

Lesch

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

**HERAUSGEBER: H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees, H. Niemann,
D. Seitzer, B. Schmidt, H. J. Schneider, H. Wedekind**

PORTABLE CODEGENERIERUNG IN ÜBERSETZERN FÜR PROZESS-PROGRAMMIERSPRACHEN

Kuno Pelz

DISSERTATION

BAND 15 · NUMMER 13 · ERLANGEN · OKTOBER 1982

ISSN 0344 - 3515

ARBEITSBERICHTE

DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

**HERAUSGEBER: H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees, H. Niemann,
D. Seitzer, B. Schmidt, H. J. Schneider, H. Wedekind**

PORTABLE CODEGENERIERUNG IN ÜBERSETZERN FÜR PROZESS-PROGRAMMIERSPRACHEN

Kuno Pelz

DISSERTATION

BAND 15 · NUMMER 13 · ERLANGEN · OKTOBER 1982

**Institut für Mathematische Maschinen
und Datenverarbeitung (Informatik)**

**Martensstraße 3
8520 Erlangen**

**Alle Rechte bleiben vorbehalten.
Wiedergabe auch auszugsweise nur mit
Genehmigung der Herausgeberschaft.**

ISSN 0344 - 3515

Herr Prof. Dr. H.J. Schneider hat mir nicht nur die Durchführung dieser Arbeit ermöglicht, sondern war mir auch immer wieder mit seinem fachlichen Rat, viel Geduld und Ermutigung eine große Hilfe, für die ich mich an dieser Stelle besonders herzlich bedanke.

Meinen Kollegen, besonders Herrn Dr.Ing. H. Göttler, danke ich sehr für ihre verständnisvolle Diskussions- und Hilfsbereitschaft.

Übersicht

Die Arbeit stellt ein Konzept vor, das den Aufwand zur Erstellung von Codegeneratoren so stark reduziert, daß von portabler Codegenerierung gesprochen werden kann. Sie ist als Beitrag zur Portabilität von Compilern gedacht, deren Zielkreis kleine bis mittlere Prozeßrechner sind. Dabei wird angestrebt, die Codegenerierung so in das Übersetzungssystem einzubetten, daß zur Implementierung nur noch möglichst einfache Mechanismen notwendig werden. Dieser Einfachheit wird höchste Priorität eingeräumt; die Formalisierung der Codegenerierung und Fragen der Codeoptimierung stehen dagegen im Hintergrund.

Ausgangspunkt ist die übliche Aufteilung des Übersetzers in Compiler-Oberteil und Codegenerator. Die Zwischensprache wird als Maschinencode einer virtuellen Maschine so entworfen, daß ihre Operationen eine Zerlegung des Codegenerators in sehr kleine Elemente erlaubt, die dann einzeln verwirklicht werden können. Es wird ein Codegenerator-Gerüst spezifiziert, das überall da unausgefüllt bleibt, wo zur Darstellung von Daten oder Befehlen die Kenntnis der realen Maschine notwendig ist. Diese Lücken werden im Dialog mit einem Generatorprogramm geschlossen.

Die Bindecodestruktur wird so festgelegt, daß der Binder nahezu zielrechnerunabhängig verwirklicht werden kann.

Die Arbeit wurde im Rahmen des Forschungsvorhabens P4.2/3 des Bundesministeriums für Forschung und Technologie, Projekt Prozeßlenkung durch Datenverarbeitung, gefördert.

Inhaltsverzeichnis

	Seite
1 Einleitung	1
2 Der Codegenerator im Übersetzungsablauf	4
3 Die Zwischensprache CODE	9
3.1 Entwurfskriterien	10
3.2 Die abstrakte CODE-Maschine	12
3.2.1 Architektur	13
3.2.2 Datentypen	20
3.2.3 Adressier-Mechanismen	25
3.2.4 Befehlssatz	27
3.2.4.1 Die Pseudobefehle	28
3.2.4.2 Platzreservierung	32
3.2.4.3 Die ausführbaren Befehle	33
3.3 Eine Konkretisierung der CODE-Maschine	40
3.3.1 Pseudobefehle	42
3.3.2 Platzreservierung	51
3.3.3 Ausführbare Befehle	62
3.3.3.1 Grobeinteilung	62
3.3.3.2 Umgruppierung und Redundanzerrhöhung	67
3.3.3.3 Übersetzung von Einwortbefehlen	70
3.3.3.4 Bearbeitung von Mehrwortbefehlen	75
4 Das Generatorprogramm	80
4.1 "Primitive" Generierparameter	80
4.1.1 Numerische Parameter	80
4.1.2 Konversionsroutinen	81
4.1.3 Daten-Codegeneratoren und Speicherregister	82
4.2 Zielcodesequenz-Muster	84
4.2.1 Sequenzaufbau und Einwortroutinen-Aufrufe	84
4.2.2 Sequenzelement-Arten	85
4.3 Nachtrag zum Binder	87
5 Zusammenfassung	87
Literatur	89
Anhänge:	
1 Erläuterungen zur Syntaxnotation	
2 Syntax	
3 Abkürzungen	
4 Abbildungsverzeichnis	
5 Stichwortverzeichnis	

1 Einleitung

Im Frühjahr 1973 erschien der erste Vorschlag zur Definition der Prozeßprogrammiersprache PEARL /EPDV73/. Eine der ersten Implementationen war "ASME-1" /APIE76/. Es wurde der Versuch unternommen, das Übersetzungssystem portabel zu gestalten, da für mehrere verschiedene Zielrechner gleichzeitig implementiert wurde. Die Einsparungen, die mit relativ einfachen Mitteln erreicht wurden, legten es nahe, in der eingeschlagenen Richtung weiter zu arbeiten.

Ausgangspunkt war die Aufteilung des Übersetzers in einen zielrechnerunabhängigen Compileroberteil (COT) und die einzeln erstellten Codegeneratoren, die Assemblercode für das jeweilige Zielsystem erzeugten. Als Schnittstelle diente die Zwischensprache CIMIC/1 /CIME76/. Ebenso wie die Codegeneratoren entstand auch der größte Teil des Laufzeitsystems rechnerabhängig "in Handarbeit". Nach Abschluß der Implementierung wurde daher angestrebt, die Portabilität der Übersetzerkomponenten

Prozeßverwaltung,
zeichenorientierte und graphische Ein-/Ausgabe,
arithmetische Bibliotheksroutinen und
Codegenerator (CG)

zu erhöhen /VEPA77/.

Seit etwa 1975 ist allgemein wachsendes Interesse an einer einfacheren Erstellung von Codegeneratoren zu bemerken: In /FRAS77/ wird eine Rechnerbeschreibungssprache /BENE71/ eingesetzt, die an der Hardware orientiert und ziemlich komplex ist, und in /GRAH80/ wird die Technik der Tabellensteuerung vorgeschlagen. Es werden aber auch eigenständige Modelle entworfen, deren abstrakte Codegenerierungsfunktionen eine weitgehende Automatisierung der Erstellung ermöglichen sollen (/CATT77/, /CATT78/, /TERM78/).

Im Unterschied zu diesen Arbeiten wird hier weder versucht, den Automatisierungsgrad hochzutreiben, noch soll die Codegenerierung allgemein formalisiert werden, wie in /PETE79/. Es ist beabsichtigt, ein Werkzeug zu konstruieren, das eine sehr vereinfachte Erstellung von Codegeneratoren erlaubt (erst der Gebrauch dieses Werkzeugs könnte dann als Umgang mit einem Formalismus verstanden werden).. Insbesondere sollen die "Architekturentscheidungen" beim Schritt zur realen Maschine immer noch vom Implementator getroffen und nicht automatisch gefällt werden. Der Effizienz der zu übersetzenden Programme wird nur niedere Priorität eingeräumt.

Wichtigstes Ziel ist ein Erstellungssystem, dessen Realisierung und Gebrauch als leicht überschau- und beherrschbar gelten kann. Seine Implementierung sollte nicht nur auf Großsystemen der Rechenzentrumsklasse möglich sein; besonders die erzeugten Codegeneratoren sollen auch auf "kleineren" Rechnern ablauffähig sein, damit eine gewisse Unabhängigkeit von Großrechnern und Cross-Compilern erreicht wird.

Eine erste Charakterisierung des Kreises der Zielrechner sei durch eine schlichte Aufzählung gegeben: ins Auge gefaßt wird zunächst eine Architektur-Klasse, die durch

Intel 8080,
Zilog Z80,
Motorola 6800,
MCS 6500,
General Instruments CP-1600,
Texas Instruments 9900 usw.

aufgespannt wird. In dieses Spektrum können auch ältere Prozeßrechner eingeordnet werden, deren Aufzählung hier überraschen mag:

Siemens-Prozeßrechner 306 (24 Bit Wortlänge),
PDP-8 (12 Bit), PDP-15 (18 Bit),
AEG 60xx (24 Bit) und andere.

Sie gehörten einer anderen Preisklasse an und stellten noch das Zielspektrum der ASME-1-Implementation dar. Von ihrer Architektur und Leistungsfähigkeit her gesehen, sind sie aber Mitglieder der oben umrissenen Klasse. Für die meisten Systeme dieser Art wird es aus Aufwandsgründen keine Spezialimplementation einer Prozeßprogrammiersprache geben; portable Übersetzer könnten diese Versorgungslücke wenigstens teilweise schließen.

Als ein weiterer Anwendungsbereich kämen "Vorab-Implementierungen" für Rechner-Neuentwicklungen in Frage, wo die Effizienz zunächst im Hintergrund steht. Ähnlich kann die Situation bei inhomogenen Prozeßrechner-Verbundsystemen gesehen werden, in denen unterschiedliche Rechner mit den gleichen Sprachen programmiert werden sollen: hochportable Compiler würden bei Erweiterungen des Verbunds den Aufwand für die Anpassung des Übersetzer-Systems niedrig halten.

Portable Codegeneratoren würden es ermöglichen, zielrechnerunabhängig programmierte Teile der Bibliotheksroutrinen einfach durch Übersetzung für neue Zielmaschinen verfügbar zu machen.

Zwei Randbedingungen für den sinnvollen Einsatz "billiger" Codegeneratoren seien aber schon hier vorweggenommen: Zum einen ist leicht vorherzusehen, daß die Effizienz der übersetzten Programme leiden wird. Für Anwendungen, in denen um jedes Byte gerungen wird (z.B. eingebettete Systeme), würden Optimierungen womöglich mehr Aufwand erfordern, als bei den übrigen Teilen der Implementation gespart werden kann. Immerhin sollten im Hinblick auf eventuelle Sicherheitsanforderungen auch solche Konstellationen nicht ausgeschlossen werden. Insgesamt spielen aber in einer Zeit, in der Hobbyrechner oft einen Hauptspeicher-Ausbau von 64 K Byte erreichen und nicht selten überschreiten, Aspekte der Speicher-Effizienz eine Nebenrolle.

Die wichtigste Voraussetzung bleibt natürlich, daß nicht nur Compileroberteil und Codegenerator sondern auch die Bestandteile des Laufzeitsystems in ein Portabilitätskonzept mit einbezogen werden. Dazu wäre eine Integration etwa der Arbeiten /RÖS78-2/, /LIND81/, /PRES82/ und des hier entworfenen Verfahrens notwendig, was aber den Rahmen dieser Arbeit sprengen würde.

2. Der Codegenerator im Übersetzungsablauf

Unter Beschränkung auf eine einfache Ablaufstruktur des Compilers (siehe Abb. 2-1) wird bei der Festlegung der Zwischensprache (in Kap. 3) nach dem Janus-Prinzip /POWA73/ vorgegangen: Nach unten blickend wird von einer Klasse von Zielrechnern abstrahiert und die Zwischensprache als Maschinencode bzw. Assembler einer virtuellen Maschine entworfen. Parallel dazu müssen mit Blick nach oben die Belange der Übersetzung problemorientierter Programmiersprachen /SCHN80/ im Auge behalten werden. Verglichen mit CIMIC/1 soll die Zwischensprache wesentlich vereinfacht und die Sprachebene deutlich gesenkt werden, um reale Maschinen mit weniger Aufwand erreichen zu können.

Die Bindecode-Schnittstelle

Als "untere" Schnittstelle solcher Codegeneratoren diene bisher häufig eine Assemblersprache des Zielrechners. Sie macht einerseits die Erzeugung von Zeichenketten notwendig, die der Syntax des Assemblerers genügen, andererseits kann in einem gewissen Rahmen die Namensbuchführung auf ihn abgewälzt werden. Im Hinblick auf das Konzept einer portablen Codegenerierung (CGg) stellt der zweite Umstand keinen nennenswerten Vorteil mehr dar, weil Namensbuch-Mechanismen weitestgehend unabhängig vom Zielrechner realisiert werden können; dagegen würde die jeweilige Generierung der Assemblersyntax neben einer gewissen Mehrarbeit für den Implementator auch eine erhebliche Komplizierung für den Entwurf des Generiersystems bedeuten.

Eine maschinencode-orientierte Bindschnittstelle wird deshalb vorgezogen. Es wird sich zeigen, daß zwar geringfügig zusätzliche Anforderungen an den Binder zu stellen sind, dieser aber nahezu zielrechnerunabhängig realisierbar sein wird. Der Ablauf einer Übersetzung stellt sich nun vergrößert so dar, wie in Abb. 2-2 wiedergegeben.

