dasjenige Nichterminalsymbol ist, das einer PL/1-Anweisung entspricht. In der in/51/angegebenen Grammatik heißt dieses Nichtterminalsymbol "executable-single-statement", die entsprechende Produktion lautet:

```
executable-single-statement::=
assignment-statement |
allocate-statement |
call-statement |
clear-statement |
close-statement |
delete-statement |
free-statement |
get-statement |                                   (14)
goto-statement |
locate-statement |
null-statement |
open-statement |
post-statement |
put-statement |
read-statement |
return-statement |
revert-statement |
rewrite-statement |
signal-statement |
stop-statement |
unlock-statement |
wait-statement |
write-statement |
```

Jede neue POL-Anweisungs-Definition erweitert diese Produktion um eine neue rechte Seite:

```
executable-single-statement::=
assignment-statement |
  :
write-statement |                               (15)
pol-statement-1 |
pol-statement-2 |
  :
```

Entsprechend werden andere Produktionen P_{PLI} um neue rechte Seiten erweitert, wenn bestehende PL/1-Anweisungen erweitert werden.
Durch die POL-Definition als PL/1-Erweiterung wird weder die unterste Grammatik-Ebene (low level syntax,/51/), die Ebene der PL/1-Symbole, noch die oberste Ebene (high level syntax), die Ebene der Prozeduren und Blöcke, sondern die mittlere Syntaxebene (middle level syntax), die Anweisungen und deren Teile beschreibt, geändert.

Rekursiver Abstieg

Zur syntaktischen Analyse komplexer POL-Anweisungen kann vorteilhaft die Methode des rekursiven Abstieges gewählt werden/78/. Bei dieser Methode wird jeder Produktion, die mehrere rechte Seiten haben kann, eine Bool'sche Prozedur zugeordnet, die den logischen Wert "true" liefert, falls eine rechte Seite zutrifft und "false" sonst. Jedem Nichtterminalsymbol der Grammatik entspricht genau eine Produktion und somit eine Bool'sche Prozedur. Wenn eine Prozedur "false" liefert, also keine der rechten Seiten zutrifft, muß der Eingabezeiger auf den Wert beim Aufruf zurückgesetzt werden (rekursiver Abstieg mit Backup). Eine rechte Seite einer Produktion trifft dann zu, wenn nacheinander die Prozeduraufrufe für Nichtterminale "true" liefern und die Terminale der Grammatik in der Eingabe vorkommen. Voraussetzung für die Anwendbarkeit des rekursiven Abstieges ist, daß die Grammatik nicht linksrekursiv ist. Wenn an jeder Stelle allein durch Betrachten des nächsten Eingabesymbols entschieden werden kann, welche der rechten Seiten einer Produktion angewandt werden muß, ist die Grammatik eine LL(1)-Grammatik/76/ und das Rücksetzen des Eingabezeigers kann vermieden werden (rekursiver Abstieg ohne Backup).

Im Interesse einer einfachen und effektiven Übersetzung sollte bei dem Entwurf der Subsystemanweisungen auf die LL(1)-Eigenschaft der Grammatik geachtet werden. Das grafische REGENT-Subsystem GIPSY/69/ bedient sich bei der Übersetzung der GIPSY-Sprache der Methode des rekursiven Abstiegs.

Schlüsselwort- und Datentypanweisung

Bei der Anweisungsdefinition wird angegeben, ob es sich um eine Schlüsselwort- (NAME) oder Datentypanweisung (DATATYPE) handelt. Datentypanweisungen beginnen nicht mit einem bestimmten Namen, sondern mit einem der elementaren Sprachbestandteile Operator, Bitkette, Zeichenkette, ganze oder reelle Zahl. Wenn also die Anweisungsdefinition mit: "STAT DATATYPE INTEGER ..." beginnt, so werden durch diese Definition alle Anweisungen behandelt, die mit einer ganzen Zahl beginnen. Bei namentlichen POL-Anweisungen muß die Teiberroutine den Namen der Anweisung nicht mehr verarbeiten, der Eingabezeiger steht rechts vom Namen der Anweisung. Bei Datentypanweisungen steht der Eingabezeiger am Beginn der Anweisung, so daß die Treiberroutine den Wert des ersten Elementes der Anweisung gewinnen kann.

PLS-Funktionen

Mit Hilfe von PLS-Funktionen kann auf Monitorfähigkeiten zugegriffen werden, um POL-Anweisungen abzuarbeiten. Sie steuern die Symbolentschlüsselung und die Ausdruckszerteilung. Die Syntax aller PLS-Funktionen ist im Anhang B 2 beschrieben. Eine Gruppe von PLS-Funktionen dient dazu, festzustellen welche Sprachbestandteile auf der POL-Anweisung vorhanden sind, eine zweite Gruppe gewinnt Information von der POL-Anweisung.

Zur ersten Gruppe gehört die Funktion TYPE, die feststellt von welchem Typ das nächste auf der Anweisung stehende Sprachelement ist (z.B. Benennung, Integer-Zahl, Operator, Zeichenkette). Mit der FIND-Funktion kann das Vorhandensein einer gegebenen Zeichenkette in der POL-Anweisung geprüft werden. Die PLS-Funktionen IDENTIFIER, WORD, BIDENTIFIER und BWORD stellen fest, ob eine vorgegebene Zeichenkette als nächstes auf der POL-Anweisung vorhanden ist und liefern

den logischen Wert '1'B zurück, falls dies zutrifft. Die (B) IDENTIFIER-Funktion prüft Benennungen, die (B) WORD-Funktion Ketten beliebiger Zeichen, das B steht für Beginn und bedeutet, daß nur die angegebenen ersten Zeichen mit den ersten Zeichen einer möglicherweise längeren Zeichenkette auf der Anweisung übereinstimmen müssen. Die Funktion ISEXPRESSION liefert den Wert '1'B, wenn als nächstes ein arithmetischer Ausdruck folgt, '0'B sonst.

Zur zweiten Gruppe von PLS-Funktionen gehören NEXT_REAL, NEXT_STRING, NEXT_BITSTRING, NEXT_OPERATOR, NEXT_IDENTIFIER, NEXT_N und NEXT_EXPRESSION. Sie liefern die nächste Real-Zahl, die nächste Zeichenkette oder Bitkette, den nächsten Operator, die nächste Benennung, die folgenden n Zeichen oder den arithmetischen Ausdruck, der auf der Anweisung als nächstes folgt. Die Funktion NEXT_ITEM liefert das nächste Sprachelement, mit Hilfe der PLS-Funktion THIS_STATEMENT kann die gesamte POL-Anweisung angefordert werden.

Makrozeitanweisungen

Eine Zusammenstellung der Makrozeitanweisungen befindet sich im Anhang B4. Die wichtigsten werden im folgenden näher beschrieben.

EXECUTE - END EXECUTE

Die Anweisungen EXEC und END EXEC umschließen die bei der Abarbeitung der POL-Anweisung zu generierenden PL/1-Anweisungen. Während der Übersetzungsphase des PLS-Übersetzers wird eine solche "EXEC-Gruppe" nicht ausgeführt, sondern in das erzeugte PL/1 Programm kopiert. Erst zur Ausführungszeit des generierten Programms werden die zwischen EXEC und END EXEC stehenden Anweisungen aktiv. Soll nur eine Anweisung erzeugt werden, kann sie auch zwischen "EXEC" und ";" stehen:

```
EXEC PUT LIST('BEISPIEL')SKIP;;
```

Dabei ist zu beachten, daß sowohl das Semikolon für die EXEC- Anweisung als auch das für die zu generierende Anweisung stehen muß. Mit Hilfe der Kurzform der EXEC-Anweisung lassen sich auch Teile von Anweisungen generieren: "EXEC MULT (A,B);" erzeugt den Teil einer PL/1-Anweisung "MULT (A,B)" (ohne Semikolon). In einer EXEC-Gruppe können beliebige PL/1-Anweisungen stehen.

Werden durch eine POL-Anweisung mehrere PL/1-Anweisungen erzeugt, so ist es erforderlich, sie als DO-Gruppe zu generieren, damit ein korrektes PL/1-Programm auch dann entsteht, wenn die POL-Anweisung später nach THEN oder ELSE benutzt wird. Beispiel: Die POL-Anweisung "DRUCKE ALLES;" soll die Anweisungsfolge "PUT EDIT (HEADLINE) (SKIP,A); CALL PRINT (3);" erzeugen. Wenn der POL-Programmierer nun schreibt: "IF X>0 THEN DRUCKE ALLES ;" und die erzeugten PL/1-Anweisungen stehen nicht zwischen DO und END ergeben sich Resultate, die der POL-Programmierer nicht erwartet (im Beispiel würde also die zweite Anweisung, "CALL PRINT (3);", nicht in der THEN -Clause stehen. Falls erforderlich, kann auch "BEGIN;" und "END"; die erzeugten Anweisungen umschließen. Dies ist immer dann nötig, wenn DECLARE- Anweisungen generiert werden, um doppelte Deklarationen zu vermeiden.

Der durch eine EXEC- Gruppe generierte PL/1-Text muß in Abhängigkeit von der in der POL-Anweisung stehenden Information variiert werden können. Eine Anweisung zur Mittelwertberechnung "MITTELWERT r_1, r_2, r_3,....r_n ;" soll den Mittelwert der reellen Zahlen r_i ausrechnen. Dazu sind Anweisungen zur Summenbildung und zur Division durch die Anzahl zu erzeugen. Die Anweisungsdefinition lautet:

```
            STATEMENT 'MITTELWERT';
            DCL I BIN FIXED (15);
            I = 0;
            EXEC SUM = 0.;;
            DO WHILE (TYP¬ = 10);
(1)                EXEC SUM = SUM + NEXT_REAL;;
                   SKIP (',') ; I = I + 1;
                   END ;
(2)         EXEC PUT LIST (' MITTELWERT:',SUM /I);;
            END STATEMENT;
```

In der EXEC-Gruppe (1) wird anstelle des Namens "NEXT_REAL" jeweils der Wert eingesetzt, den die PLS-Funktion aus der Eingabe gewinnt. Ebenso wird in der EXEC-Gruppe (2) der Name der Makrozeitvariablen I durch ihren Wert ersetzt. Variable, deren Namen in der EXEC-Gruppe durch ihren Wert ersetzt werden, heißen " Ersetzungsvariable", ihr Wert heißt " Ersetzungswert".