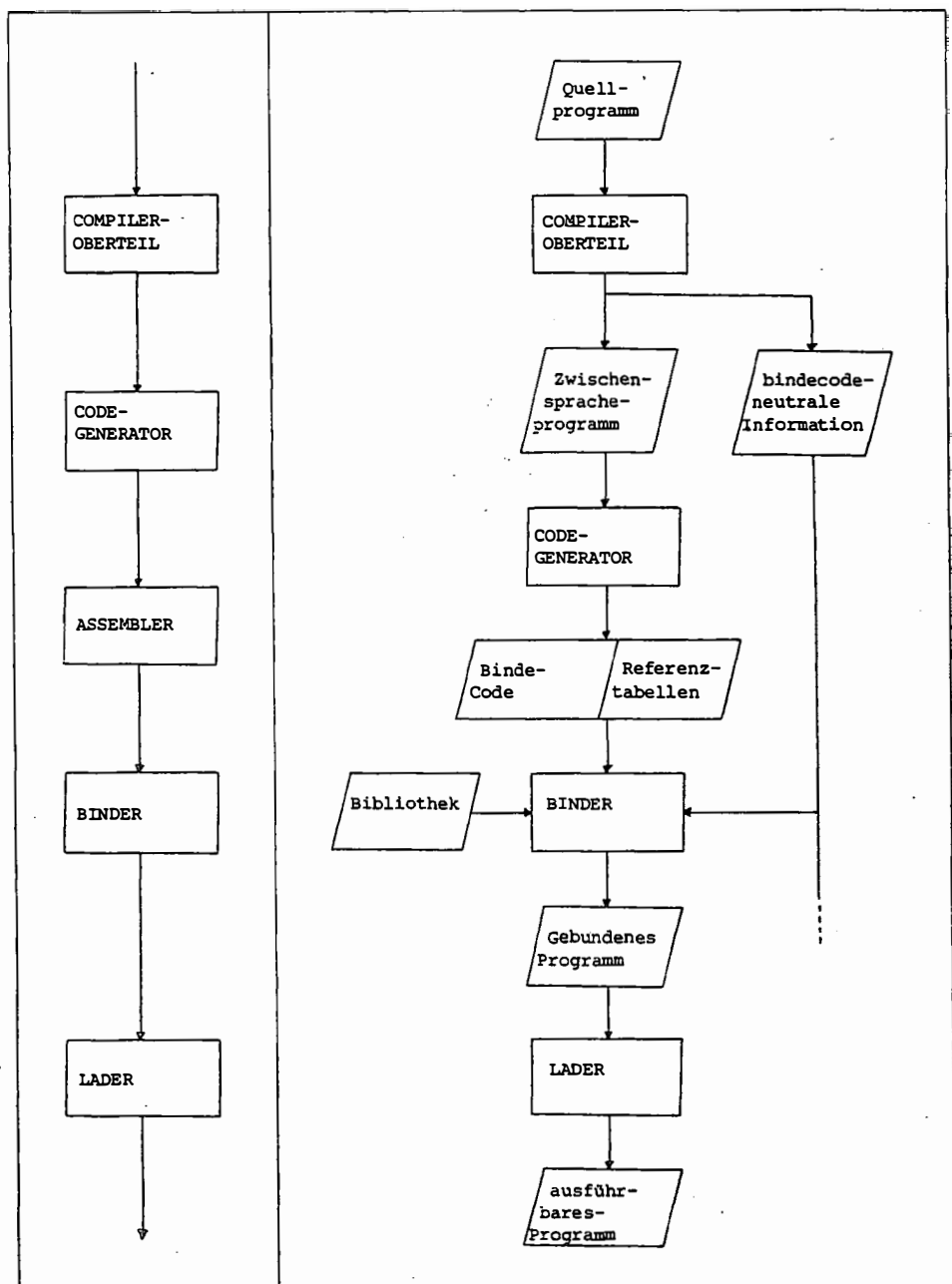


Abb. 2-1: Üblicher
Übersetzer-Ablauf

Abb. 2-2: Geringfügig detaillierter/modifizierter
Übersetzer-Ablauf

Die virtuelle Zwischensprachmaschine

Die zwei wichtigsten Arbeitspunkte werden also die Definition der Zwischensprache und der Entwurf des Generierverfahrens sein. Auf die Kriterien für die Zwischensprach-Festlegung wird in 3.1 eingegangen. Hier sei nur auf einige generelle Aspekte hingewiesen. Zum einen wird das Konzept der "virtuellen Maschine" eine gewisse Rolle spielen: es werden Mechanismen angesprochen, die von der Arbeitsweise realer Rechner her bekannt sind, wobei auch die üblichen Terme verwendet werden. Allerdings wird nicht versucht, eine Formalisierung in Richtung einer Kalkulierbarkeit (im weitesten Sinne) zu erreichen oder auch nur anzunähern; statt dessen sollen die realen Rechner im Blickfeld bleiben. Eine virtuelle Maschine ist hier also eine Art abstrakter de-facto-Standard.

Anzustreben ist eine Zwischensprache, die sehr tief liegt, d.h. die virtuelle Maschine soll deutliche Ähnlichkeit mit realen Rechnern haben.

Überarbeitung der Zwischensprache

Ausgehend von Eigenschaften und Fähigkeiten realer Maschinen werden jene Elemente in die virtuelle Maschine übernommen, die so elementar erscheinen, daß ihre Realisierung auf allen Rechnern des Zielspektrums einfach möglich sein sollte. Dieses Vorgehen hat zur Folge, daß die virtuelle Maschine realen Rechnern zwar sehr nahe stehen, in vieler Hinsicht aber eher den Charakter eines Durchschnitts ihrer Vorbilder haben wird, als den einer Vereinigung, wie die abstrakte Drei-Adreß-Maschine in /LIND81/.

Als zweites wird bei der Neufassung der Zwischensprache darauf geachtet, jene Übersetzungsarbeiten aus der Codegenerierung auszugliedern, die schon auf Zwischensprachebene - also vor der Codegenerierung - ausführbar sind.

Das Generieren

Auf der Grundlage der überarbeiteten Zwischensprache soll ein weitgehend mechanisierter Ablauf der Codegenerator-Erstellung so entworfen werden, daß er durch ein "Generatorprogramm" unterstützt werden kann. Entscheidungen, die bei den meisten oder gar allen Implementierungen gleichartig gefällt werden, sollen vorweggenommen und in ein CG-Gerüst eingebracht werden.

Der Weg zu diesem Generierv erfahren ist ausgesprochen heuristisch. Für eine weitgehend konkretisierte Zwischensprache wird versucht, ein CG-Programm zu entwerfen; an den Stellen, an denen Information über Zielrechner eingeht, muß dieses Programm unvollständig bleiben. Die Lücken definieren sogenannte Generierparameter. Mit Hilfe des Generatorprogramms werden diese Lücken geschlossen. Es wird im wesentlichen die Rolle eines einfachen Edier-Werkzeuges haben. Der unfertige CG wird im weiteren auch Rahmen oder Skelett genannt.

Zusammenfassend:

Die Wegweiser, denen bei der Lösung dieser Aufgabe gefolgt werden soll, tragen die Aufschriften

Üblicher Übersetzungs-Ablauf,
Sprachunabhängigkeit,
Zielrechner-Unabhängigkeit,
Niveausenkung der Zwischensprache und
Vereinfachung der CG-Schnittstellen.

(siehe Abb. 2-3)

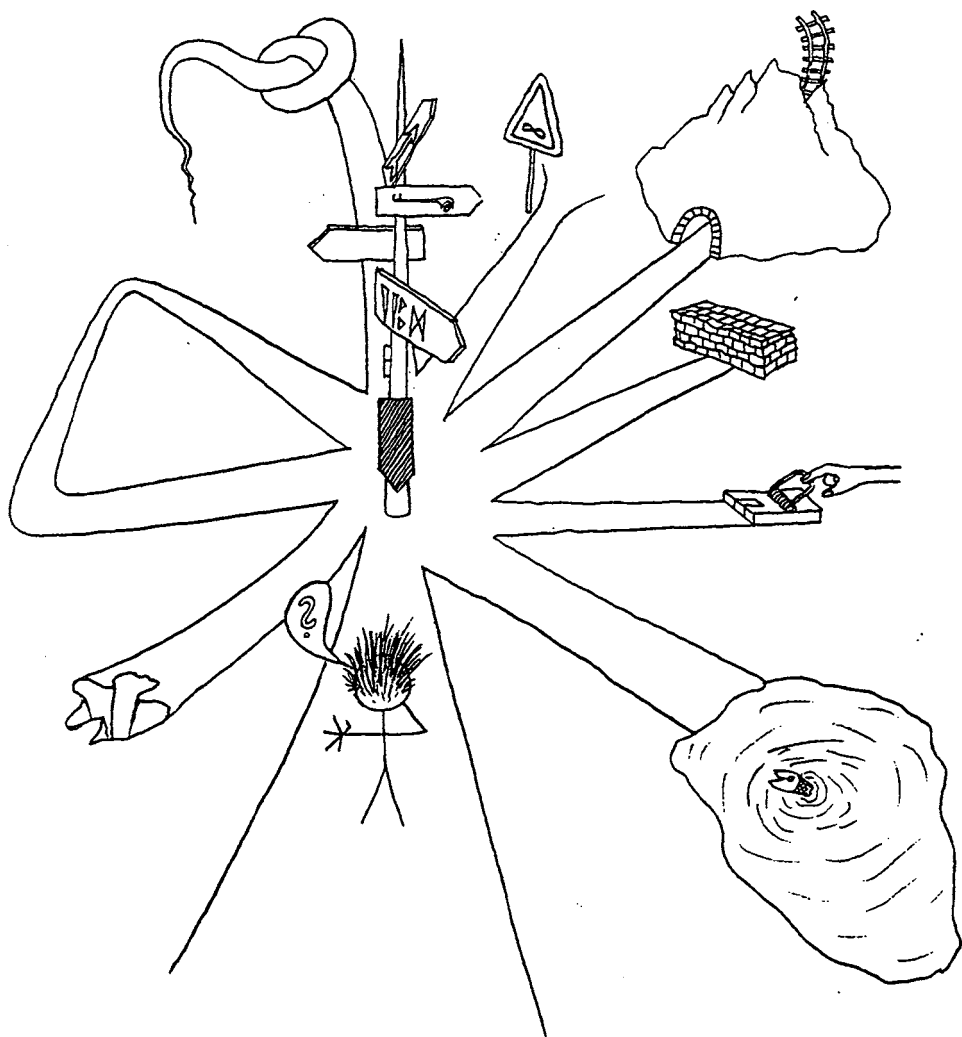


Abb. 2-3: Der Lösungsweg ("Sie können ihn gar nicht verfehlen!")

3. Die Zwischensprache CODE

Die Zwischensprache, die hier beschrieben wird, hat den Namen CODE, d.h. CIMIC-Objekt-Dialekt-Erlangen. Diese Namensgebung ist historisch bedingt: der Entwurf von CODE ist vor dem Hintergrund der diversen CIMIC-Dialekte zu sehen (/CIME76/, /CIMC76/, /CIMP77/). Während diese Zwischensprachen auf einer relativ hohen Ebene stehen, wird hier versucht, die Sprachebene so weit nach unten zu verlegen, daß CODE eher die Rolle eines Objektcodes spielen könnte. Die so entstandene Zwischensprache kann als ein weiterer Dialekt der CIMIC-Familie betrachtet werden.

Analog zum Vorgehen bei den CIMIC-Definitionen wird auch CODE als Assembler bzw. Maschinencode einer virtuellen Maschine beschrieben. Die Begriffe "Zwischensprache CODE" und "virtuelle CODE-Maschine" werden deshalb hier als gleichbedeutend betrachtet (es wird jeweils derjenige Begriff verwendet, der einfachere Formulierungen gestattet). Die Sprachbeschreibung selbst erfolgt in drei Schritten:

Zuerst werden unter der Überschrift "Entwurfskriterien" allgemeine Gesichtspunkte erörtert, die den Entwurf wesentlich beeinflußt haben. Im zweiten Teil werden Architektur-Merkmale, Datentypen, Adreßmodi und Operationen einer "abstrakten" CODE-Maschine festgelegt. Um dabei Implementierungen so wenig wie möglich vorwegzunehmen, kommt für die Definition der Operationen eine abstrakte Syntax zur Anwendung. Im letzten Abschnitt wird daraus eine "konkretisierte" CODE-Maschine abgeleitet. Auch sie ist immer noch eine virtuelle Maschine; zur Beschreibung dient jedoch eine unvollständige konkrete Syntax. Hier werden Implementierungsentscheidungen eingebracht, die zur Spezifikation der realen Codegeneratoren dienen.

Daneben werden noch Einschränkungen gegenüber der abstrakten Maschine vorgenommen, die auf eine weitgehende Vereinfachung von Pilotimplementierungen des Generiersystems zielen.

3.1 Entwurfskriterien

Um eine Auffächerung der Codegeneratoren in unnötig viele Ablaufpfade zu vermeiden, wird angestrebt, die Anzahl der CODE-Befehle niedrig zu halten und die virtuelle Maschine insgesamt aus einfach implementierbaren Elementen aufzubauen, die möglichst häufig direkte Entsprechungen in der realen Prozessorwelt haben sollen. Ein gewisses Maß an Orthogonalität soll dabei allerdings gewahrt bleiben, damit keine Sammlung von Ausnahmefällen entsteht.

Ein wichtiger Punkt bei der Verkleinerung des Befehlssatzes wird sein, alle Übersetzungsarbeiten aus der Codegenerierung auszugliedern, die zielrechnerunabhängig ausführbar sind und den Bindecode nicht beeinflussen. Beispiele: rechnerunabhängige Optimierungen, Bibliotheksverwaltung, sprachspezifische Bindefunktionen oder etwa Export von CODE-Identifikatoren der Unterprogramm-Anfangsadressen; dieser Export ist notwendig, da insbesondere kein Unterschied zwischen Marken einerseits und den Eintrittspunkten von Unterprogrammen, Koroutinen und Tasks andererseits gemacht werden wird. Semaphore, Formatlisten, Dateiverwaltungs-Funktionen usw. werden nicht vorgesehen. Alle solchen Objekte und Operationen werden rechnerunabhängig durch einfachere CODE-Elemente darstellbar sein.

Aber auch zielrechnerabhängige Komplexe, wie etwa Interruptverwaltung oder das Anlegen von Geräteverwaltungs-Blöcken, sollten nicht erst während der Codeerzeugung behandelt werden, sondern im CODE-Programm schon aufgelöst erscheinen. Das Eliminieren solcher Objekte hat zur Folge, daß es eigentlich unerheblich ist, ob die Codegenerierung im Zusammenhang mit Prozeßsprachen der Art von PEARL gesehen wird oder nicht: es wird im weiteren lediglich noch berücksichtigt, daß auch Programme übersetzbar sein müssen, deren Ausführung nicht nur aus dem Ablauf eines einzigen Rechenprozesses besteht.

So wird deutlich, daß die ursprüngliche Unterteilung des Übersetzers in einen zielrechnerunabhängigen Compileroberteil und Behandlung aller maschinenspezifischen Aufgaben im CG revidiert werden muß.

In einer zielrechnerunabhängigen Zwischensprache dürfen Objektadressen und Adreßdifferenzen nur symbolisch ausgedrückt werden. Bei der CGg muß dann ein Namensbuch geführt werden, in dem zu jeder symbolischen Adresse der entsprechende zielrechnerspezifische Adreßwert enthalten ist (ausgenommen zu Identifikatoren externer Objekte). Die Adreßwerte von Objekten, die dem aktuell übersetzten Modul angehören, sind relativ zum Anfang dieses Moduls zu interpretieren. Da keine Typprüfungen notwendig sind, entfallen auch die entsprechenden Einträge ins Namensbuch.

Eigenschaften von Assemblersprachen, die dem Komfort menschlicher Programmierer dienen, wie Makro-Ersetzung, lexikalische Äquivalenz und anderes, brauchen im CODE-Entwurf ebenfalls nicht berücksichtigt werden. Schließlich soll CODE einfach darstellbar sein, um im CG das Erkennen der Befehle und - zu Testzwecken - eine Konvertierung in eine assemblerartig-mnemotechnische Form zu erleichtern.

3.2 Die abstrakte CODE-Maschine

Die Bezeichnung "abstrakte CODE-Maschine" wurde gewählt, weil insbesondere die CODE-Befehle mit Hilfe einer abstrakten Syntax definiert werden, die Zwischensprache also nicht konkret festgelegt wird. So kann die Funktion der Befehle geschildert werden, ohne die Implementierung vorwegzunehmen. In geringerem Umfang wird von dieser Syntaxnotation auch schon bei der Beschreibung der Architektur der virtuellen Maschine in 3.2.1 und der CODE-Datentypen in 3.2.2 Gebrauch gemacht. Erläuterungen und Beispiele zu dieser Schreibweise, sowie eine alphabetisch sortierte Liste der Produktionen sind in Anhang 1 und Anhang 2 zu finden.

Syntaxvariable werden im Text manchmal als technische Begriffe eingesetzt. Sie sind am Wortanfang durch Sonderzeichen markiert; ihre Bedeutung geht meist schon aus dem Wort selbst hervor. In vielen Fällen wird deshalb darauf verzichtet, Produktionen für solche Syntaxvariable zu geben; statt dessen werden sie doppelt markiert. In der folgenden Tabelle sind alle Markierungen aufgeführt:

Kennzeichnung	von Wörtern ...
\$ _	... aus dem Bereich der CODE-Daten und -Datentypen (D-ollar wie D-aten)
\$\$ _	..., die mit der Architektur der CODE-Maschine zu tun haben (Proze-nt wie Proze-ssor)
% _	... aus der Syntax für die Befehle (Be-ta wie Be-fehle)
%% _	... aus anderen Produktionen
β _	...
ββ _	...
& _	...
&& _	...
\$ _	..., mit denen Generierparameter bezeichnet werden (s. Kap. 4) (Para-graph wie Parameter)

Tabelle 3.2-1: Markierungen für Wörter mit spezieller Bedeutung

3.2.1 Architektur der CODE-Maschine

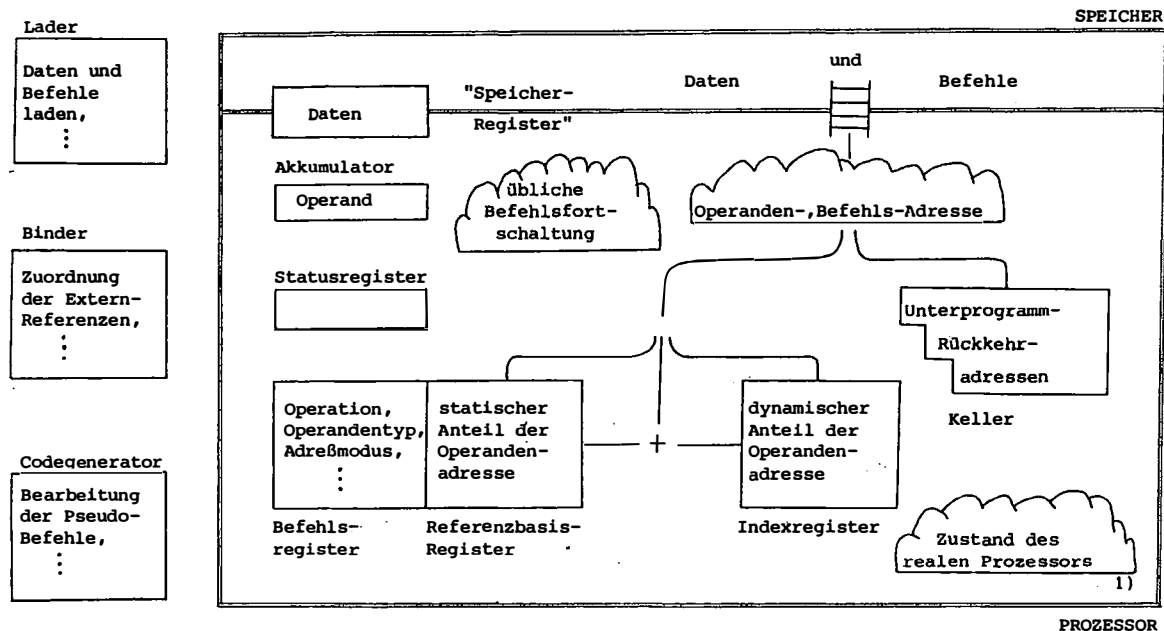
In Abb. 3.2.1-1 ist der Aufbau der CODE-Maschine skizziert, wobei die Registerausstattung im Vordergrund steht und die Adressierungsmechanismen noch angedeutet sind. Die folgende Beschreibung, wie CODE-Programme insgesamt bearbeitet werden, hat den Zweck, mögliche Mißverständnisse bei der Implementierung auszuschließen; ein Beispiel wäre die mehrfache Bearbeitung von Platzreservierungen zur Zeit der Ausführung übersetzter Programme.

Die Arbeitsweise

Die Bearbeitung eines Programms durch die CODE-Maschine erfolgt in zwei Phasen: in der ersten, der Übersetzungsphase werden die Befehle in der Reihenfolge ihrer Notierung durchlaufen, wobei aus Platzreservierungen und ausführbaren Befehlen der Bindecode erzeugt wird. In der zweiten Phase arbeitet die CODE-Maschine genau dann definiert, wenn im Programmablauf ausschließlich ausführbare Befehle erreicht werden; es gibt allerdings keinen vorbestimmten Startpunkt. Die Wirkung der ausgeführten Befehle ist in 3.2.4.3 und 3.3.3 festgelegt.

Über die ausdrücklich angegebene Wirkung hinaus werden von ihnen keine Registerinhalte verändert; z.B. verändern Transferbefehle nicht den Inhalt der Quelle. Diese harmlos erscheinende Regelung erleichtert es dem Compileroberteil, weniger ineffiziente Programme zu erzeugen, und stellt für den CG-Implementator eine zusätzliche Fehlermöglichkeit dar.

Auf das `%_Befehlsregister` ist kein expliziter Zugriff möglich; es gibt nur die übliche Befehlsfortschaltung, in die mit Sprungbefehlen eingegriffen werden kann; sie bewirken die entsprechende Veränderung der Befehlsadresse.



- ✘ Unterbrechungen
- ✘ periphere Elemente, E/A
- ✘ Mächtigkeit des Befehlssatzes

Abb. 3.2.1-1: Die virtuelle CODE-Maschine (in der Ausführungsphase)

1) Wolkentechnik nach /EBER79/

Nur 1 Verarbeitungsregister

Die CODE-Maschine hat nur einen einzigen %_Akkumulator, der als universelles Verarbeitungsregister dient: er kann Objekte aller in CODE bekannten Datentypen aufnehmen. Auch Adreßberechnungen können nur hier ausgeführt werden und nicht im %_Indexregister, das nur mit einem Akkumulator-Inhalt oder aus dem %_Speicher geladen werden kann.

Diese Ausstattung ist archaisch, muß aber vor dem Hintergrund der oft stark verkümmerten Auslegung realer Rechner gesehen werden. Es wäre zwar moderner gewesen, einen Operandenkeller vorzusehen, um für die Übersetzung arithmetischer Ausdrücke die real vorhandenen Register besser zu nutzen, jedoch stehen dem zwei praktische Gründe entgegen:

a) Es müßte die Orthogonalitätsforderung erfüllt sein, daß alle Operationen mit allen Kellerzellen, also auch den ins Auge gefaßten Registern, möglich sind; die meisten 8-Bit-Mikroprozessoren bieten in dieser Hinsicht sehr schlechte Voraussetzungen: entweder ist von vornherein die Registerausstattung dürftig oder die Operationen sind alles andere als orthogonal auf die vorhandenen Register verteilt. Die Tabelle ZIEL-3/1 in /LIND81/ belegt dies eindrucksvoll. Es bliebe noch die Möglichkeit, die oft vorhandenen realen Speicherkeller einzusetzen. Da diese aber schon zur Aufnahme der Unterprogramm-Rücksprungsadressen dienen, würde eine statische Realisierung des Operandenkellers verhindert.

b) Bei der Erstellung der Codegeneratoren müßten für die Operationen auf gekellerten Objekten zwei Versionen konstruiert werden, eine register- und eine speicherbezogene Ausgabe.

Beides zusammen legt den Verzicht auf einen Operandenkeller nahe. Dieser Verzicht wirkt sich auch in der nächsten Entscheidung aus:

Einadreß-Architektur

Ein Teil der realen Rechner sind sogenannte Zweiadreß-Maschinen, bei denen dyadische Befehle zwei Speicher-Referenzen enthalten können. Das Operationsergebnis ersetzt einen der beiden Operanden. Damit ließe sich der Operandenkeller, auf den eben verzichtet wurde, leicht implementieren. Ohne ihn stellt diese Form der Zweiadreß-Architektur nur noch eine Optimierungsmöglichkeit dar. Die CODE-Maschine weist deshalb nur ein-fache Operandenadressierung auf.

Fast keine Statusanzeigen

Die Ausformung des %_Anzeigeregisters hat den Charakter einer Notwehr gegen das reale "Architekturchaos": nach der Ausführung von Numerik-Befehlen (s. 3.2.4.3) zeigt es an, welches Vorzeichen das Ergebnis hat, ob es Null ist und ob ein Fehler aufgetreten ist (Überlauf, Division durch Null usw.). Sein Inhalt kann auf andere Art nicht verändert werden; lesender Zugriff erfolgt ausschließlich durch die Ausführung bedingter Sprung-Befehle.

Hier könnte durch eine geringfügige Verbesserung der realen Rechner erreicht werden, daß Ausnahmefall-Behandlungen und(!) die Auswertung von Statusinformation einfach und effizient implementiert werden könnten:

Angefallene Statusinformation dürfte nicht nur in Registern abgelegt werden; zusätzlich müßte nach Vergleich mit einem vorgebbaren erwarteten Status eine maskierbare Unterbrechung im Sinne der Richtlinie VDI/VDE 3554 /VDIE76/ erzeugt werden. In Abb. 3.2.1-2 ist dieser Ablauf dargestellt.

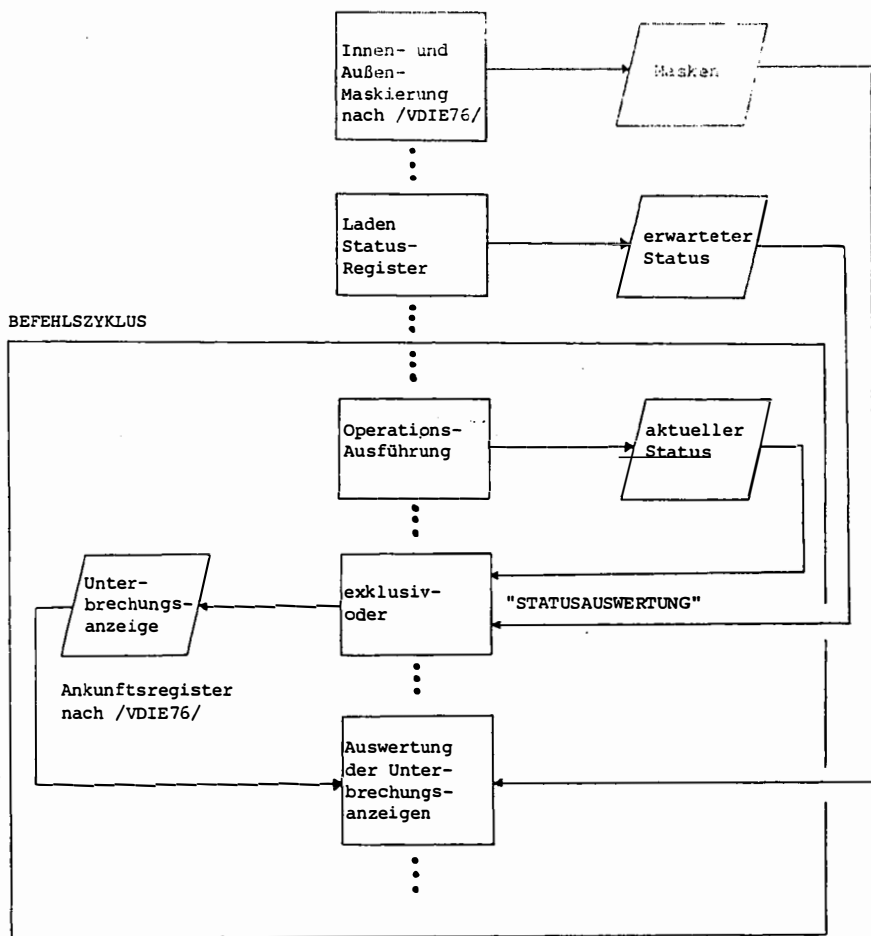


Abb. 3.2.1-2: Erzeugung interner Unterbrechungen in der Befehlsphase "Statusauswertung"

CODE-Register und Prozeßwechsel

Der `%Rücksprungkeller` ist genauso im CODE-Prozessor untergebracht, wie der Zustand des realen Prozessors. Damit gehören beide - so wie alle anderen CODE-Register auch - zum Kontext des aktuell ausgeführten Prozesses; es wird vorausgesetzt, daß dieser bei Prozeßwechsel vom Betriebssystem verwaltet wird.