Eine Ersetzung findet nur für <u>aktive</u> Variable statt. Aktiv sind neben den PLS-Funktionen alle arithmetischen oder Zeichenkettenvariablen (mit den Attributen BINARY, DECIMAL, PICTURE oder CHARACTER)

innerhalb des Blocks, in dem sie durch eine DECLARE- Anweisung deklariert sind. Im obigen Beispiel sind also die Variablen I (da als BINARY deklariert) und NEXT_REAL (da PLS-Funktion) aktiv.
Mit Hilfe von ACTIVE-und UNACTIVE-Anweisungen können unaktive Variable aktiviert und aktive desaktiviert werden.

ACTIVE

Die ACTIVE-Anweisung ist keine ausführbare Anweisung, sondern eine Deklaration, die innerhalb des PL/1- Blockes gültig ist, in dem sie steht. Die nach "ACTIVE" aufgeführten Variablennamen werden, wenn sie in einer EXEC-Gruppe angetroffen werden, durch den Wert der Variablen dieses Namens ersetzt. Mit der ACTIVE-Anweisung können Variable aktiviert werden, die nicht standardmäßig aktiv sind, wie etwa Bitketten. Auch nicht explizit deklarierte Variable oder solche, denen durch eine DEFAULT- Anweisung Attribute zugewiesen werden, können nur durch die ACTIVE -Anweisung zu Ersetzungsvariablen werden.

UNACTIVE

Ebenso wie ACTIVE ist UNACTIVE keine ausführbare Anweisung, sondern eine Deklaration. Die ausgeführten Variablen sind in dem PL/1-Block, in dem die Anweisung steht, nicht aktiv. Es können standardmäßig aktive Variable somit von der Ersetzung in EXEC- Guppen ausgeschlossen werden. Im äußersten Block einer STATEMENT- oder CLAUSE-Definition können auch PLS-Funktionen desaktiviert werden.

EXECUTE LINK routine

Innerhalb einer EXEC- Gruppe ist nur eine Nicht-PL/1-Anweisung zulässig, nämlich LINK. Diese Anweisung dient dazu, zur Ausführungszeit des POL- Programmes Entries in PLR-Moduln dynamisch aufzurufen. "routine" muß ein Name eines PLR-Moduls und zugleich ein Entry-Point in diesem Modul sein. Das dynamische Laden und Ausführen des Moduls wird von der REGENT-Modulverwaltung RMM vorgenommen. Der Modul muß daher ein vom REGENT-Modulgenerator erzeugter PLR-Modul sein.
An den Modul können Argumente übergeben werden. Ein Entry, der in einem Subsystem duch EXEC LINK aufgerufen wird, muß in der Subsystem-Datenstruktur (vorteilhafterweise im Subsystem-Common) als DYNAMIC ENTRY mit allen Parametern deklariert sein.

SKIP

Die SKIP-Anweisung dient zum Übergeben eines Elementes in der Eingabe, "SKIP;" setzt den Eingabezeiger hinter das gerade anstehende Symbol. "SKIP condition;" verändert den Eingabezeiger nur, wenn "condition" zutrifft. "SKIP REAL;" bewirkt, falls als nächstes eine reelle Zahl in der Eingabe ansteht, das Setzen des Zeigers hinter das Ende dieser Zahl. "SKIP ('XYZ');" verändert den Zeiger nur, wenn die Zeichen "XYZ" als nächstes auf der POL-Anweisung vorhanden sind. Entsprechend gibt es SKIP-Anweisungen für Zeichenketten, Integer-Zahlen, Bennungen, Operatoren und Ausdrücke.

PLI

Wenn PL/1 - Anweisungen oder Systemanweisungen um neue Bestandteile erweitert werden sollen, ist eine STATEMENT-Definition mit dem Namen der PL/1- (bzw. System-) Anweisung erforderlich. Wenn bei der Abarbeitung der Anweisung festgestellt wird, daß es sich um eine nicht erweiterte PL/1- (System-) Anweisung handelt, kann die PLI-Anweisung benutzt werden, um dem Übersetzer mitzuteilen, daß er die betreffende POL-Anweisung als unmodifizierte PL/1-oder System-Anweisung betrachten soll.
Beispiel: Die PUT-Anweisung, die in PL/1 eine der Optionen EDIT, LIST oder DATA enthalten muß, soll um eine Option DISPLAY zur Ausgabe auf ein Sichtgerät erweitert werden. Die Anweisungsdefinition könnte lauten:

```
STATEMENT 'PUT';
IF ¬FIND ('DISPLAY') THEN PLI;
     Übersetzung für PUT-Anweisung
     mit DISPLAY - Option
END STATEMENT;
```

Wenn in der PUT-Anweisung die Benennung "DISPLAY" nicht vorhanden ist, wird die Anweisung als nicht erweiterte PL/1-Anweisung behandelt.

Clauses

Enthalten mehrere POL-Anweisungen gleiche Teilgruppen, so kann zur Abarbeitung dieser Teile ein (externes) Unterprogramm aufgerufen werden. Diese Unterprogramme heißen Clauses, sie werden durch die CLAUSE-Anweisung definiert. Die Makrozeitanweisung "LINK clausename;" dient zum Aufruf von Clauses, "clausename" ist der bei der CLAUSE-Definition angegebene Name.
Beispiel: Es sollen die drei Anweisungen PRINT, PUNCH und PLOT definiert werden mit der folgenden Syntax:

```
    PRINT gruppe;
    PUNCH gruppe;
    PLOT  gruppe;
    gruppe ::=   identifier [, identifier ] *
```

Die Definition lautet:

```
    STA 'PRINT';
    EXEC ... ;
    LINK GRUPPE;
    END STA;
    STA 'PUNCH';
    EXEC ... ;
    LINK GRUPPE;
    END STA;
    STA 'PLOT';
    EXEC ... ;
    LINK GRUPPE;
    END STA;
    CLAUSE'GRUPPE';
           Verarbeite Gruppe von Benennungen
    END CLAUSE;
```

Durch die Anweisung "CLAUSE INITIAL;" wird eine Clause definiert, die beim Subsystemstart (ENTER subsystem;) aufgerufen wird.
Sie kann zum Initialisieren von Datenstrukturen und Dateien dienen und z.B. die Ausgabeliste des Sybsystems mit einer Überschrift versehen.

Löschen von Anweisungen

Sowohl neu definierte POL-Anweisungen und Clauses als auch PL/1- und Systemanweisungen können mittels der PLS-Anweisung DESTROY gelöscht werden. Eine Subsystemsprache kann dadurch so kozipiert werden, daß sie lediglich einen PL/1-Subset oder gar kein PL/1 enthält.

Fehlerbehandlung in Anweisungsdefinitionen

Der Subsystemprogrammierer hat die Möglichkeit, Fehler in der Syntax von POL-Anweisungen zu erkennen. In manchen Fällen wird PLS eine Fehlernachricht erzeugen, z.B. immer dann, wenn ein bestimmter Datentyp (NEXT_REAL, NEXT_ID, ...) in einer STATEMENT- oder CLAUSE-Definition erwartet wird, aber in der POL-Anweisung nicht vorhanden ist. Ebenso, wenn ein Anweisungstreiber die Kontrolle an den PLS-Übersetzer zurückgibt, ohne daß die POL-Anweisung vollständig abgearbeitet ist.

In vielen Fällen wird jedoch der Subsystemprogrammierer selbst eine Fehlernachricht an den Subsystembenutzer geben und eine Fehlerreaktion vorsehen wollen. Die einfachste Möglichkeit für Fehlermeldungen ist, eine PUT-Anweisung auf die Standard-Ausgabe-Datei SYSTPRINT in die STATEMENT oder CLAUSE-Definition hineinzuschreiben, z.B. : PUT EDIT ('FEHLER IM ZEICHNE-STATEMENT') (SKIP,A);. Für informatorische Nachrichten, d.h. solche, die bei jedem Aufruf der Treiberroutine erfolgen sollen, ist dies auch die adäquate Methode. Fehler treten jedoch i.a. nicht ständig, sondern selten auf, so daß durch die Fehlermeldungs-Anweisungen samt zugehörigen PL/1 -Printroutinen die Treibermodule durch Code aufgebläht werden, der nur im Fehlerfalle ausgeführt wird. Es gibt die Möglichkeit, Fehlerroutinen, die ein Standardformat besitzen, dynamisch aufzurufen. Sie werden also erst im Fehlerfalle aus einer Bibliothek geladen und ausgeführt. Den Aufruf einer Fehlerroutine erledigt eine Interface-Routine (QQERROR).
Der Fehlerroutine können Daten zur Verfügung gestellt werden oder Ergebnisse zurückgegeben werden. Alle Fehlerroutinen enthalten Anweisungen zur Ausgabe einer Nachricht auf die Standard-Ausgabe-Datei. Die Fehlerroutine kann außerdem beliebige andere Anweisungen enthalten. Sie kann auch auf PLS-Funktionen zugreifen.

Datenstruktur-Definition

Datenstrukturen in Form von PL/1 Datendeklarationen, die in einem Subsystem während dessen Ausführung benötigt werden, können bei der Subsystemerstellung einmal definiert werden. Sie können danach bei jeder Subsystemanwendung benutzt werden. Datenstrukturen, die zur Übersetzungszeit der Subsystemsprache (zur Makrozeit) zum Zweck der Sprachübersetzung (z.B. zur Kommunikation zwischen Anweisungstreibern) verwendet werden, heißen Makrozeit-Datenstrukturen. Datenstrukturen, die zur Ausführungszeit des Subsystems benutzt werden, werden hier Subsystem-Datenstrukturen genannt.

Subsystem-Datenstrukturen werden zwischen den PLS-Anweisungen "DATA-STRUCTURE dname;" "END DATASTRUCTURE ;" definiert als Folge von PL/1 DECLARE -Anweisungen. Diese Deklarationen werden beim Übersetzen von POL-Anweisungen nach dem Eröffnen eines Subsystems (ENTER subname;) in das erzeugte PL/1-Programm eingeschoben und stehen somit für die Dauer der Subsystem-Anwendung zur Verfügung. Beliebig viele Subsystem-Datendeklarationen können angegeben werden. Jede Folge von Daklarationen wird durch einen Namen identifiziert. Durch Angabe dieses Namens in einer "DESTROY DATASTRUCTURE" -Anweisung kann die Datenstruktur wieder gelöscht werden. Die Subsystem-Datenstruktur dient zur Aufbewahrung von Subsystem-Daten während der Ausführungszeit und zur Kommunikation zwischen Subsystem-Routinen. Dem POL-Programmierer kann die Möglichkeit gegeben werden, in der Subsystem-Datenstruktur von den Subsystem-Moduln abgelegte Werte zu verwenden. Eine ausgezeichnete Datenstruktur für jedes Subsystem ist der Subsystem-Common. Er wird mittels der Anweisung "DATA STRUCTURE COMMON;" definiert. Eine Deklaration in Form einer einzigen PL/1-"Structure" wird benutzt, um alle diejenigen Subsystem-Daten aufzunehmen, die nicht nur in der POL, sondern auch in allen Subsystem-Moduln ansprechbar sein sollen. Auch alle externen subsystemspezifischen POL-Routinen können auf den Subsystem-Common zugreifen.

Während eine namentlich identifizierte Subsystem-Datenstruktur-Definition neben beliebigen DECLARE-Anweisungen auch ausführbare PL/1-Anweisungen enthalten darf, die beim Start des Subsystems ausgeführt werden sollen, darf der Subsystem-Common nur die Deklaration der COMMON-Struktur enthalten. Die Subsystem-Datenstrukturen müssen

deklariert werden, bevor sie in einem Modul benutzt werden können. Für die Datendeklarationen sind außer PL/1-Datenattributen auch die REGENT-Attribute DYNAMIC ENTRY, BANK und BASEDESCRIPTOR zulässig. Für jeden zu dem Subsystem gehörenden auf der PLS-Ebene dynamisch aufrufbaren Modul (also für jeden Modul, dessen Name in einer EXEC LINK-Anweisung in einer STATEMENT- oder CLAUSE- Definition erscheint) ist im Subsystem-Common eine "DECLARE.....ENTRY(.....) DYNAMIC"-Deklaration erforderlich.
Der Subsystem-Common wird beim Subsystem-Start angelegt und initialisiert. Die Initialisierung ist möglich durch Angabe von INIT-Optionen in der Deklaration oder durch Anweisungen, die in einer "INITIAL CLAUSE" stehen. Die Übergabe des Common an die Subsystemmodule und externe POL-Prozeduren erfolgt über den REGENT-Basispointer (QQ), siehe Abb. 29. Die namentlich gekennzeichneten Datenstrukturen werden nicht automatisch an externe POL-Prozeduren und Subsystem-Moduln übergeben. Die Übergabe von wichtigen Daten an Subsystem-Moduln liegt in der Verantwortung des Subsystem-Programmierers. Ein Subsystem kann zur Ausführungszeit statt auf externe Moduln auf interne PL/1-Routinen zugreifen. Diese müssen in einer Subsystem-Datenstruktur enthalten sein. Aufgerufen werden sie zweckmäßigerweise über ENTRY-Variable, die im Subsystem-Common stehen und mit den internen Prozeduren initialisiert sind.
Beispiel:

```
DATA COMMON;
  DCL  1,
         .
         .
         .
       2 RV1 ENTRY VARIABLE INIT (R1),
       2 RV2 ENTRY VARIABLE INIT (R2);
END DATA;
DATA 'PROCEDURES';
  R1:  PROC;
         .
         .
       END R1;
  R2:  PROC;
         .
         .
       END R2;
END DATA;
```

Über die Namen RV1 und RV2 kann so im gesamten Subsystem (in der POL, in allen Moduln und in allen externen subsystemspezifischen POL-Routinen) auf die Prozeduren R1 und R2 zugegriffen werden.

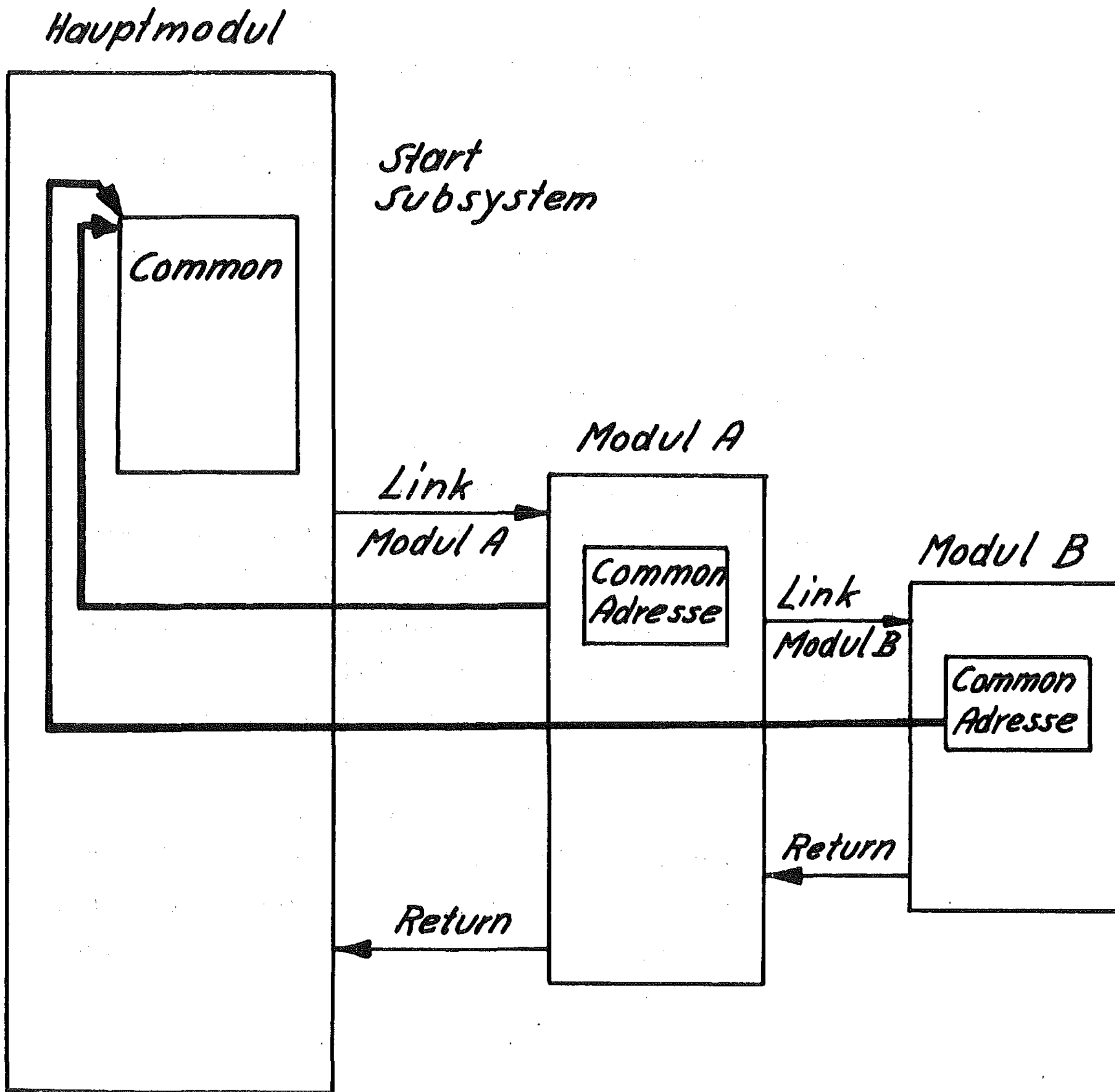


Abb. 29: Zugriff auf die Subsystem - Common-Daten - struktur aus Subsystemmoduln

Wenn R1 und R2 externe Prozeduren sind, muß die Datenstruktur-Deklaration in obigem Beispiel lauten:

```
DATA'PROCEDURES';
DCL (R1,R2) ENTRY EXTERNAL;
END DATA;
```

Die Subsystem-Datenstrukturen existieren während einer Subsystem-Anwendung nur zur Ausführungszeit des kompilierten POL-Programmes, während der Übersetzungsphase sind sie nicht vorhanden. Die Anweisungs-Treiber-Routinen, die dazu dienen, eine bestimmte Anweisung zu expandieren, sind als unabhängige Module in einer Bibliothek gespeichert und werden bei Bedarf dynamisch in den Arbeitsspeicher geladen. Sie können daher untereinander nicht über globale interne oder auch externe Variable kommunizieren. Mit der Anweisung "MAKROTIME DATA-STRUCTURE" wird eine Übersetzungszeit-Datenstruktur deklariert, die den Treiberroutinen eine Kommunikation untereinander ermöglicht. Die hier deklarierten Variablen sind den globalen Makrozeit-Variablen der OS/360-Assembler Macrolanguage/61/oder in PL/1-Macrotime-Procedures /47/ deklarierten Variablen vergleichbar. Die Übersetzungs-Zeit-Datenstruktur ist eine BASED PL/1-Struktur mit festen Längen und Dimensionen. Sie wird im PLS-Übersetzer-Kern angelegt. Ein Zeiger auf die Struktur wird an alle Treiberroutinen übergeben, so daß auf die globalen Übersetzungszeitvariablen in allen Routinen zugegriffen werden kann. Zur Übersetzungszeit können zum Abspeichern von Werten verkettete Listen (linked lists) verwendet werden. Der Listenkopf muß dann in der Übersetzungszeit-Datenstruktur gespeichert sein, damit alle Treiberroutinen auf die Listen zugreifen können. Mit Hilfe von verketteten Listen können so auch zur Übersetzungszeit Daten mit variablem Speicherplatzbedarf verwendet werden.

Wie die Subsystem-Datenstrukturen, muß auch die Übersetzungszeit-Datenstruktur deklariert werden, bevor sie in einer STATEMENT- oder CLAUSE-Definition benutzt wird. Erweiterungen am Ende der Struktur können vorgenommen werden, ohne daß alle Treiberroutinen neu übersetzt werden müssen. Wird jedoch die Übersetzungszeit-Datenstruktur abgeändert, so müssen alle STATEMENT- und CLAUSE-Definitionen wiederholt werden, die auf die Struktur zugreifen.

Verwaltung von Bibliotheken

Die zur Übersetzung von Subsystem-POLS erforderlichen Bibliotheken werden während der POL-Definition erweitert oder modifiziert. Da die Bibliotheken als "Partitioned Data Sets" (PDS) realisiert sind, können sie durch allgemein verfügbare Dienstprogramme allokiert, gelöscht und kopiert werden. Die Bestandteile (Members) der Treiberroutinenbibliothek sind ausführbare Lademodule, die Datenbibliothek enthält die Subsystem- und Anweisungs-Tabellen und die Datenstrukturdeklarationen in Form von PL/1- Deklarationen. Da es in PL/1 keine Anweisungen zum Erweitern und Ändern von Bibliotheken gibt, wurden zur Bibliotheksverwaltung eine Reihe von Assemblerroutinen zum Schreiben , Lesen und Löschen von PDS-Members erstellt. Wird ein Member eines PDS geändert, wird die neue Version an das Ende der Bibliothek angefügt, der Platz, den das Member vorher belegt hatte, läßt eine ungenutzte Lücke zurück. Daher wächst der Platzbedarf eines PDS bei häufiger Änderung immer mehr an. Durch die Neuordnung der Members können die Lücken in einem PDS wieder beseitigt werden, dieses "Komprimieren" wird durch das Dienstprogramm IEBCOPY/82/ durchgeführt. Um ein Überlaufen zu verhindern, werden die PDS-Bibliotheken am Ende jedes PLS-Laufes, in dem sie verändert wurden, standardmäßig komprimiert. Durch die Anweisung "NOCOMPRESS" kann das Komprimieren verhindert, durch "COMPRESS" ein Komprimieren erzwungen werden.