Bei den `%Speicherregistern` handelt es sich um "lokale" Speicherplätze, die in Zielcode-Sequenzen (siehe CG-Generierung in Kap. 4) benötigt werden, und von denen gewünscht wird, daß auch sie als CODE-Register behandelt werden. Obwohl viel Ähnlichkeit besteht, darf diese Art Speicherregister nicht mit Register-Konstruktionen in der sogenannten speicherorientierten Architektur (z.B. TI-9900) verwechselt werden.

Der Unterschied zwischen Speicherzellen und Registern der CODE-Maschine besteht ausschließlich darin, daß die Information, die in Registern enthalten ist, bei Prozeßwechsel automatisch gerettet bzw. restauriert wird, während dies für Speicherinhalte nur in dem erwähnten Sonderfall gilt. Insbesondere wird keine Aussage darüber gemacht, ob die Register des CODE-Prozessors durch reale Register oder durch Speicherbereiche des Zielrechners darzustellen sind.

Keine Kontrollen

Viele der Vorschriften, die in den Abschnitten 3.2.4 und 3.3 gemacht werden, könnten den falschen Eindruck hervorrufen, daß ihre Einhaltung in irgendeiner Weise überprüft würde. Da aber CODE primär als Übersetzerinterne Schnittstelle gedacht ist, werden Fehlerkontrollen während der CGg nicht vorgesehen.

Die Verletzung solcher Regeln führt nichtsdestoweniger dazu, daß das Ergebnis des CG-Laufs undefiniert ist.

Abschließend sei noch bemerkt, daß die CODE-Maschine keine Befehle der Art "Stop", "No operation" usw. kennt. Sie werden im Rahmen des vorgeschlagenen Konzepts nicht benötigt.

Die folgenden Produktionen stellen die Komponenten der CODE-Maschine zusammenfassend dar:

%_Die_virtuelle_CODE_Maschine_in_der_Ausführungsphase

```
= %%_Speicher_für_Daten_und_Befehle
- %_Registersatz_des_Prozessors
/*Offenes Ende : */
- %%_andere_Komponenten_der_virtuellen_Maschine
```

%_Registersatz_des_Prozessors

```
= %%_Akkumulator
- %_Statusregister
- %%_übliche_Befehlsfortschaltung
- %_Befehlsregister
- %_Befehls_und_Operanden_Adresse
- %%_Rücksprungadressen_Keller
- %%_Speicherregister /*Näheres siehe 4.1*/
- %%_Zustand_des_realen_Prozessors
```

%_Statusregister

```
= %%_Fehleranzeige
- %%_Vorzeichen
- %%_Nullanzeige
```

%_Befehlsregister

```
= %%_Befehlsschlüssel
- %%_Typ
- %%_Adressiermodus
- %%_Sprungbedingung
- %%_Schiebe_Richtung_und_Schrittzahl
```

%_Befehls_und_Operanden_Adresse

```
= %%_Referenzbasis-Register
- %%_Indexregister
```

Abb. 3.2.1-3: Die wesentlichen Komponenten der CODE-Maschine

3.2.2 Die CODE-Datentypen

In CODE dienen Datentypen

- a) als Längenmaße, d.h. zur zielrechnerunabhängigen Darstellung von Adreßdifferenzen, z.B. für die Reservierung von Speicherbereichen,
- b) zur Angabe der Codierung bei der Erzeugung von Daten und
- c) zur Qualifikation der Operationen in ausführbaren Befehlen.

Die Punkte b) und c) stellen den wesentlichen Teil der Schnittstelle zwischen der Laufzeitunterstützung und der CGg dar: die Darstellung der Datenobjekte muß zwischen dem CG und den Laufzeitroutinen abgesprochen sein, damit nicht während der Ausführung der übersetzten Programme Wandlungsroutinen aufgerufen werden müssen.

Da die Bedeutung eines ausführbaren Befehls nicht allein von der Operation abhängt sondern u.a. auch vom Typ, geht die Typinformation in die Aufrufe von Laufzeitprogrammen ein (siehe 3.3.3 und Kap. 4). Der Punkt c) hat überdies noch Einfluß auf die Verzweigungsstruktur der CGen bei der Übersetzung der ausführbaren Befehle ("Befehlsentschlüsselung" in 3.3.3).

Auswahl der Datentypen

Geht man von den realen Rechnern aus, die in Betracht gezogen wurden (s. Kap. 1), so kommen als Datenarten, die durch eigene Binärcodierung und/oder entsprechende Operationen ausgezeichnet sind, in Betracht:

- natürliche und ganze Zahlen,
- rationale Zahlen,
- Bitketten und
- Adressen.

Es wird versucht, CODE im wesentlichen auf diese Typen zu beschränken.

Natürliche und ganze Zahlen

Bei der Darstellung ganzer Zahlen im Einer- oder Zweier-Komplement genügen zur Unterscheidung ganzer und natürlicher Zahlen Statusanzeigen, die in Abhängigkeit vom Ergebnis ein und derselben Operation gesetzt werden. Das Einerkomplement ist allerdings nur noch selten vertreten (z.B. in der PDP-15). Die Darstellung durch binär codierte Dezimalziffern ("BCD-Zahlen") könnte in der Zukunft bei Prozeßrechnern mehr Bedeutung als bisher erlangen, da solche Systeme immer häufiger auf dem Gebiet der Textkommunikation und -Verarbeitung eingesetzt werden, wo diese Art der Darstellung Vorteile hat.

Insgesamt kann davon ausgegangen werden, daß für ganze Zahlen wenige, allgemein übliche Codierungen existieren. Somit ist es sinnvoll, in der Zwischensprache verschiedene Darstellungen zu berücksichtigen und bei der CG-Generierung einen entsprechenden Auswahlmechanismus vorzusehen (s. 3.3.2 und Kap.4).

Rationale Zahlen

Dazu steht die Situation bei rationalen Zahlen in Gegensatz. Erstens ist Gleitkomma-Arithmetik bei Rechnern mit bis zu 16 Bit Wortlänge eher die Ausnahme; zweitens erfordert es, wie bei /LIND81/ zu sehen, beträchtlichen Aufwand, die verschiedenen realen Darstellungsarten in portabler Technik zu bedienen. In CODE gibt es daher nur eine einzige Form rationaler Zahlen, die erst bei CG-Generierung in Abhängigkeit vom Zielrechner festgelegt wird (s. 3.3.2 und Kap.4).

Bitketten

Bitketten sollen dem Compiler-Oberteil nicht zuletzt gestatten, Zeichen und Zeichenketten darzustellen, die in CODE nicht eingeführt werden. Dies ist sinnvoll, da die Codierung von Zeichen weniger ein Merkmal der Rechen-Prozessoren ist, als vielmehr der peripheren Geräte und der entsprechenden Systemroutinen.

Die Konsequenz ist nun nicht, daß der COT unbedingt alle zielsystemspezifischen Zeichen-Verschlüsselungen kennen müßte; es liegt vielmehr nahe, auf der Ebene der rechnerinternen Verarbeitung nur eine Darstellung zu berücksichtigen und die Wandlungen, die für Ein- und Ausgabe notwendig sind, den entsprechenden Formatier Routinen zu überlassen. Die "Aufweitung" von Datenobjekten (siehe 3.2.4.2) gestattet es, bei Bedarf in zielrechnerabhängiger Weise zu steuern, wie viele Zeichen jeweils in einem realen Rechnerwort gespeichert werden sollen.

Die bis hier diskutierten Datenarten sind allerdings erst Darstellungsarten, die in CODE noch durch entsprechende Längenangaben zu ergänzen sind. Diese Längen müssen ebenfalls rechnerunabhängig angegeben werden; die Einheit kann (je nach Konkretisierung) von der Darstellungsart abhängen.

Adressiereinheit des Zielrechners ?

Um einen direkten Durchgriff auf die Zielcode-Ebene zu ermöglichen, könnte ein Datentyp `$$Adressiereinheit` eingeführt werden, mit der unorthogonalen Einschränkung, nur in `ß`-Platzreservierungen (s. 3.2.4.2 und 3.3.2) erlaubt zu sein. Als Objekte dieses Typs würden Bitketten einer bestimmten Länge zugelassen, die durch den Generierparameter `$ _AdrEinh_Länge` (s. Kap. 4) gegeben wäre. So hätte der COT bzw. ein Programmierer die Möglichkeit, Bindecode in genau den Portionen zu erzeugen, die am Zielrechner adressierbar sind. Im Rahmen der weiteren Betrachtungen ist dieser Aspekt aber von untergeordneter Bedeutung.

Adressen

Adressen werden innerhalb der CG-Programme in Einheiten der eben erwähnten Adressiereinheiten gezählt; es könnte also erwartet werden, daß sie in der Zwischensprache keinen eigenen Typ definieren und nur als Teilmenge eines anderen Typs, z.B. natürlicher Zahlen mit einer zielrechnerabhängig ausgezeichneten Länge betrachtet werden. Trotzdem werden sie in CODE getrennt als `$$_Adressen` eingeführt.

Das liegt sowohl daran, daß auf den meisten realen Rechnern die Adreßarithmetik eine Sonderrolle spielt, als auch an der Rolle von Adreßwerten während der CGg selbst: der erste Gesichtspunkt betrifft die Effizienz der Adreßberechnungen bei der Ausführung der übersetzten Programme; der zweite bezieht sich auf die Notwendigkeit, bei der CGg zwischen Adreßdifferenzen und Objektadressen unterscheiden zu können: beim Ablegen von Adreßwerten im Bindecode muß feststellbar sein, ob es sich um Adreßdifferenzen handelt, die vom Binder nicht mehr modifiziert werden dürfen. Die vorgenommene Unterteilung ist zwar äquivalent mit der in "absolute" und "relative" Adressen, aber im Rahmen der CODE-Definition erlaubt die erste Wortwahl eine anschaulichere Beschreibung.

Eine Kennzeichnung dieses Unterschieds wird jedoch nur bei der Erzeugung von Datenobjekten des Typs \$\$_Adresse (s. 3.2.4.2 und 3.3.2) notwendig sein; bei den Referenzen B_ausführbarer_Befehle ist diese Markierung nicht wünschenswert, da sonst Objekte zielrechnerabhängig, referenziert würden. Also genügt es, die Namensbuchzugriffe bei der Vorbelegung von Adreß-Objekten so auszuformen, daß die jeweilige Interpretation des eingetragenen Pegelwertes mit anzugeben ist.

Es wird nur ein einziger Typ \$\$_Adresse definiert und beispielsweise nicht zwischen langen oder verkürzten Adressen unterschieden; somit wird die Existenz nur eines einzigen Adreßbereichs berücksichtigt, dessen Umfang durch den Generierparameter \$_Adreßlänge indirekt festgelegt ist (s. Kap. 4). Auch Unterschiede zwischen Halbwort-, Wort-, Doppelwortadressen usw. gibt es nicht: alle Daten- und Befehlsobjekte müssen an beliebigen Zielrechneradressen gespeichert werden können.

In der Syntax 3.2.2-1 sind die CODE-Datentypen noch einmal zusammengestellt.

\$_Datentyp

```

-----
=  $$_Adressiereinheit
=  $_Akkutyp
/* Offenes Ende : */
=  $$_andere_Datentypen
-----

```

\$_Akkutyp

```

-----
=  $$_Adresse
=  $_Arithmetiktyp
-----

```

\$_Arithmetiktyp

```

-----
=  $$_Bit - $$_Länge
=  $_Numeriktyp
-----

```

\$_Numeriktyp

```

-----
=  $$_Ratio - $$_Länge
=  $_Ganztyp
-----

```

/*****/

\$_Ganztyp

```

-----
=  $$_E_Kompl - $$_Länge
=  $$_Z_Kompl - $$_Länge
=  $$_gep_BCD - $$_Länge
=  $$_ung_BCD - $$_Länge
-----

```

\$_Rationaltyp

```

-----
=  $$_Ratio - $$_Länge
-----

```

\$_Bittyp

```

-----
=  $$_Bit - $$_Länge
-----

```

\$_Adreßtyp

```

-----
=  $$_Adresse
-----

```

Abb. 3.2.2-1: Die CODE-Datentypen

3.2.3 Adressier-Mechanismen

Geht man von realen Rechnern aus, lassen sich drei grundlegende und verbreitete Arten der Operandenadressierung feststellen:

- "unmittelbar",

d.h., der Operand ist im Befehl enthalten,

- "direkt",

wobei die Adresse des Operanden Teil des Befehls ist,

- "indirekt",

bei der im Befehl kein Adreßanteil vorliegt sondern die Operandenadresse in einem Register zu finden ist.