Da im allgemeinen die Veränderung der Bibliotheken durch neue POL-Definitionen gleichzeitig mit der Benutzung der Bibliotheken durch POL-Übersetzungen anderer Anwender geschieht, muß auf jeden Fall eine Zerstörung der Bibliotheksinhalte durch umkoordinierten gleichzeitigen Zugriff verhindert werden. Dies wird dadurch erreicht, daß bei POL-Definitionen mittels des Subsystems PLS die Bibliotheken für exklusiven Zugriff reserviert werden. Während der Zeit, in der ein Anwender die Dateien verändert, kann sie also ein anderer Anwender weder ebenfalls ändern noch benutzen. Für die Dateireservierung wird über eine Assemblerroutine auf eine Betriebssystemfähigkeit (RESERVE-Makro,/83/) zugegriffen.

Ablauf der Definition in zwei Pässen

Bei der Bearbeitung der STATEMENT-und CLAUSE- Definitionen werden aktive Makrozeitvariable daran erkannt, daß sie im betreffenden PL/1-Block in einer Deklaration oder einer ACTIVE-Anweisung vorkommen und nicht in einer UNACTIVE-Anweisung aufgeführt sind. Daher muß die Blockstruktur erkannt werden. Da Deklarationen in PL/1 nach den Anweisungen stehen können, in denen die entsprechenden Variablen referiert werden, sind mindestens zwei Durchgänge (Pässe) zur Bearbeitung der Definitionen erforderlich. Der erste Pass, bei dem die Blockstruktur erkannt, die Deklarationen, ACTIVE- und UNACTIVE-Anweisungen untersucht und der Typ der restlichen Anweisungen festgestellt wird (PL/1- und PLS- Makrozeitanweisungen), wird während der Übersetzungsphase des PLS-Programmes durchgeführt. Der modifizierte Text der STATEMENT- oder CLAUSE- Definition wird über eine Zwischendatei an die Ausführungsphase des PLS-Programmes übergeben. Die Zwischendatei mit dem OS-DD- Namen COMFILE ist ein PDS, jeder STATEMENT- oder CLAUSE- Definition entspricht ein Member. Zur PLS-Ausführungszeit werden die Members wieder eingelesen, in einem zweiten Pass wird die Definition ergänzt und eine komplette PL/1-Prozedur erzeugt, die Treiberroutine (siehe Abb. 30). Für jede Treiberroutine wird der PL/1- Compiler gerufen, der sie übersetzt und den Objektcode auf eine Zwischendatei schreibt. Durch die FILE- Anweisung (siehe Anhang B 3) wird das Binden der Anweisungstreiber zu ausführbaren Lademoduln, die Übertragung der Module auf die Treiberbibliothek und die Erweiterung der Anweisungstabellen veranlaßt. Anweisungsdefinitionen können so erst auf korrekte Syntax geprüft werden, erst wenn die Treiberroutinen fehlerfrei sind, erfolgt durch die FILE-Anweisung die permanente Änderung der Bibliotheken. Die FILE-Anweisung kann außerdem benutzt werden, um verschiedene STATEMENT- und CLAUSE- Definitionen in einen Treibermodul zusammenzufassen. Dies kann in Sonderfällen die Effektivität der POL-Übersetzung steigern. Das Binden wird durch den Linkage Editor des OS/ 360 /84/ durchgeführt. Mit den PLS-Anweisungen CPARMS und LPARMS können die Parameter für den Optimising Compiler und den Linkage Editor während der Treibermodul-Erzeugung geändert werden.

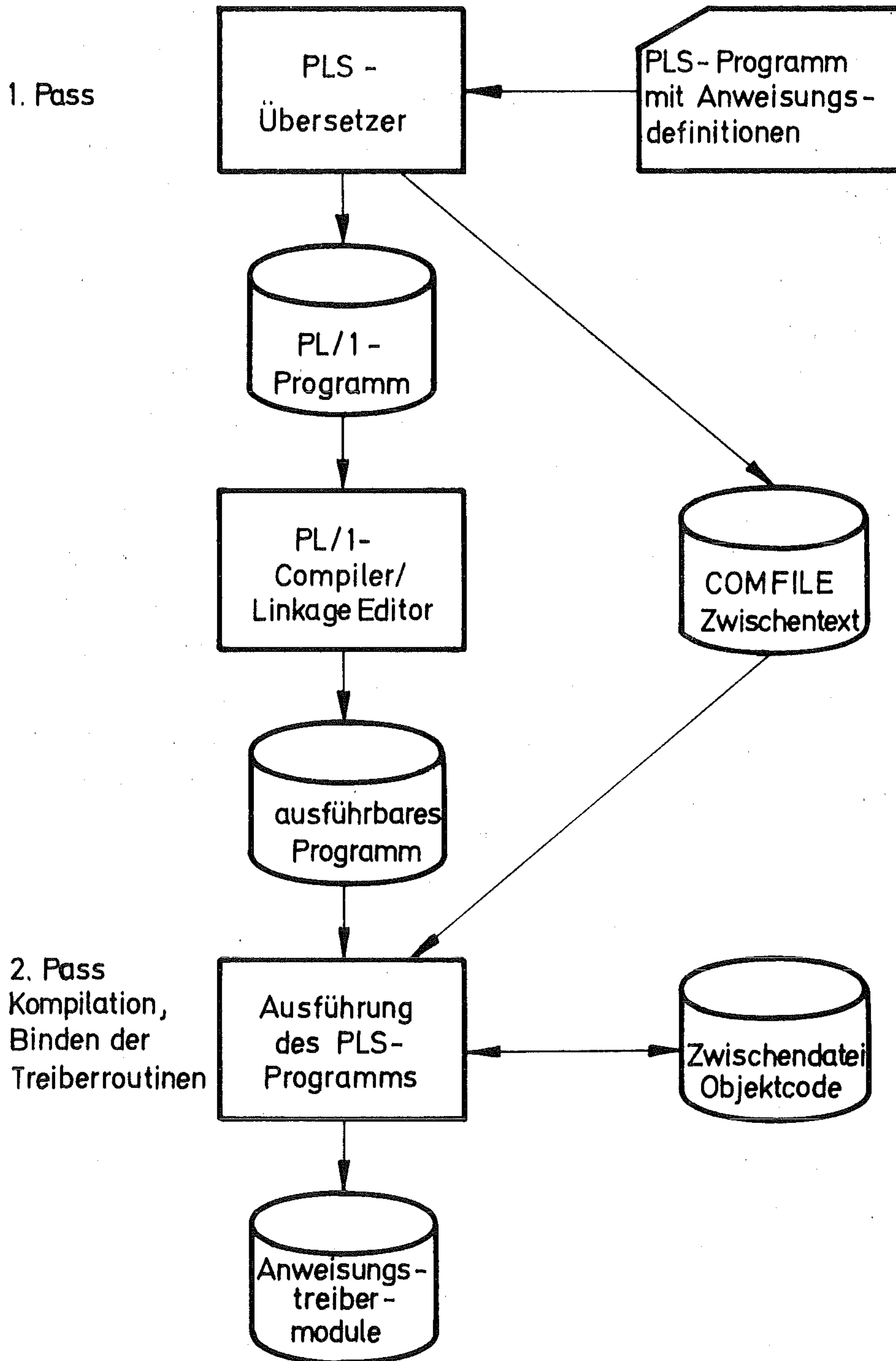


Abb. 30: Realisierung der Bearbeitung von Anweisungsdefinitionen

Die Datenstrukturdefinitionen werden wie die Anweisungsdefinitionen über die Zwischendatei COMFILE an die PLS- Ausführungsphase übergeben. Dort werden sie auf die Datenbibliothek übertragen, die Tabellen werden ergänzt. Der Subsystem-Common muß dabei in zwei Versionen generiert werden, einmal als AUTOMATIC- Struktur, wie sie während der Subsystemausführung in das aus dem POL- Programm generierte PL/1-Programm eingefügt werden muß und zum zweiten als BASED- Struktur. Diese zweite Common-Deklaration dient in Subsystem-Moduln und externen POL- Prozeduren als Maske für den Zugriff auf den im Hauptmodul angelegten Common.

Bootstrapping

Während der Erstellung des PLS- Subsystems konnte nicht von Anfang an auf die Sprachdefinitionsfähigkeiten von PLS selbst zugegriffen werden. Daher wurden die Systemanweisungen "ENTER subsystem;" und "END subsystem;" sowie die PLS- Anweisungen "STATEMENT" und "CLAUSE" direkt programmiert. Alle anderen Systemanweisungen und PLS-Anweisungen wurden in zwei Bootstrapping-Schritten mittels der schon vorhandenen Grundfähigkeiten des PLS- Subsystems definiert.

Anhang D

Zeichenklassen und Symboltypen

D1. Zeichenklassen der Routine GETCHAR

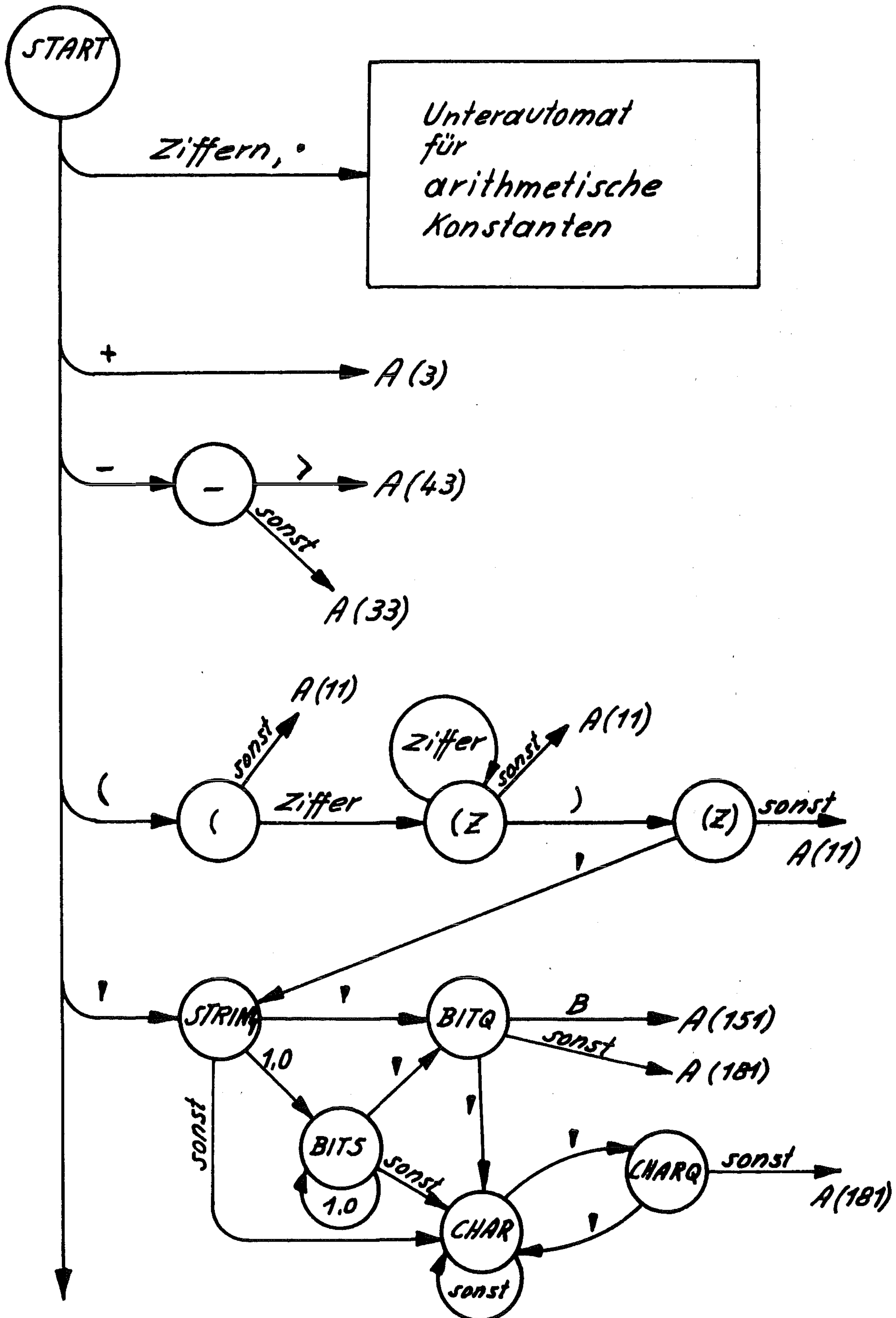
<table>
<tr><th colspan="2">CHARACTER-CLASSES DELIVERED BY PROCEDURE GETCHAR IN VARIABLE CHAR</th></tr>
<tr><th>CLASS</th><th>TYPE OF CHARACTER</th></tr>
<tr><td>1</td><td>DIGITS 0,1</td></tr>
<tr><td>2</td><td>DIGITS 2,3,...9</td></tr>
<tr><td>3</td><td>LETTERS A,B,...Z,$,#,@</td></tr>
<tr><td>4</td><td>+</td></tr>
<tr><td>5</td><td>-</td></tr>
<tr><td>6</td><td>'</td></tr>
<tr><td>7</td><td>*</td></tr>
<tr><td>8</td><td>/</td></tr>
<tr><td>9</td><td>></td></tr>
<tr><td>10</td><td><</td></tr>
<tr><td>11</td><td>=</td></tr>
<tr><td>12</td><td>¬</td></tr>
<tr><td>13</td><td>&</td></tr>
<tr><td>14</td><td>|</td></tr>
<tr><td>15</td><td>(</td></tr>
<tr><td>16</td><td>)</td></tr>
<tr><td>17</td><td>.</td></tr>
<tr><td>18</td><td>;</td></tr>
<tr><td>19</td><td>,</td></tr>
<tr><td>20</td><td>:</td></tr>
<tr><td>21</td><td>_</td></tr>
<tr><td>22</td><td>%</td></tr>
<tr><td>23</td><td>BLANK</td></tr>
<tr><td>24</td><td>OTHER CHARACTERS</td></tr>
<tr><td>25</td><td>END OF FILE ON INPUT</td></tr>
</table>

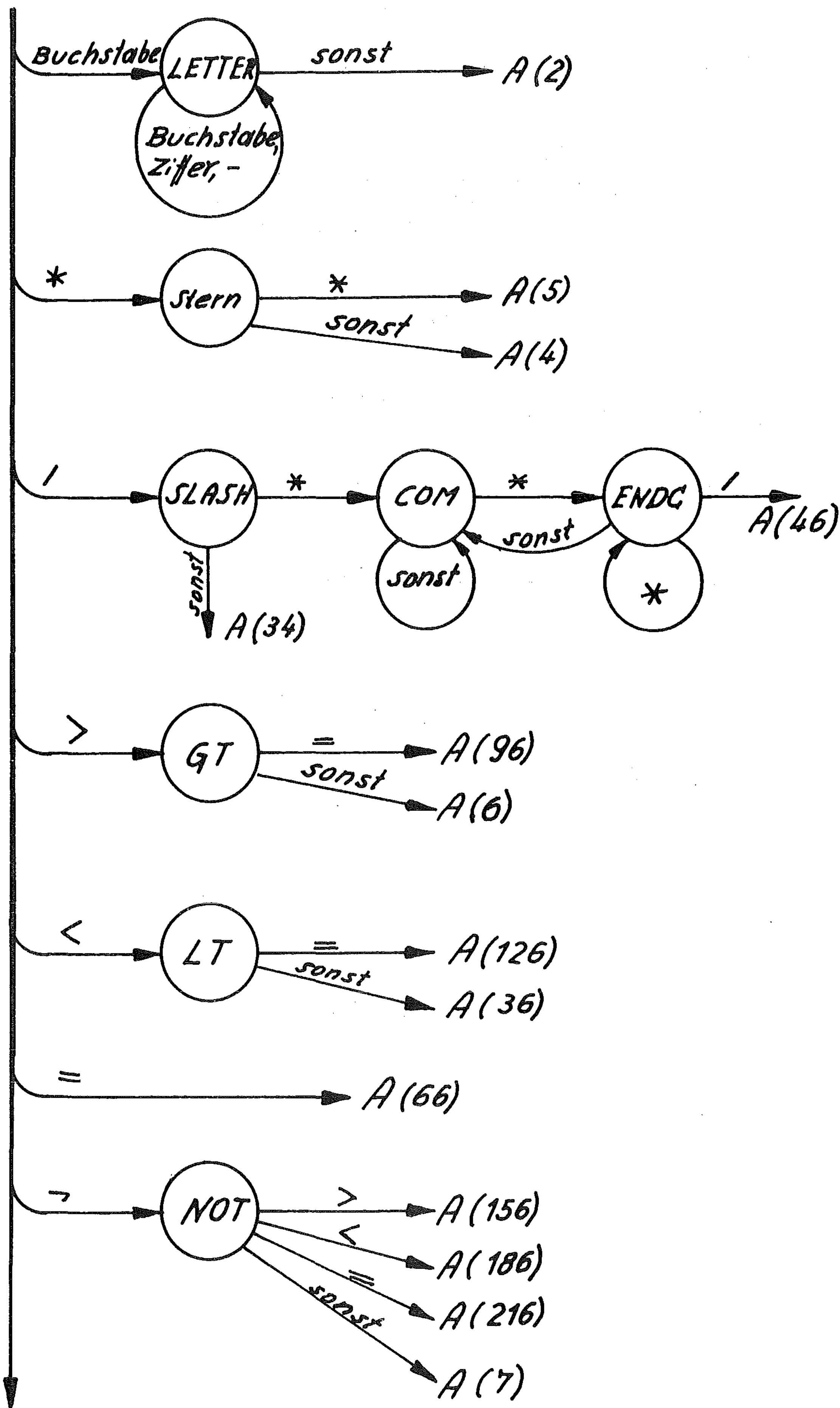
D2. Symboltypen der Symbolentschlüsselung

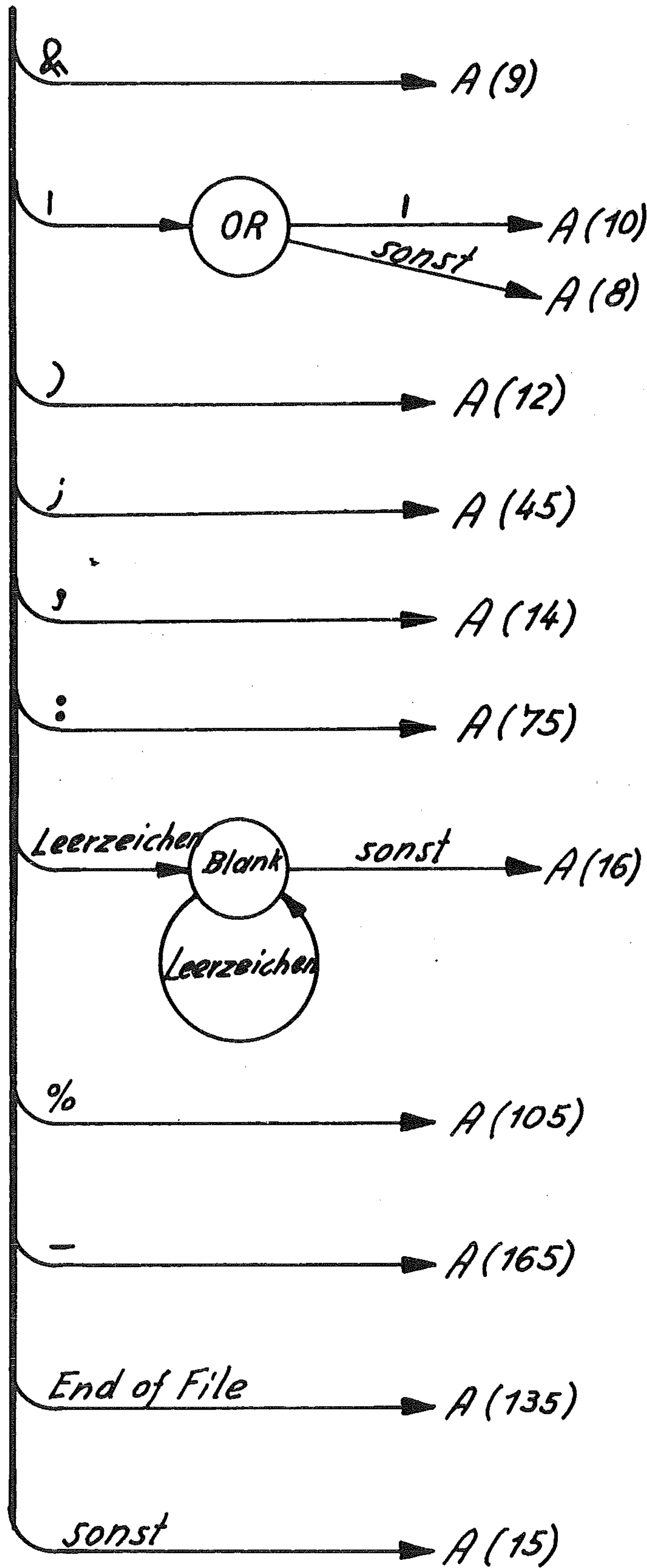
TYPE	SYMBOL
1	DECIMAL INTEGER
31	DECIMAL REAL
61	BINARY INTEGER
91	BINARY REAL
121	IMAGINARY CONSTANT
151	BITSTRING
181	CHARACTERSTRING
2	IDENTIFIER
3	+
33	-
4	*
34	/
5	**
6	>
36	<
66	=
96	>=
126	<=
156	¬>
186	¬<
216	¬=
7	¬
8	|
9	&
10	||
11	(
12	)
13	.
14	,
43	->
16	BLANK(S)
15	NON-PL/1-CHARACTER
45	;
46	COMMENT
75	:
105	%
135	END OF FILE ON INPUT
165	_