Diese Grundmuster korrelieren unmittelbar mit den drei Referenzstufen 0, 1 und 2 /BAGO71/. Von ihnen sind die unterschiedlichsten Variationen, Kombinationen, Schachtelungen usw. zu finden, die von der unerschöpflichen Kreativität der Rechnerkonstrukteure bereitetes Zeugnis ablegen.

Auf die Berücksichtigung von Hardware-Eigenschaften, die Optimierungszwecken dienen, wird hier verzichtet; als Beispiel seien Befehlsmodifikationen erwähnt, die eine automatische Veränderung der (Komponenten der) Operandenadresse vor oder nach dem Zugriff bewirken und zur Realisierung eines Operandenkellers (s. 3.2.1) eingesetzt werden könnten.

Auch der Adressiermodus "unmittelbar" kann als Optimierung betrachtet werden; er ist zwar häufig verwirklicht, aber die Wertebereiche der Operanden sind fast immer drastisch beschnitten. Ihn in CODE einzubeziehen, würde deshalb die Implementierung der CGen belasten statt erleichtern.

In den ausführbaren Befehlen von CODE, mit denen auf Operanden im Speicher zugegriffen wird, werden daher - wie schon in Abb. 3.2.1-1 zu erkennen ist - drei Adressiermodi zur Verfügung gestellt:

-direkt

(symbolische Adresse im CODE-Befehl; keine Auswertung des Indexregisters),

-indiziert

(zusätzlich: Addition des Indexregisterinhalts zum Wert der symbolischen Adresse) und

-indirekt

(keine symbolische Adresse im Befehl; Operandenadresse im Indexregister).

In dieser Beschränkung auf eigentlich nur zwei Mechanismen (direkt, indirekt) und ihre Kombination zu einem dritten wird wiederum deutlich, daß diese Art, die CODE-Maschine aufzubauen, weitgehend den Charakter einer Durchschnittsbildung hat.

3.2.4 Der Befehlssatz der abstrakten CODE-Maschine

Die CODE-Befehle können in drei Klassen eingeteilt werden, die β _Pseudobefehle, die nur der Übersetzungssteuerung dienen, zwei β _Platzreservierungen, die wie die Pseudobefehle den Zustand der virtuellen Maschine in der Ausführungsphase nicht verändern (s. 3.2.1), aber wie die β _ausführbaren_Befehle zur Erzeugung von Bindecode führen können, und die dritte Gruppe, die ausführbaren Befehle:

β _CODE_Befehl

```
-----  
=    $\beta$ _Pseudobefehl  
=    $\beta$ _Platzreservierung  
=    $\beta$ _ausführbarer_Befehl  
=   /* Offenes Ende : */  
=    $\beta$ _andere_CODE_Befehle  
-----
```

β _Pseudobefehl

```
-----  
=    $\beta$ _Modulanfang  
=    $\beta$ _Modulende  
=    $\beta$ _NaB_eintragen /* Namensbuchführung */  
=    $\beta$ _Import  
=    $\beta$ _Export  
-----
```

β _Platzreservierung

```
-----  
=    $\beta$ _Pegel_verstellen  
=    $\beta$ _Datencode_erzeugen  
-----
```

β _ausführbarer_Befehl

```
-----  
=    $\beta$ _Transfer  
=    $\beta$ _Numerik  
=    $\beta$ _Adreß_Rechnung  
=    $\beta$ _Bitketten_Verarbeitung  
=    $\beta$ _Sprung  
-----
```

3.2.4.1 Die Pseudobefehle

Die ersten beiden Befehle haben den Zweck, zu Beginn bzw. am Ende eines CG-Laufes die notwendigen Initialisierungen bzw. Abschlußarbeiten anzustoßen:

ß_Modulanfang

```

-----
=   Befehlsschlüssel : ß_Modul_eröffnen
-----

```

ß_Modulende

```

-----
=   Bef.Schl.       : ß_Modul_abschließen
-----

```

Namensbuch-Führung

Im Gegensatz zu Assemblersprachen bildet die Definition von Objektadressen und Adreßdifferenzen in CODE nicht einen Teil anderer Anweisungen, sondern stellt einen eigenen Befehl dar, das Vornehmen einer Namensbuch(NaB)-Eintragung:

ß_NaB_eintragen

```

-----
=   Bef.Schl. : ß_Definition_einer_symbolischen_
               Adresse
-   Identifikation : &&_Definierendes_Auftreten_
               eines_Identifikators
-   Zugeordneter Pegel : &&_Adreß_Ausdruck
-----

```

Der Adreßausdruck wird ausgewertet und ein entsprechender Eintrag ins Namensbuch vorgenommen; der Wert des Ausdrucks darf negativ sein. Die Konstituenten des Ausdrucks sind ausschließlich symbolischer Natur und müssen bereits definiert sein. Insbesondere kann lesend auf den aktuell erreichten Wert der Pegelzählung zugegriffen werden (s. Pegelverstellung in 3.2.4.2).

In CODE wird weder ein Unterschied zwischen Adressen von Daten und Befehlen gemacht (s. 3.2.1), noch zwischen Adressen und Adreßdifferenzen (mit einer Ausnahme; s. Typ Adresse in 3.2.2 und Datencode-Erzeugung in 3.3.2). So können die Adreßausdrücke 'sowohl zum Zugriff auf Teile zusammengesetzter Objekte /SCHNBO/ eingesetzt werden als auch zum Anlegen von Sprungverteilern.

Import

Für den Zugriff auf modulexterne Objekte gibt es in CODE eine Spezifikation in primitiver Ausführung:

ß_Import

```

-----
=   Bef.Schl. : ßß_Spezifikation_eines_externen_
      Objekt
-   Ident.    : &&_Definierendes_Auftreten_eines_
      Identifikators
-----

```

Das Bild 3.2.4.1-1 zeigt in vereinfachter Darstellung, warum während der CGg die Kenntnis der globalen Identifikatoren externer Objekte nicht nötig ist. Als Wert des NaB-Eintrags genügt eine Markierung, daß der Pegelwert für den (lokalen) Identifikator innerhalb der aktuellen Übersetzung undefiniert bleiben wird. Diese Markierung ist zwar nicht unbedingt notwendig, erleichtert aber eine effiziente Implementierung des Binders.

Export

Um externe Zugriffe auf modulinterne Objekte zu ermöglichen, muß der Binder den modulrelativen Pegel erfahren, der zum Identifikator dieses Objekts gehört; dazu dient der Exportbefehl:

ß_Export

```

-----
=   Bef.Schl. : ßß_Erzeugung_eines_Eintrags_in_der_
      Externdefinitions_Tabelle
-   Ident.    : &&_angewandtes_Auftreten_eines_
      Identifikators
-----

```

Auch hier wird im CODE-Programm der globale Identifikator nicht benötigt (s. Abb. 3.2.4.1-2).

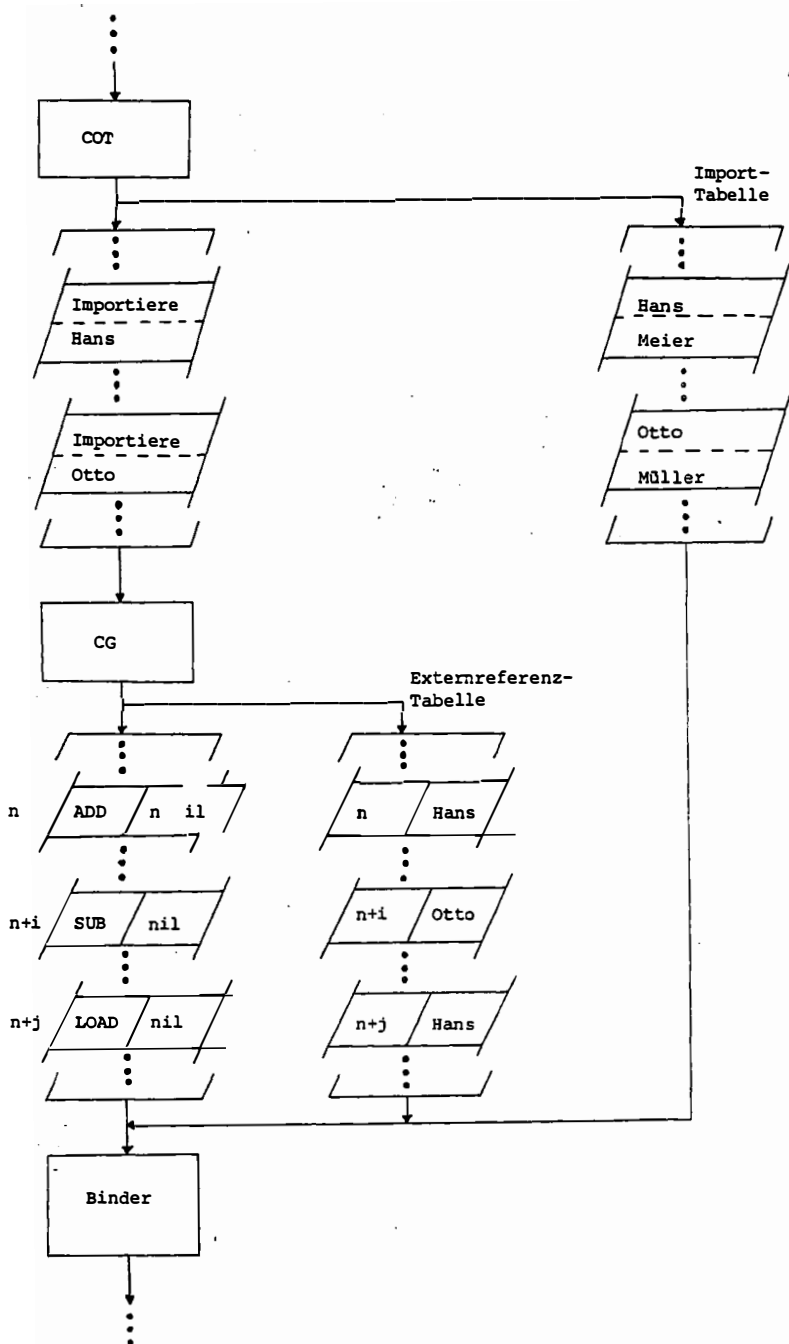


Abb. 3.2.4.1-1: Behandlung von Externreferenzen

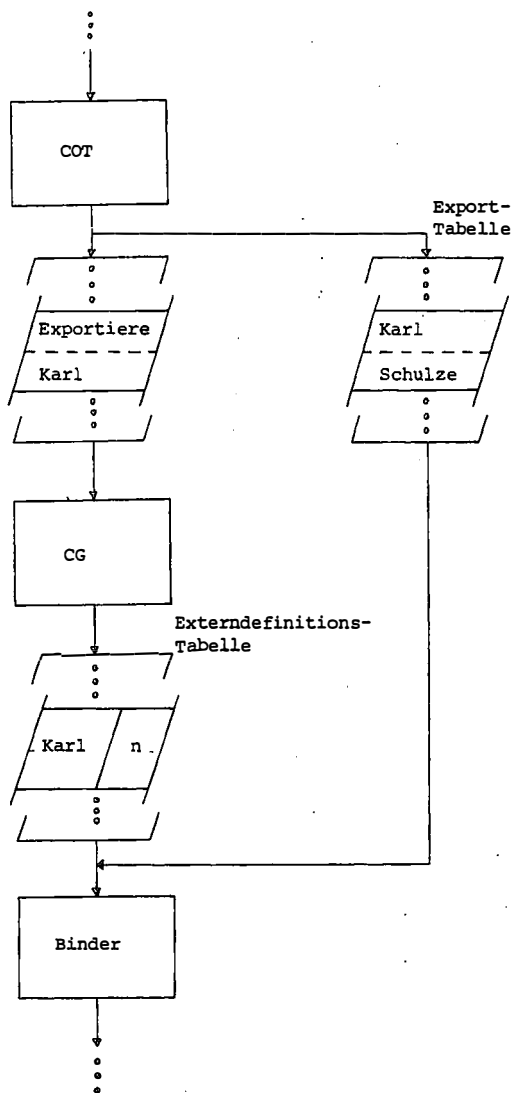


Abb. 3.2.4.1-2: Exporte aus dem aktuellen Modul

3.2.4.2 Platzreservierung

Reservierung von Speicherplatz ohne Vorbelegung erfolgt durch entsprechende Erhöhung des aktuell erreichten Pegelwertes:

β Pegel_verstellen

```

-----
= Bef.Schl.      :  $\beta\beta$  Verschiebung des
                  aktuellen Pegels
- Pegelveränderung : && Adreß_Ausdruck
-----

```

Für den Adreßausdruck gelten die gleichen Regeln, wie beim β NaB eintragen (s. 3.2.4.1). Die Pegelverstellung kann auch zur Realisierung statischer Blockstruktur eingesetzt werden (s. Adreßbuchführung in 3.3.1, Pegelverstellung in 3.3.2).

Platzreservierung mit Vorbelegung:

β Datencode_erzeugen

```

-----
= Bef.Schl.      :  $\beta\beta$  Erhöhung des aktuellen Pegels
                  durch Vorbelegung
- Vorbelegung    : && zielrechnerunabhängige
                  Darstellung des Objekts
-----

```

Der Bindecode wird um die Datenobjekte erweitert, die vorher in zielrechnerspezifische Darstellung umzuwandeln sind; der aktuelle Pegel wird entsprechend erhöht (zu den Datentypen siehe 3.2.2, zur Umwandlung siehe 3.3.2).