Anhang E: Automatendiagramm der Symbolentschlüsselung

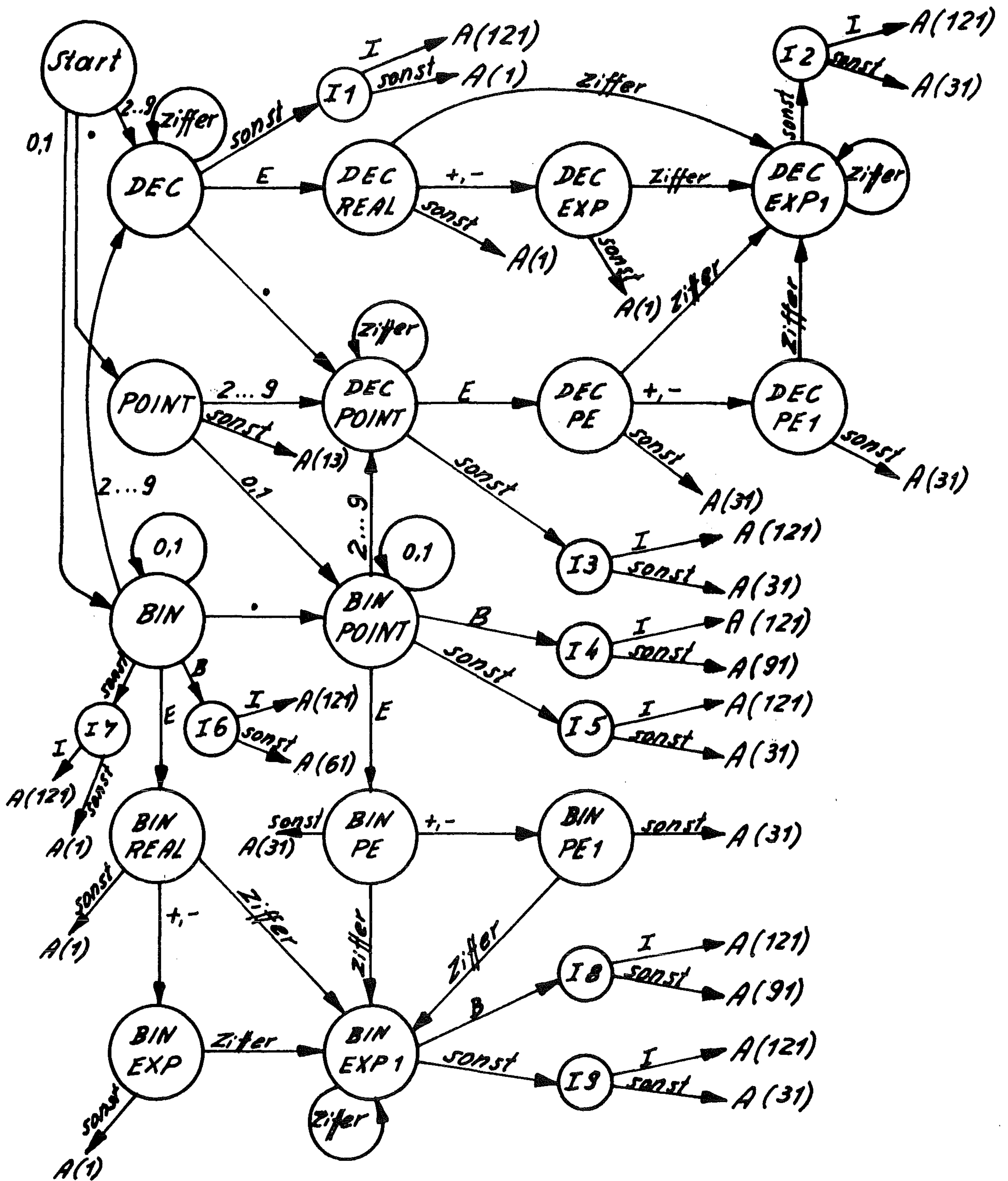
A (i) bedeutet Ausgabe : Symboltyp = i




Buchstabe
LETTER
sonst
A (2)
Buchstabe, Ziffer, -
*
Stern
*
A(5)
sonst
A(4)
/
SLASH
*
COM
*
ENDC
/
A(46)
sonst
sonst
*
sonst
A (34)
>
GT
=
A (96)
sonst
A (6)
<
LT
=
A (126)
sonst
A (36)
=
A (66)
¬
NOT
>
A (156)
<
A (186)
=
A (216)
sonst
A (7)


&
A (9)
|
OR
|
A (10)
sonst
A (8)
)
A (12)
;
A (45)
,
A (14)
:
A (75)
Leerzeichen
Blank
sonst
A (16)
Leerzeichen
%
A (105)
–
A (165)
End of File
A (135)
sonst
A (15)

Unterautomat für arithmetische Konstanten.



Anhang F: REMAC-Sprachspezifikation

1. Kontrollraum

```
CONTROLROOM  [GEOMETRY] [{  CYLINDRICAL  }]   [,]
                         { →PLANE        }

             [ SIZE  [{R}] r   [{Z}]   z ]   ;
                       {X}       {Y}
```

r,z Kontrollraumgröße in Längeneinheiten
Reihenfolge von GEOMETRY, SIZE beliebig

CYLINDRICAL:

z

r

Z

R

PLANE:

z

r

Y

X

2. Zellendaten

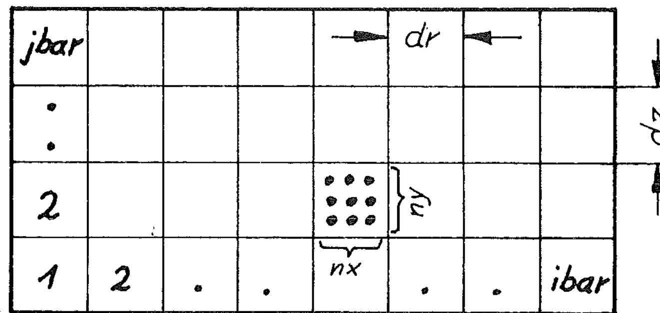
```
CELL         [ SIZE   [{R}] dr   [{Z}]   dz ]   [,]
                        {X}        {Y}

             [ NUMBER [{R}] ibar  [{Z}] jbar ]   [,]
                        {X}         {Y}

             [[LIQUID] PARTICLES [{R}] nx   [{Z}]  ny ] ;
                                   {X}        {Y}
```

dr, dz	Zellengröße
ibar, jbar	Anzahl Zellen in R oder Z - Richtung
nx, ny	Anzahl der Partikel pro Zelle in der Flüssigkeit in R- oder Z-Richtung.

Entweder r,z,ibar,jbar müssen angegeben werden, dann ist dr = r/ibar, dz = z/jbar, oder dr,dz,ibar,jbar werden angegeben, dann ist r = dr * ibar, z = dz * jbar. Sonst Fehler.



3. Hindernisse

OBSTACLE FROM [CELL] i_1 TO [CELL] j_1
[WITH] HEIGHT h_1 [CELLS] [,]
[FROM [CELL]i_2 TO [CELL] j_2
[WITH] HEIGHT h_2 [CELLS] [,]
[FROM [CELL] i_3 TO [CELL] j_3
[WITH] HEIGHT h_3 [CELLS]]] ;

i_k, j_k linke und rechte Begrenzung in Anzahl Maschen , k = 1, ..3

h_k obere Begrenzung in Anzahl Maschen, k = 1, .. 3

Nicht angegebene Hindernisse sind nicht vorhanden.

4. Zufluß

INFLOW { BETWEEN | FROM } [CELL] j_1 { AND | TO } [CELL] j_2 [,]

[WITH] VELOCITY ul ;

j_1, j_2	Untere und obere Begrenzung der Zuflußöffnung in Anzahl Maschen
ul	Zuflußgeschwindigkeit

5. Abfluß

OUTFLOW { BETWEEN | FROM } [CELL] j_3 { AND | TO } [CELL] j_4 [,]

[WITH] VELOCITY ur ;

j_3, j_4, u_r	entsprechend wie bei INFLOW

6. Flüssigkeit

```
LIQUID      [ DOMAIN ]    expr (R,Z)   [,]  [VISCOSITY   nu] ;
  :
UO = f1 (R,Z);
  :
VO = f2 (R,Z);
  :
LIQUID END  [,]  [VISCOSITY    nu ]  ;
```

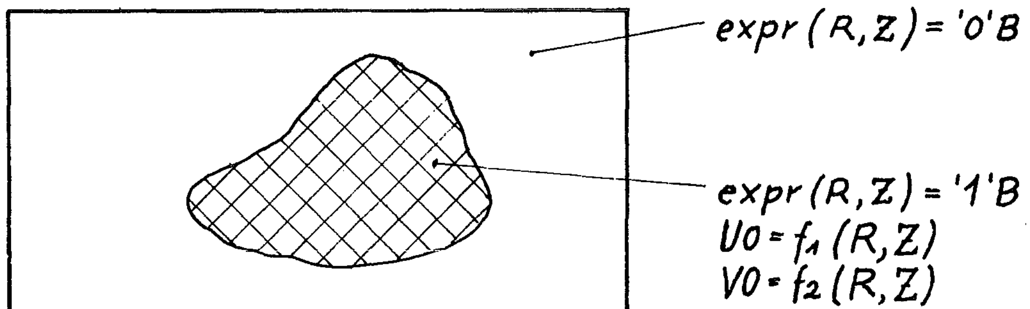
expr (R,Z): Ausdruck, der in Abhängigkeit von R und Z wahr ('1'B) ist, falls am Ort (R,Z) Flüssigkeit vorhanden ist, falsch ('0'B) sonst.

f_1, f_2 (R,Z): Funktionen f r die Anfangsgeschwindigkeiten der Flüssigkeit in R- und Z- Richtung.

nu: Kinematische Zähigkeit der Flüssigkeit, kann bei LIQUID DOMAIN oder LIQUID END angegeben werden.

Beispiel:

```
        LIQUID DOMAIN L, VIS 1.E-6; DCL L BIT INIT('0'B);
        IF R< 3  GOTO LAB1;
        IF R> 5  GOTO LAB1;
        IF Z< 4 THEN IF  Z> 3 THEN DO; L = '1'B; GOTO LAB1; END;
        IF Z>10 THEN IF SQRT (Z-3)> 3 THEN L='1'B;
LAB1:   UO = 3. * R + Z;
        VO = 1. E-2;
        LIQUID END;
```



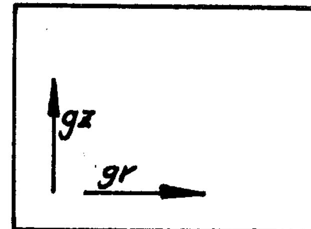
7. Haftung der Flüssigkeit an der Wand

```
                [BOTTOM[{ FREE }]] [,] [RIGHT [{ FREE }]]  [,]
                [       {→NO   }]            {→NO   }
                [TOP    [{ FREE }]] [,] [LEFT  [{ FREE }]]
[WALL] SLIP  {           {→NO   }              {→NO   }      }      ;
                 ALL    [{ FREE }]
                         {→NO   }
```

Haftung (Noslip) oder nicht (Freeslip) der Flüssigkeit an der oberen rechten, unteren oder linken Wand des Kontrollraumes oder an allen Wänden.

8. Schwerkraft

```
GRAVITY      [ [{R}]    gr] [,] [ [{Z}]    gz] ;
                {X}                 {Y}
```



Schwerkraft in R- oder Z- Richtung

9. Zeiten

TIME [START t_o] [,] [END twfin] [,] [STEP dt] ;

to = Anfangszeit der Rechnung

twfin = Endzeit der Rechnung

dt = Zeitschritt (bei Zeitschrittautomatik erster Zeitschritt)

10. Numerische Parameter

```
NUMERICAL    [PARAMETERS ] [OVERRELAXATION  α ]   [, ]
             [ACCURACY       eps    ] [,]
             [STABILITY      safety ] [,]
             [TIMER  [{→ON }]]               ;
                      { OFF}
```

α: Überrelaxtionsparameter
eps: Konvergenzschranke
safety: Stabilitätsparameter
TIMER ON: Zeitschrittautomatik ein

11. Start der Rechnung

```
SOLVE         [UNTIL twfin] [,] [MAXIMUM [NUMBER]
              [OF] [PARTICLES] [=]   max  ] ;
```

twfin: Endzeit (auch in TIME angebbar).

max: Geschätzte Anzahlt Partikel

12. Zurücksetzen der Parameter

```
RESET ;
```

13. Druckausgabe

```
PRINT         [EVERY   { twprt                  } ]  [,]
                       { cyclprt       CYCLES   }

              [START  [TIME]   tprtstart      ]  [,]
              [END    [TIME]   tprtend        ]  [,]
              [INPUT] ;
```

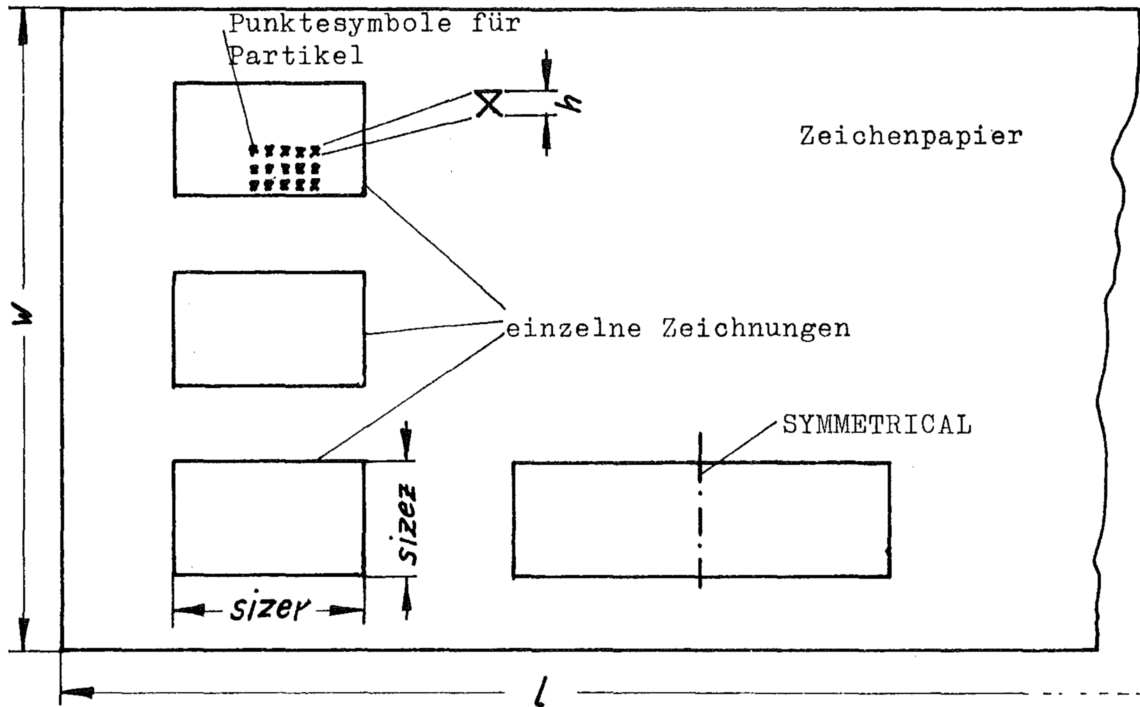
twprt: Zeit zwischen Ausdrucken der Werte
cyclprt: Zyklen zwischen Ausdrucken der Werte
tprtstart: Erste Zeit eines Ausdrucks
tprtend: Letzte Zeit eines Ausdrucks
INPUT: Liste der Eingabeparameter

14. Zeichenangabe

```
                   ┌PRINTER                                  ┐
PLOT       [DEVICE {                                         } ]
                   └PLOTTER [LENGTH l ] [WIDTH  w ]          ┘
                 ┌NORMAL                                     ┐
           [,][FORMAT SIZE [{R/X}] sizer  [{Z/Y}]   sizez      ]
                 │FACTORS  { [{R/X}] fr   [{Z/Y}]  fz }       │
                 └          ALL  fa                          ┘

           [,] [POINTS  [HEIGHT h] [,] [SYMBOL  s ] ]   [,]
           [EVERY {dtwplt                 }      [, ]
                  {cyclplt       CYCLES   }
           [START  [TIME]  twplt   ]     [, ]
           [END    [TIME]  tpltend ]     [, ]
           [SYMMETRICAL]
```

l,w :	Papierlänge- und -breite
sizer, siz	Größe einer einzelnen Zeichnung in cm
fr, fz, fa:	Skalierungsfaktor in R-, Z- oder beide Richtungen: cm Papier pro Längeneinheit
h:	Höhe der Punktesymbole
s:	Symbolnummer der Punktesymbole
dtwplt:	Zeit zwischen Zeichnungen
cyclplt:	Schritte zwischen Zeichnungen
twplt:	Zeit der ersten Zeichnung
tpltend:	Zeit der letzten Zeichnung
SYMMETRICAL:	Bei Zylinderkoordinaten wird auch die an der Symmetrieachse gespiegelte Hälfte gezeichnet.


Punktesymbole für
Partikel
h
Zeichenpapier
w
einzelne Zeichnungen
SYMMETRICAL
sizez
sizer
l

Literatur

/1/ Schlechtendahl, E.G. :
Programmiersprachen im CAD-Bereich, CAD-Mitteilungen 1/1973,
GfK, Karlsruhe, 1973

/2/ Schlechtendahl, E.G.; Enderle, G. :
What can we learn from practical application of ICES for the design of an integrated CAD system nucleus? Proc. Colloque international sur les systemès intégrés en genie civil (CEPOC), Université de Liège, Belgique, 1972

/3/ Lopez, L.A. :
POLO, A Supervisor for Integrated Systems Development, Proc. Colloque international sur les systèmes intégrés en genie civil (CEPOC), Université de Liège, Belgique, 1972

/4/ Audoux,M.; Katz, F.; Olbrich, W.; Schlechtendahl, E.G. :
SEDAP - An Integrated System for Experimental Data Processing.
KFK 1594, Jan. 1973

/5/ Ladisch, R. :
Umstellung des Programms SEDAP auf interaktive Betriebsweise.
Diplomarbeit, Universität Karlsruhe, 1973

/6/ Logcher, R.D.; Connor, J.J.; Nelson, M.F. :
ICES STRUDL II, Engineering User's Manual, Vol.1-3, MIT,
R68-91, R70-77, R70-35, 1968-1971

/7/ McDonnell - ECI:
ICES STRUDL DYNAL User's Manual, McDonnell Douglas Automation Company, May 1974

/8/ Daniels, R.L.; Hall, E.J. :
ICES PROJECT I -General Description, First Edition, MIT, R68-19, March 1968

/9/ Enderle, G.; Schlechtendahl, E.G.; Schumann, U.; Schuster, R. :
Design Principles of the GRAPHIC system, KFK 1722, 1973

/10/ Schuster,R.; Enderle, G.; Leinemann, K.; Schlechtendahl, E.G.; Schnauder, H.; Schumann, U. :
Sprach- und Datenstruktur des Systems GRAPHIC, Gesellschaft für Informatik, 2. Jahrestagung, Karlsruhe, 2.-4.Oktober 1972, Lecture Notes in Economics and Mathmatical Systems, Springer Verlag

/11/ Enderle, G.; Leinemann, K.; Schlechtendahl, E.G.; Schnauder, H.; Schumann, U.; Schuster, R. :
Fähigkeiten und Implementierung der GRAPHIC-Sprache, KFK-Ext.8/72-7

/12/ Enderle, G.; Schlechtendahl, E.G.; Schuster, R. :
Anleitung zur Erweiterung des GRAPHIC-Systems um neue Funktionen, KFK-Ext. 8/74-1

/13/ Schnauder, H. :
GRAPHIC-Handbuch, KFK-Ext. 8/73-2, Dezember 1973

/14/ EXAPT, Möglichkeiten und Anwendungen der automatischen Programmierung für NC-Maschinen, Carl Hanser Verlag, München, 1969