Das Aufweiten (erster Teil)

Der Bindecode wird in diskreten Portionen erzeugt, den $\S\S$ Adressiereinheiten des Zielrechners. Dies wirft bei Adressiereinheiten selbst und Adressen keine Probleme auf; bei den Objekten anderer Typen muß jedoch ggf. durch entsprechende Verlängerung dafür gesorgt werden, daß keine "halben Portionen" entstehen. Diese Aufweitung von Arithmetiktyp-Objekten wird aber nicht nur bis zum Erreichen der nächsten Adreßgrenze des Zielrechners durchgeführt, sondern gleich bis zum nächsten Vielfachen seiner \S Wortlänge (siehe Generierparameter in Kap. 4). So wird die Möglichkeit geschaffen, ausführbare Befehle, für die prinzipiell unterschiedlicher Zielcode vorzusehen wäre, gleich zu behandeln (s. 3.3.3).

Natürlich muß diese Aufweitung schon bei Auswertung der Adreßausdrücke im `B_NaB` eintragen berücksichtigt werden: in der "Längenumrechnung" bei Bearbeitung dieses Befehls (s. 3.3.1) wird dem Rechnung getragen.

3.2.4.3 Die ausführbaren Befehle

Die hier gewählte Einteilung erfolgt noch nicht in Hinsicht auf Implementierungen, sondern hat das Ziel, Befehlsklassen derart zu bilden, daß die Erweiterungsmöglichkeiten sichtbar bleiben; der gegebene Befehlssatz hat also den Charakter einer (ziemlich) unverbindlichen Vorschlagsliste. Welche Befehle in einer aktuellen Implementierung des Generiersystems berücksichtigt werden, kann in Abhängigkeit davon entschieden werden, welche Anforderungen von der zu übersetzenden Programmiersprache gestellt werden. Ein Beispiel für eine solche Auswahl stellt die Konkretisierung in 3.3.3 dar.

`B_ausführbarer_Befehl`

```
-----  
=   B_Transfer  
-   B_Numerik  
-   B_Adreß_Rechnung  
-   B_Bitketten_Verarbeitung  
-   B_Sprung  
-----
```

Transferbefehle

Bei den Transferbefehlen wird der Inhalt des Ziels durch den der Quelle ersetzt; die Information in der Quelle bleibt erhalten:

B_Transfer

```

-----
= BB_Laden_Akku      - $ _Akkutyp - B_Referenz
= BB_Speichern_Akku  - $ _Akkutyp - B_Referenz
= BB_Laden_XReg      /* $$_Adresse */ - B_Referenz
= BB_vom_Akku_ins_XReg /* auch $$_Adresse */
-----

```

Da festgelegt wurde, daß im Indexregister keine Rechenergebnisse anfallen, ist ein Befehl zum Speichern seines Inhalts überflüssig.

B_Referenz

```

-----
= Adressiermodus : BB_direkt
- Ident. : &&_angew_Auftreten_eines_Identifikators
= Adressiermodus : BB_indiziert
- Ident. : &&_angew_Auftreten_eines_Identifikators
= Adressiermodus : BB_indirekt
-----

```

Siehe auch 3.2.3 (Adressiermechanismen).

Der Wert des NaB-Eintrags für den Identifikator muß positiv sein. Es darf sich nicht um eine Adreßdifferenz handeln. Bei indizierter Adressierung muß der Indexregisterinhalt eine positive Adreßdifferenz sein, bei indirektem Zugriff eine Objektadresse, wie sie bei direkter Referenzierung erlaubt ist.

Damit wird ausgeschlossen, daß die Adreßauswertung im CODE-Prozessor zur Addition von Objektadressen mißbraucht wird, oder zur Objektreferenzierung mit absoluten Adressen. Auch gibt es zur Ausführungszeit keine negativen Anteile von Operandenadressen, wie sie in realen Rechnern (z.B. TR-440, MCS-6500, ...) durchaus vorkommen. Allerdings ist diese Eigenschaft eher für den menschlichen Assemblerprogrammierer eine Erleichterung, als im Rahmen der Übersetzung höherer Programmiersprachen.

Numerische Operationen

B_Numerik

```

=====
= Befehlsschlüssel : B_Numerik_Operation
- Operationstyp : $_Numeriktyp
- Operand : B_Referenz
=====

```

B_Numerik_Operation

```

=====
= BB_Addit = BB_Subtr = BB_Multi = BB_Divis
= BB_Negat
/* Besonders offenes Ende :
= BB_andere_Numerik_Operationen */
=====

```

Der Akku-Inhalt wird entsprechend der Operation verarbeitet bzw. mit dem zweiten Operanden verknüpft. Das Ergebnis befindet sich im Akkumulator. Die Anzeigen im %_Statusregister werden gesetzt; im Fehlerfall ist der Ergebniswert im Akku undefiniert.

Keine Vergleiche, Aufweitung (zweiter Teil)

Es wird festgelegt, daß Vergleiche in CODE-Programmen mit Hilfe von Subtraktion und Anzeigenauswertung (durch bedingte B_Sprünge) dargestellt werden. Von der Subtraktion ganzer Zahlen

BB_Subtr - \$_Ganztyp - B_Referenz

wird deshalb gefordert, daß sie in Bezug auf das Setzen und Zurücksetzen der %%_Nullanzeige auch für Bitketten definiert arbeitet: bei Operandengleichheit wird die Nullanzeige gesetzt, bei Ungleichheit zurückgesetzt. Der Akkuinhalt und die übrigen Anzeigen dürfen undefiniert sein.

Diese Forderung wirkt sich auch auf die Implementierung der Datenobjekte aus (s. 3.3.2: Konversionsfunktionen beim B_Datencode_erzeugen), wo bei Aufweitung definierter Bindecode erzeugt werden muß, und bei der Realisierung ausführbarer Befehle (z.B. Schieben von Bitketten: s.u. und 3.3.3).

Adreßrechnung

Die Adreßrechnungs-Befehle unterliegen besonderen Einschränkungen. Hier wird bewußt gegen das Prinzip der Orthogonalität verstoßen, da sonst zu viele Konstrukte entstehen würden, die gar nicht benötigt werden, z.B. Multiplikation zweier Adressen oder ähnliches.

ß_Adreß_Rechnung

```

-----
=  ßß Addit - $$_Adresse - ß Referenz
/*_1.Operand muß ebenfalls eine Adresse sein */
=  ßß Subtr - $$_Adresse - ß Referenz
-----
=  ßß Multi - $$_Adresse - ß Referenz
/*_Akkuinhalt muß vom Typ
   $$_Ganztyp($$_Adresse)
   (s.u.) sein */
-----

```

Die Regelung für den Vergleich von Adressen ist analog zu der, die für den Vergleich numerischer Objekte gilt.

Bei der Adreß-Multiplikation wird vorausgesetzt, daß der Akkuinhalt ganzzahlig ist. Die Länge dieses (ausgezeichneten) Ganzzahltyps muß dem COT bekannt sein, stellt also für ihn einen zielrechnerabhängigen Generierparameter dar. In den CGen wird diese Information nicht benötigt.

Typ des aktuellen Akkuinhalts

Außer dieser Adreßmultiplikation gibt es keinen dyadischen Befehl, bei dem erster und zweiter Operand von verschiedenem Typ sein dürfen.

Die Ergebnisse aller Operationen sind ausnahmslos vom Typ der bzw. des Operanden. Insbesondere entsteht bei einer Multiplikation kein Ergebnis, das eine größere Länge hätte als die Operanden.

Das Ziel ist, an jeder Stelle des CODE-Programms den Typ des aktuellen Akkuinhalts aus dem letzten davorliegenden Numerik-Befehl bestimmen zu können. Dazu muß dem COT noch verboten werden, Konstruktionen der folgenden Art erzeugen:

```

:
:
BB_Laden_Akku - $_Ganztyp - B_Referenz
BB_Defin_einer_symbol_Adresse - "Hans" - "aktueller_Pegel"
:
:
BB_Laden_Akku - $$_Adresse - B_Referenz
BB_Springen - BB_unbedingt - BB_direkt - "Hans"
/* Sprungbefehle siehe übernächste Befehlsgruppe */
:
:

```

Sie gehören zum Bereich der maschinenorientierten Trickprogrammierung.

Bitketten-Verarbeitung

Für die Bitkettenverarbeitung werden jene Operationen vorgeschlagen, die nicht - wie etwa die Konkatenation - Ergebnisse größerer Länge liefern, als sie die Operanden haben.

B_Bitketten_Verarbeitung

```

-----
= Bef.Schl.      : B_Dyadische_Bitketten_Operation
- Operationstyp : $_Bittyp
- Operand       : B_Referenz
-----
= Bef.Schl.      : B_Monadische_Bitketten_Operation
- Operationstyp : $_Bittyp
-----

```

B_Dyadische_Bitketten_Operation

```

-----
= BB_Und = BB_einschl_Oder = BB_ausschl_Oder
-----

```

B_Monadische_Bitketten_Operation

```

-----
= BB_Komplex
= BB_Schieben - BB_Richtung - BB_Schritte
-----

```

Die Schrittzahl von Schiebefehlen muß kleiner als die Länge des Operanden sein. Alle Binärstellen des Operandenwerts, die über den rechten bzw. linken Rand (definiert durch die Länge des Operationstyps) hinausgeschoben werden, gehen verloren. Beim Implementieren der Komplementierung und der Verschiebung in Richtung auf das "aufgeweitete Ende" des Operanden (s. 3.2.4.2), wird zu berücksichtigen sein, daß der Vergleich von Bitketten durch Ganzzahl-Subtraktion erfolgt (s.o.).

Ablauf-Steuerung (mit Idealtendenz)

Bei den Sprungbefehlen wird die CODE-Maschine in ihrer abstrakten Version zunächst recht orthogonal und umfangreich aufgebaut, aber es ist beabsichtigt, im Laufe der Konkretisierung (s. 3.3.3) die Mächtigkeit wieder auf ein Maß zurückzunehmen, das eine schnelle CG-Implementierung für die realen Zielmaschinen erlaubt.

Zunächst jedoch wird für alle Befehle, die der Ablaufsteuerung dienen, u.a. die Möglichkeit vorgesehen, bedingt ausgeführt zu werden:

B_Sprung

```
- = Befehlsschlüssel : BB_Unterprogramm_Rücksprung
- - Sprungmodus      : B_Bedingung
-
- = Befehlsschlüssel : B_Sprung_Operation_mit_Ref
- - Sprungmodus      : B_Bedingung
- - Operand          : B_Referenz
```

B_Sprung_Operation_mit_Ref

```
- = BB_Springen          = BB_Unterprogramm_Sprung
- = BB_Koroutinen_Sprung = BB_Betriebssystemaufruf
```

B_Bedingung

```
- = BB_unbedingt
- = BB_wenn_Fehler          = BB_wenn_kein_Fehler
- = BB_wenn_positiv         = BB_wenn_negativ
- = BB_wenn_null            = BB_wenn_ungleich_null
```

Mit diesen fünf Sprungoperationen und sieben Bedingungen ergibt sich für drei Adressiermodi die Anzahl von 105 verschiedenen Sprungbefehlen. Im Hinblick auf die Möglichkeiten, die von der modernen Technologie der Rechnerherstellung geboten werden (insbesondere Mikroprogrammierung und hoch integrierte Halbleiterschaltungen), wäre das als Wunschliste an die Hersteller ohne weiteres vertretbar. Es ist aber klar, daß die Implementierung - allein dieses Satzes von Sprungbefehlen - für den zugrunde gelegten Zielrechnerkreis nicht als aufwandsarm gelten könnte.

Ablauf-Steuerung (mit Realitäts-Tendenz)

Von diesem optimistisch-orthogonal aufgebauten Satz wird deshalb eine Teilmenge gebildet, die schon mehr realitätsbezogen und immer noch einigermaßen orthogonal ist:

ß_Sprung

=	Befehlsschlüssel	:	ßß_Unterprogramm_Rücksprung
=	Befehlsschlüssel	:	ßß_Unterprogramm_Sprung
-	Operand	:	ß_Referenz
=	Befehlsschlüssel	:	ßß_Betriebssystemaufruf
-	Operand	:	ß_Referenz
=	Befehlsschlüssel	:	ßß_Springen
-	Sprungmodus	:	ß_Bedingung /*immer noch der
-	Operand	:	ß_Referenz Faktor 21 !*/
		:	..

Bei den bedingten Sprüngen ist zu beachten, daß die Statusanzeigen nur unmittelbar nach der Ausführung von Befehlen definiert ist, die diese Anzeigen auch beeinflussen (siehe auch CODE-Architektur in 3.2.1).

3.3 Eine Konkretisierung der CODE-Maschine

Um zu Implementierungsvorgaben für die CGen zu kommen, werden ausgehend von den Befehlen der abstrakten CODE-Maschine prinzipiell jeweils 3 Konkretisierungsschritte ausgeführt:

Im ersten werden Befehlsfelder gebildet, deren Reihenfolge definiert und ihr Inhalt ausführlicher festgelegt, als dies in 3.2.4 getan wurde. Es liegt in der Natur solcher Festlegungen, daß sie nicht nur den Charakter von Präzisierungen haben, sondern oft auch mehr oder minder starke Einschränkungen darstellen. Daher werden in diesem Schritt auch Einschränkungen vorgenommen, deren Ziel die Entlastung von Implementierungen ist, ohne daß in jedem Einzelfall darauf hingewiesen wird.

Als zweites wird eine mnemotechnische Form der konkretisierten Befehle eingeführt, damit bei der Aufstellung der Implementierungs-Regeln und -Anforderungen (dem dritten Schritt) ggf. bequem auf CODE-Befehle Bezug genommen werden kann. Dort wird in Form von

Umgangssprache,
Anweisungen und Kommentaren einer selbsterklärenden, nicht
näher definierten Programmiersprache und
Struktogrammen

besonders auf die Aspekte eingegangen, die im Zusammenhang mit der Erstellung der CGen stehen, d.h. wo Generierparameter (siehe auch Kap. 4) eingeführt werden. Die genannten Schritte werden nicht immer sequentiell sondern häufig quasiparallel ausgeführt.

Ergänzung zu den Datentypen aus 3.2.2

Der Einfachheit halber wird festgelegt, daß alle Längenangaben in Bit zu machen sind. Die Produktionen für die $\$_{Arithmetiktypen}$ (siehe Syntax am Ende von 3.2.2) werden geringfügig modifiziert, indem $\$_{Länge}$ ersetzt wird durch

```
 $\$_{Länge\_in\_Bit}$ 
+++++
+ = &&_natürliche_Zahl +
+++++
```

Weitergehende Konkretisierungen der Datentypen werden nicht mehr benötigt.

Die Typbezeichner \$\$_Adresse und \$\$_Adressiereinheit werden später auch als vordefinierte Identifikatoren benutzt, für die entsprechende NaB-Initialisierungen vorgesehen werden müssen (siehe ß_Modulanfang in 3.3.1). Der Wert des zweiten Eintrags muß 1 sein, der des ersten gibt an, wieviele Adressiereinheiten zur Speicherung eines Zielrechner-Adreßwerts notwendig sind.

Die Lesefunktion der Codegeneratoren

Anstatt eine zu starr formatierte Darstellung für CODE-Befehle einzuführen, wird zugelassen, daß sie unterschiedlich lang sein können, und eine Längenangabe vorgesehen:

ß_Befehlszeile

```

+++++
+ =      ßß Zeilenlänge
+   + ßß Befehlsdarstellung
+++++

```

Für die Befehlsdarstellung selbst wird folgender simpler Aufbau definiert:

ß_Befehlsdarstellung

```

+++++
+ =      ßß Befehlsschlüssel Feld
+   + ßß Folge der Versorgungs_Felder
+++++

```

Das erste Feld enthält immer den Befehlsschlüssel; es wird "automatisch" gelesen, wodurch die Übersetzung des jeweiligen Befehls eingeleitet wird. In Abhängigkeit vom Operationscode werden dann die restlichen Befehlsfelder nachgezogen.

In den Produktionen für die mnemotechnische Form von CODE wird die Art der Felder vor einem Doppelpunkt angegeben (siehe auch Erläuterungen zur Syntaxnotation in Anhang 1).

Reihenfolge und Kontext im CODE-Programm

In CODE müssen nur für zwei Befehle, den Modulanfang und das Modulende, Vorschriften gemacht werden, an welchen Stellen sie im Programm stehen dürfen und wie oft. Weitere Einschränkungen müssen nicht ausdrücklich aufgeführt werden, sondern ergeben sich als Konsequenz anderer Regeln, z.B. wenn beim ß_NaB_eintragen verlangt wird, daß alle Konstituenten des Adreßausdrucks bereits definiert sein müssen (s. 3.2.4.1 und 3.3.1).

3.3.1 Die Pseudobefehle

Für den Modulanfang müßte nicht unbedingt ein eigener CODE-Befehl definiert werden: alle Aktionen, die bei seiner Bearbeitung ausgeführt werden, könnten auch gleich bei Start des CGs angestoßen werden. Er wird im Interesse einer einheitlichen Beschreibung von CODE eingeführt.

β _Modulanfang

```

+++++
+ = Befehlsschlüssel :  $\beta\beta$ _Modul_eröffnen +
+++++
*****
* = optcod : "modbeg" *
*****

```

Im folgenden Struktogramm sind die hier relevanten Initialisierungen angegeben:

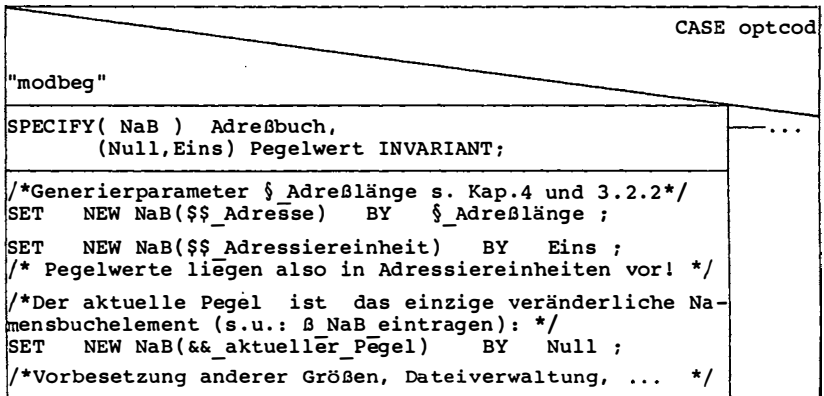


Abb. 3.3.1-1: Aktionen bei CG-Start

Wenn in anderen Konkretisierungen von CODE die Längen der Arithmetiktypen (s. 3.2.2) nicht einheitlich in Bit anzugeben wären, sondern etwa in Tetraden für gepackte BCD-Zahlen und in Byte für ungepackte, dann könnten entsprechende Namensbuch-Elemente vorgesehen werden, für die ebenfalls Vorbelegungen vorzunehmen wären.

Eine andere Erweiterung des β -Modulanfangs in Hinsicht auf die Arithmetiktypen wird im Zusammenhang mit der Adreßbuch-Führung (siehe übernächster Befehl) diskutiert.

Modulanfang bzw. Modulende dürfen nur je einmal (am Beginn bzw. am Schluß) im CODE-Programmtext stehen.

β -Modulende

```

+++++
+ = Befehlsschlüssel :  $\beta\beta$ -Modul abschließen
+++++
*
* = optcod : "modend"
*
*****

```

Am Ende eines CG-Laufs wird das komplette Namensbuch als "Tabelle der modul-lokalen Definitionen" an den Binder übergeben (s.u. Namensbuch-Führung und β -Datencode_erzeugen in 3.3.2).

Namensbuch-Führung

β -NaB_eintragen

```

+++++
+ = Befehlsschlüssel :  $\beta\beta$ -Definiere Pegelwert
+   + Identifikation :  $\beta\beta$ -definierendes Auftreten_
+                       eines Identifikators
+   + Befehlsrestlänge :  $\beta\beta$ -Anzahl der folgenden_
+                       Summanden
+   + Adreßausdruck :  $\beta\beta$ -Folge der Pegelsummanden
+++++
*
* = optcod : "define"
*
*   + name : &&-da_Identor
*
*   + psanz : &&-natürliche_Zahl
*
*   + adrexp : /*Folge der*/  $\beta$ -Pegel_Summanden
*
*****

```

β -Pegel_Summand

```

*****
*
* = einheit : $-Arithmetiktyp /*mit Länge in Bit*/
*   + mult : &&-ganze_Zahl
*
* = einheit : &&-aa_Identor /*angewandtes Auftreten*/
*   + mult : &&-ganze_Zahl
*
*****

```

Damit das Eintragen sofort durchgeführt werden kann, wird gefordert, daß die Identifikatoren in den Pegelsummanden bereits definiert und nicht importiert sind (siehe nächster Befehl). Als Multiplikatoren sind auch negative Ganzzahlen zulässig. Der erste Teil der Produktion soll die Bildung von Adreßdifferenzen für den Zugriff auf Elemente zusammengesetzter Objekte ermöglichen; im zweiten Fall werden die Werte bereits existenter Einträge des Namensbuches verwendet, bei denen es sich sowohl um Adreßdifferenzen als auch um Pegelwerte von Objekten aus dem Bindecode handeln kann.

Mit diesen Mitteln lassen sich in zielrechnerunabhängiger Form auch Adreßausdrücke bilden, die nicht mehr als Objektadressen oder Adreßdifferenzen interpretierbar sind; dies zu vermeiden ist Sache des COTs (siehe auch Typ `$_Adresse` in 3.2.2). Eine sinnvolle Anwendung wäre dagegen die Realisierung statischer Blockstruktur, indem bei der Bildung der Adreßausdrücke für verschiedene Objekte von gleichen Basispegeln ausgegangen wird (s. Abb. 3.3.1-5).

Einmal vorgenommene Namensbuch-Einträge können nicht mehr geändert werden; davon ausgenommen ist nur das Element mit dem vordefinierten Namen `&_aktueller_Pegel` (siehe auch 3.3.2). Weitere vordefinierte Identifikatoren sind `$$_Adressiereinheit` und `$$_Adreßlänge`; alle drei sind bei CG-Start (s.o.: `ß_Modul_öffnen`) geeignet zu initialisieren. Der aktuelle Pegel wird nicht verändert!

Am Ende des CG-Laufs wird das gesamte Namensbuch an den Binder übergeben (als Tabelle der lokalen Definitionen; s.o. `ß_Modulende`), der dann die lokalen Referenzen entsprechend einsetzen kann (s. Abb. 3.3.1-2). So wird erreicht, daß während der CGg keine Vorwärtsreferenzen aufgelöst werden müssen und ein einziger Lauf durch das CODE-Programm genügt, um den Bindecode zu erzeugen.

Es wird keine Kontrolle auf Überschreitung irgendeiner Adreß-Obergrenze durchgeführt; das ist frühestens beim Binden sinnvoll, da die Modulanfangsadresse nicht eher vorliegen kann.

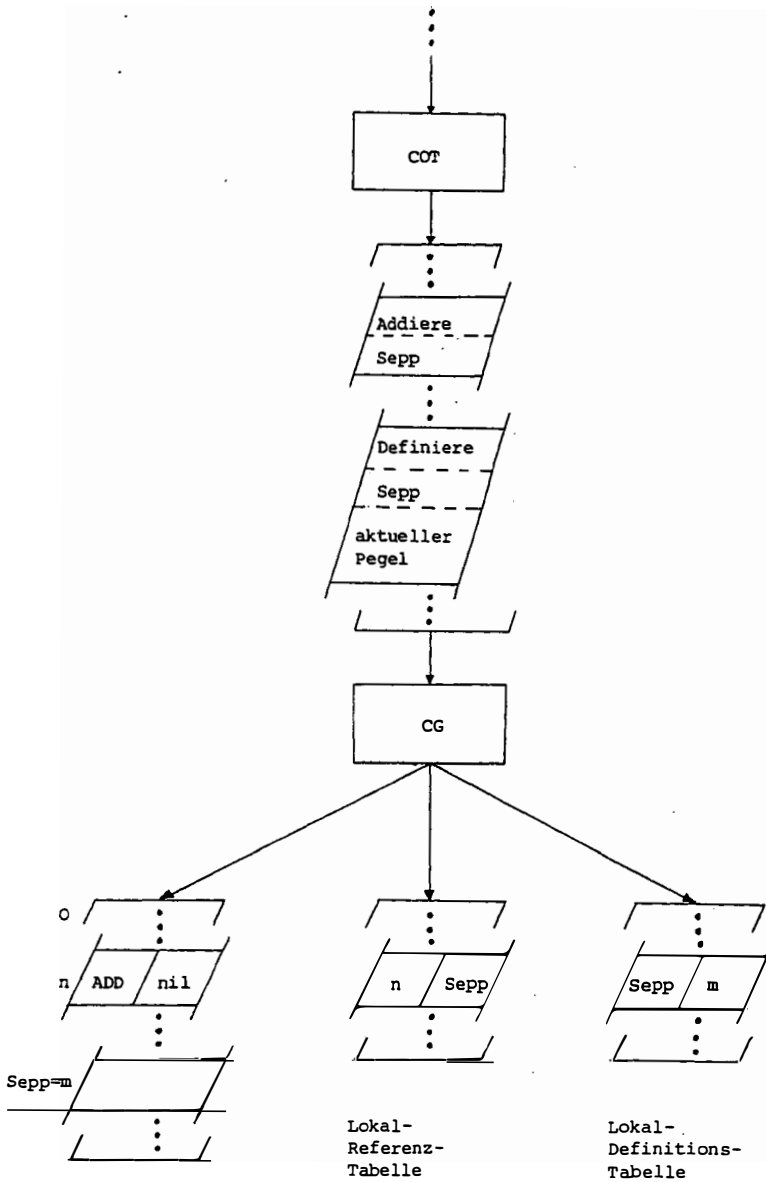


Abb. 3.3.1-2: Lokale Referenzen

Die Bearbeitung dieses Befehls insgesamt sieht folgendermaßen aus:

```

SPECIFY
  (name,psanz,einheit,mult) Befehlsfeldart INVARIANT,
  (NaB) Adreßbuch,
  (Null) Pegelwert INVARIANT,
  (Lesen)
    FUNCTION(Befehlsfeldart)
      RESULT(Befehlsfeldinhalt),
  (Längenumrechnung)
    FUNCTION( UNION OF(Typ,Identifikator) )
      RESULT(Länge_in_Adressiereinheiten);

DECLARE
  (Bezeichner) Identifikator LOCAL,
  (LgEinheit) UNION OF(Typ,Identifikator) LOCAL,
  (Summandanzahl,Multip1) ganze Zahl LOCAL,
  (Länge,Ergebnis) Pegelwert LOCAL;

SAME ARE (Pegelwert,Länge_in_Adressiereinheiten) ;

SET Bezeichner BY CALL Lesen(name) ; /*neuen Namen lesen*/

/***** Auswertung des Adreßausdrucks *****/

SET Summandanzahl BY CALL Lesen(psanz) ;
/*Anzahl der folgenden Pegelsummanden lesen*/

FROM 1 TO Summandanzahl /*Summanden-Bearbeitung*/
  SET LgEinheit BY CALL Lesen(einheit); /*Einheit und*/
  SET Multip1 BY CALL Lesen(mult);/*Multiplikator lesen*/
  SET Länge BY CALL Längenumrechnung(LgEinheit);
  /*Ermittlung der Länge in Abhängigkeit von der Einheit aus
  dem Pegel-Summanden (s. Strukt. 3.3.1-4)*/
  INCREMENT Ergebnis BY Länge * Multip1; /*Summierung*/

SET NEW NaB(Bezeichner) BY Ergebnis ;/*Anlegen des Elements*/