/15/ IBM Corporation :
General Purpose Simulation System, Introductory User's Manual, IBM Form SH20-0693, October 1969

/16/ Faber, D.J.; Griswold, R.E.; Polansky, I.P. :
The SNOBOL3 Programming Language, Bell System Tech. Journal 45, 1966

/17/ ECAP - Electronic Circuit Analysis Program User's Manual, IBM, 20-0170-2

/18/ Soop, K. :
The design and use of a PL/1-based graphic programming language, ONLINE 72, Conf. Proc., 1972

/19/ Smith, D.N. :
GPL/1 - A PL/1-extension for computer graphics, Proc. Spring Joint Computer Conference, 1971

/20/ Tobey, R.; Baker, J.; Grews, R.; Marks, P.; Victor, K. :
PL/1-FORMAC Interpreter User's Manual, IBM Contributed Program Library 306D, 03.3.004, 1967

/21/ Kampe, G. :
SIMSCRIPT, Vieweg-Verlag, Braunschweig, 1971

/22/ CSMP - Continuous System Modeling Program, Application Programmer User's Manual, IBM, H20-0367

/23/ Carlson, C.R. :
The study of the appliability of the Vienna definition language to the specification of the semantics of the next event simulation concept, Tech. Report no. 59, University of Iowa, Department of Mathematics, August 1972

/24/ Hurwitz, A.; Citron, I.P.; Yeaton, I.B. :
Graphic Additions to FORTRAN, Proc. Spring Joint Computer Conference, 1967

/25/ Brown, P.J. :
A survey of macro processors, Annual Review in Automatic Programming, Vol. 6, part 2, 1969

/26/ Schlechtendahl, E.G.; Schumann, U. (Ed.):
Erfahrungen mit dem Programmsystem ICES bei ingenieurtechnischen Anwendungen, KFK 1586, 1972

/27/ Leinemann, K.; Schumann, U. :
Ein Beitrag zu Grundlagen des rechnergestützten Entwurfs und einer Konstruktionssprache, KFK-Ext. 8/72-5, Februar 1973

/28/ Leinemann, K.; Schumann, U. :
KOSPRA - Entwurf einer Konstruktionssprache zur Beschreibung der Geometrie technischer Objekte, KFK-Ext. 8/72-6, Februar 1973

/29/ Roos, D. (Ed.):
ICES System - General Description, MIT, R67-49, 1967

/30/ Roos, D. :
ICES System Design, MIT-Press, 2nd ed., 1967

/31/ Alcock, Shearing and Partners :
GENESYS Reference Manual, The GENESYS Center, Loughborough, 1971

/32/ Siemens System 4004,
Informationssystem Technik, IST, Programmierhandbuch, 1973
und Kommandosprache, 1973

/33/ Pahl, J.P. :
ISB - Informationssystem für das Bauwesen, BMBW-FB DV 72-09,
Dezember 1972

/34/ Lopez, L.A. :
POLO, Problem Oriented Language Organizer, Computers and
Structures 2, 1972

/35/ Roose, D.T. :
The AED Approach to Generalized Computer Aided Design,
Proc. ACM 2nd Annual Conf., 1967

/36/ Deprez, G. :
SYSFAP: Description Générale de Systèmes, Colloque International
sur les Systèmes Intégrés, (CEPOC), Liège, Belgique, 1972

/37/ Enderle, G.; Katz, F.; Leinemann, K.; Schlechtendahl, E.G.;
Schumann, U.; Schnauder, H.; Schuster, R. :
Erster REGENT - Halbjahresbericht, Oktober 71 - März 72,
KFK-Ext. 8/72-2, 1972

/38/ Enderle, G.; Leinemann, K.; Schlechtendahl, E.G.;
Schnauder, H.; Schumann, U.; Schuster, R. :
Zweiter REGENT - Halbjahresbericht, April 72 - September 72,
KFK-Ext. 8/72-41, 1972

/39/ Enderle, G.; Leinemann, K.; Olbrich, W.; Schlechtendahl, E.G.;
Schumann, U.; Schuster, R. :
Dritter REGENT-Halbjahresbereicht, Oktober 72 - März 73,
KFK-Ext. 8/73-1, 1973

/40/ Schlechtendahl, E.G. (Ed.):
Vierter REGENT-Halbjahresbericht, April 73 - September 73,
KFK-Ext. 8/73-4, 1973

/41/ Schlechtendahl, E.G. (Ed.):
Fünfter REGENT-Halbjahresbericht, Oktober 73 - März 74,
KFK-Ext. 8/74-3, 1974

/42/ Schlechtendahl, E.G. (Ed.):
Sechster REGENT-Halbjahresbericht, April 74- September 74,
KFK-Ext. 8/74-5, 1974

/43/ Enderle, G.; Schlechtendahl, E.G. :
The Design of the Integrated CAD-System REGENT, Proc. Workshop
on General Purpose CAD Systems, Toulouse, December 1974

/44/ Enderle, G.; Schlechtendahl, E.G. :
The CAD-System REGENT, 12th Design Automation Conf.,
Boston, Mass., June 1975

/45/ Schlechtendahl, E.G.; Enderle, G.; Leinemann, K.;
Schumann, U.; Schuster, R. :
Design Criteria for REGENT - a CAD System for Mechanical and
Chemical Engineers. Int. Congress on the Use of Electronic
Computers in Chemical Engineering, Paris, April 1973

/46/ Becker, J.; Woit, J.; Jajonc, H. :
Einsatz der elektronischen Datenverarbeitung in der Bundes-
republik 1970, KFK-Ext. 2/71-2, Dezember 1971

/47/ IBM Corp. :
OS PL/1 Checkout and Optimizing Compilers: Language Reference
Manual, SC33-0009-2

/48/ IBM Corp. :
OS PL/1 Optimizing Compiler: Programmers's Guide, SC33-0006

/49/ Nelson, D.A. :
Whatever happended to PL/1. The Diebold Research Program, Conf. Proc. Meeting XXIX, Genova, 1973, Doc. No. EC29

/50/ Siemens :
PBS 4004/35-100, PL/1 Language Reference Manual

/51/ ECMA/ANSI :
Basis/1-9, ECMA.TC10/ANSI.X3J1, PL/1, BASIS/1, 1972

/52/ Mangelsdorf, R. :
Zeitvergleiche FORTRAN-PL/1 bei Matrixmultiplikationen, interne Notiz

/53/ IBM Corp. :
OS-PL/1 Optimizing Compiler: Execution Logic, SC33-0025

/54/ IBM Corp. :
OS/360 FORTRAN IV Language, GC28-6515

/55/ IBM Corp. :
OS/360 FORTRAN IV (G and H) Programmer's Guide, GC28-6817

/56/ Anderson, R.H.; Farber, D.J. :
Extensions of the PL/1 Language for interactive computer graphics, memorandum RM-6028-ARPA

/57/ Dodd, G.G. :
APL - A language for associative data handling in PL/1, Proc. Fall Joint Computer Conference, 1966

/58/ Bayes, A. :
PLITRAN - A generalized PL/1 Macro Facility, Proc. 4th Australian Computer Conference, Adelaide, 1969

/59/ IBM Corp. :
OS PL/1 Checkout Compiler: Execution Logic, SC33-0032

/60/ Cuff, R.N. :
A Conversational Compiler for Full PL/1, The Computer Journal 15, 1973

/61/ IBM Corp. :
OS Assembler Language, GC 28-6514, 1972, pp. 61 - 106

/62/ Naur, P. (Ed.):
Revised Report on the Algorithmic Language ALGOL 60, Comm. ACM 6, 1973

/63/ Schumann, U.; Olbrich, W.; Schlechtendahl, E.G. :
RMM - REGENT Module Management, KFK, 1975, wird veröffentlicht

/64/ Leinemann, K. :
Dynamische Datenfelder des REGENT-Systemkerns, KFK, 1975, wird veröffentlicht

/65/ Leinemann, K.; Bechler, K.H. :
Die Datenbankverwaltung des REGENT-Systems, KFK, 1975, wird veröffentlicht

/66/ IBM Corp. :
OS/360 Time Sharing Option, Command Language Reference, GC 28-6732

/67/ Amsden, A.A.; Harlow, F.H. :
The SMAC Method: A Numerical Technique for Calculating Incompressible Fluid Flows, LA-4370, 1970

/68/ Schultheiß, G.F.; Enderle, G. :
REMAC - Ein REGENT-Subsystem zur Berechnung inkompressibler Strömungen, KFK-Ext., 1975, wird veröffentlicht

/69/ Schuster, R. :
GIPSY - Ein REGENT-Subsystem zur Behandlung zwei- und dreidimensionaler graphischer Objekte, KFK, wird veröffentlicht

/70/ Doll, F. :
Erzeugung der Problemsprache FLOWCHART mit Hilfe von PLS, Studienarbeit, Fachhochschule Pforzheim, 1975

/71/ IBM Corp. :
System 360 FLOWCHART User's Manual, IBM-Form H20-0293-2

/72/ Amsden, A.A.; Hirt, C.W. :
Yaqui: An Arbitrary Lagrangian-Eulerian Computer Program for Fluid Flow at All Speeds, Los Alamos Scientific Lab., LA-5100, 1973

/73/ Diebold, M. :
Entwicklung einer problemorientierten Eingabesprache für REGENT-SEDAP, Diplomarbeit, Universität Karlsruhe, 1975

/74/ Enzmann, J.; Hohn, U. :
Benutzerhandbuch der Anlagen IBM/360-65 und IBM/370-168, GfK, ADI

/75/ Calcomp GmbH., Düsseldorf :
Programme für Calcomp-Plotter der Serie 500, 600 und 700

/76/ Gries, D. :
Compiler Construction for Digital Computers, John Wiley and Sons, New York, 1971

/77/ IBM Corp. :
IBM System/370, Principles of Operation, GA 22-7000

/78/ Hofmann, F. :
Formale Behandlung syntaktischer Funktionen, Siemens Forschungs- und Entwicklungsberichte 3, Nr. 6, 1974

/79/ IBM Corp. :
OS/360 Supervisor Services and Macro Instructions, GC28-6646

/80/ Gross, M.; Lentin, A. :
Introduction to Formal Grammars, Springer Verlag Berlin, 1970

/81/ Urschler, G. :
Concrete Syntax of PL/1, IBM Lab. Vienna, TR 25.096, 1969

/82/ IBM Corp. :
OS/360 Utilities, GC 28-6586

/83/ IBM Corp. :
OS/360 System Programmer's Guide, C28-6550

/84/ IBM Corp. :
OS/360 Linkage Editor and Loader, C28-6538