```

Abb. 3.3.1-3: Namensbuch-Eintragung

Die Längenumrechnung, Aufweitung (dritter Teil)

Ist die Einheit in der Pegelkomponente \$\$_Adresse, \$\$_Adressiereinheit, &&_aktueller_Pegel oder ein anderer bereits definierter Identifikator, kann der entsprechende Zielrechner-Pegelwert einfach aus dem NaB entnommen werden; handelt es sich um einen \$_Arithmetiktyp, muß von seiner Länge (in Bit; siehe Anfang von 3.3) auf die nächste Wortgrenze verlängert (siehe Aufweitung in 3.2.4.2) und das Ergebnis in Adressiereinheiten ausgedrückt werden:

DECLARE(Längenumrechnung) FUNCTION(Einheit) RESULT(Pegelwert); BY_VALUE(Einheit) UNION_OF(Typ,Identifikator) ; DECLARE(Ergebnis) Pegelwert LOCAL ;	
CALL QUERY_TYPE(Einheit) EQUAL Identifikator ?	
N	J
SPECIFY(Einheit) Typ STRUCT OF (Darstellung,Länge_in_Bit)	SPECIFY(Einheit)Identifikator;
/* Aufweitung: */ SET Ergebnis TO NEXT MULTIPLE GREATER THAN (\$_Wortlänge, Einheit.Länge) ; /*Umrechnung in Adressiereinheiten :*/ SET Ergebnis BY Ergebnis / \$_AdrEinh_Länge ;	/*Pegelwert aus Namensbuch entnehmen*/ SET Ergebnis BY NaB(Einheit) ;
RETURN(Ergebnis) ; /*Zu \$_AdrEinh_Länge und \$_Wortlänge siehe 3.2.2 und Kap. 4*/	

Abb. 3.3.1-4: Längenumrechnung

Damit bei der CGg die Aufweitung nicht unnötigerweise wiederholt ausgeführt werden muß, "sei dem COT empfohlen", für die jeweils benötigten Arithmetiktypen an den Beginn des CODE-Programms entsprechende β NaB eintragen-Befehle zu legen und anschließend von der Möglichkeit, die Arithmetiktypen selbst anzugeben, keinen Gebrauch mehr zu machen.

Eine verbesserte Aufgabenverteilung

Das Aussprechen dieser Empfehlung, sowie der Umstand, daß in der Längenumrechnung Argumente von unterschiedlicher Art verarbeitet werden müssen, macht einen Entwurfsfehler der bisherigen CODE-Konkretisierung deutlich:

Eigentlich müßten die Adreßbuch-Einträge für die Arithmetiktypen zu den Initialisierungen bei CG-Start (β Modulanfang) verlegt werden:

β Modulanfang mit NaB Vorlauf für Arithmetiktypen

```

+++++
+ = Befehlsschlüssel :  $\beta$  Modul eröffnen
+ + Befehlsrestlänge :  $\beta$  Anzahl der folgenden
+ + + Typdefinitionen
+ + NaB.Vorlauf :  $\beta$  Folge der
+ + + Typdefinitionen
+++++

*****
* = optcod : "modbeg"
*
* + tdanz : && natürliche_Zahl
*
* + nabvorl : /*Folge der*/  $\beta$  Typdefinitionen
*****
 $\beta$  Typdefinition
*****
* = name : && da_Identor
*
* + einheit: $ Arithmetiktyp
*****

```


Bei den Adreßbucheinträgen, die nun bei der Bearbeitung des Befehls "modbeg" zusätzlich vorzunehmen sind, genügt der linke Ast der obigen Längenumrechnung, das Aufweiten. Beim β _NaB_eintragen wird statt einer Längenumrechnung nur noch das "Nachschlagen" im Namensbuch benötigt (und eine Empfehlung erübrigt sich):

β _NaB_eintragen

```
+++++
+/* wie bisher */+
*****
```

β _vereinfachter Pegel Summand

```
*****
* =   einheit : &&_aA_Identor      *
*                                     *
* + mult   : &&_ganze_Zahl         *
* *****
```

Beim Befehl β _Pegel_verstellen (s. 3.3.2) wird nur noch dieser vereinfachte Aufbau von Adreßausdrücken berücksichtigt.

```

:
:
/* Der Wert von &&_aktueller_Pegel sei 123 */
optcod : "define"
name   : "Adam"
psanz  : "1"
einheit: &&_aktueller_Pegel
mult   : "+1"
/*Das NaB wird um ("Adam","123") erweitert*/
optcod : "define"
name   : "Eva"
psanz  : "2"
einheit: &&_aktueller_Pegel
mult   : "+1"
einheit: $$_Adressiereinheit
mult   : "-7"
/*Eintrag: ("Eva","116")*/
optcod : "define"
name   : "Kain"
psanz  : "2"
einheit: "Adam" /* angewandtes Auftreten */
mult   : "-1"   /* ganz legal */
einheit: "Eva"
mult   : "+1"
/*Eintrag: ("Kain","-7")*/
optcod : "define" /*Statische Blockstruktur*/
name   : "Blockbasis"
psanz  : "1"
einheit: &&_aktueller_Pegel
mult   : "+1"
optcod : "define"
name   : "Objekt_2_in_Block_eins"
psanz  : "2"
einheit: "Blockbasis"
mult   : "+1"
einheit: $$_Adresse /*Typ des 1. Objekts*/
mult   : "+5"        /*also 5 Zeiger*/
optcod : "define"
name   : "Objekt_1_in_parallelem_Block_zwei"
psanz  : "1"
einheit: "Blockbasis"
mult   : "+1"
:
:
/*Hier ist auch zu sehen, daß kein Unterschied zwischen
  Objektadressen und Adreßdifferenzen gemacht wird!*/

```

Abb. 3.3.1-5: Beispiele für "define"-Befehle

Extern- und Globalspezifikationen

β_Import

```
*****
* =   optcod : "import"
* - name   : && dA Identor
*****
```

In Adreßausdrücken (Adreßbuch-Führung s.o., Pegelverstellung s. 3.3.2) sind diese Identifikatoren verboten. Sie können angewandt nur im β_Datencode_erzeugen für Adreßobjekte und in den β_Referenzen von β_ausführbaren_Befehlen auftreten (siehe 3.3.2 und 3.3.3). Statt eines Pegelwertes wird eine Externmarkierung ins zugehörige Namensbuchelement geschrieben.

β_Export

```
*****
* =   optcod : "export"
* - name   : && aA Identor
*****
```

Der CG muß den Identifikator in die Externdefinitions-Tabelle (s. Abb. 3.2.4.1-2) eintragen.

Es wäre zwar möglich gewesen, die Befehle für Import und Export als Befehlssteile des β_NaB_eintragens zu entwerfen, doch hätte dies die CGen - wenn überhaupt - nur unwesentlich entlastet, die Beschreibung aber unübersichtlicher gemacht.

3.3.2 Platzreservierung

Pegelerhöhung

Da schon die Adreßausdrücke beim β_NaB_eintragen die Darstellung statischer Blockstruktur erlauben, werden in dieser CODE-Konkretisierung negative Pegelveränderungen nicht zugelassen. Das verlängert die CODE-Programme, weil in den Adreßausdrücken durch den (bzw. die!) ersten Summanden der aktuelle Stand des Pegels innerhalb des Blocks darzustellen ist. Zum Ausgleich werden die CGen dahingehend entlastet, daß nicht zusätzlich zum &&_aktuellen_Pegel ein Bindecodepegel geführt werden muß (s. Abb. 3.3.1-5).

Die einfachste Art, diesen zweiten Pegel zu führen, wäre, für ihn einen weiteren Pegelverstell-Befehl vorzusehen und die Einhaltung von Konsistenzbedingungen dem COT zu überlassen. In der hier vollzogenen Konkretisierung gibt es aber nur einen einzigen. Er ähnelt in seinem Aufbau weitgehend dem Namensbuch-Eintragen aus 3.3.1:

ß_Relative_Veränderung_des_aktuellen_Pegels

```

+++++
=   Befehlsschlüssel : ßß_Pegelverstellung
    + Befehlsrestlänge : ßß_Anzahl_der_folgenden
                        Summanden
    + Adreßausdruck   : ßß_Folge_der_Pegelsummanden
+++++
*****
*   =   optcod : "level"
*
*   + psanz  : &&_natürliche_Zahl
*
*   + adrexp : /*Folge_der*/ ß vereinfachten_Pegel_
*               Summanden
*****

```

ß vereinfachter Pegel Summand

```

*****
*   =   einheit : &&_aA_Identor
*
*   + mult   : &&_ganze_Zahl
*****

```

Es gelten - soweit anwendbar - die gleichen Regeln wie beim ß_NaB-eintragen. Zusätzlich wird ein positives Ergebnis des Adreßausdrucks gefordert.

Innerhalb des Ausdrucks kann auf den zu verändernden aktuellen Pegel lesend zugegriffen werden; alle solchen Zugriffe liefern den alten Wert und erst das Ergebnis des gesamten Ausdrucks wird zum aktuellen Pegel addiert.

Wieder wird nicht geprüft, ob irgendeine Obergrenze überschritten wird: frühestens beim Binden ist es sinnvoll, die Längen der einzelnen Programm-Module zusammenzuzählen und die Summe mit der maximal zulässigen Adresse zu vergleichen.

Das Struktogramm in Abb. 3.3.2-1 zeigt die Bearbeitung des Befehls "level" im CG:

CASE optcod	
"level"	
SPECIFY(Bindecode) Datei ;	...
/* Auswertung des Adreßausdrucks wie in Abb.3.3.1-2 */	
/* Pegelführung: */	
INCREMENT NaB(&&_aktueller_Pegel) BY Ergebnis ;	
/*Verlängerung des Bindecodes um einen Bereich mit undefiniertem Inhalt: */	
EXTEND Bindecode BY UNSPEC(Ergebnis) ;	

Abb. 3.3.2-1: Pegelerhöhung

Platzreservierung mit Vorbelegung

ß_Datencode_erzeugen

```

+++++
=   Befehlsschlüssel   : ß_Datencode
+   Objektart         : ß_Länge_und_Typ_des_
                        Objekts
+   Vorbelegungswert   : ß_Zielrechnerunabhängige_
                        Objektdarstellung
+++++

*****
*   =   optcod : "datcod"
*
*   +   länge   : &&_aA_Identor
*
*   +   typ     : $_Datentyp /*die Länge ist redundant,
*                           nicht nutzlos (s.S.56)*/
*
*   +   datobj  : $ CODE Datenobjekt
*****

```

Beim "datcod"-Befehl wird sowohl die Angabe des Namens verlangt, unter dem im Adreßbuch die Länge (Pegelwert) des Objekttyps zu finden ist (s. "Verbesserte Aufgabenverteilung" in 3.3.1), als auch die Angabe des Typs selbst. Bei dem Identifikator handelt es sich also nicht etwa um das definierende Auftreten eines Bezeichners für das anzulegende Objekt; zur Einführung von Namen dient ausschließlich der "define"-Befehl!

```

$_CODE_Datenobjekt
*****
* = $$_Bitkette /*entweder der Länge % AdreEin Länge
*                für Adressiereinheiten (s. 3.2.2)
*                oder der im $_Datentyp angegebenen
*                Länge*/
*
* = $$_ganze_Zahl
*
* = $$_rationale_Zahl
*
* /* Adreß-Objekt: */
* = $ Adreßart      +      && aA Identor zwei
*****
$_Adreßart
*****
* = "refer"      /* Adresse eines Objekts */
*
* = "differ"     /* Adreßdifferenz */
*****

```

Bearbeitung von "datcod"

Handelt es sich um ein Objekt eines \$_Arithmetiktyps, muß die Konversion in die zielrechnerspezifische Darstellung durchgeführt und der Bindecode des aktuellen Moduls um das so erzeugte Zielobjekt erweitert werden.

Im Fall einer Referenz wird der Bindecode in der entsprechenden Länge (undefinierter Inhalt) erweitert; zusätzlich ist hier noch ein Eintrag in der Extern- bzw. Lokal-Referenztabelle vorzunehmen, je nachdem, ob sich die Referenz auf ein externes Objekt bezieht oder nicht. Das Eintragen des Adreßwerts fällt dem Binder zu (s. "Adressen" in 3.2.2, sowie "modend"- und "define"-Befehl in 3.3.1).

Der Begriff Referenz wurde soeben in einem etwas eingeschränkten Sinn gebraucht, da hier nur ein Adressiermodus zu berücksichtigen ist (s. 3.2.4.3).

In einer Konkretisierung, die konsequent auf einfachste Implementierung abzielt, sollte auch das Einsetzen der Adreßdifferenzen dem Binder übertragen werden. In der CGg ist dann keine Umwandlung von Pegelwerten in die Zielcodedarstellung von Adressen mehr notwendig. Der direkte Anteil von Referenzen \$_ausführbarer_Befehle (s. 3.2.4.3 und 3.3.3) wird genauso zu behandeln sein wie ein "refer"-Datenobjekt.)

Ad hoc wäre eine umgekehrte Aufgabenverteilung zwischen CG und Binder zu erwarten, daß nämlich der CG die Konversion erledigen würde und der Binder ausschließlich Adressen in Zielcodedarstellung zu verarbeiten hätte. Da aber der Binder portabel sein soll, arbeitet auch er noch mit ganzzahlig dargestellten Pegelwerten, die erst beim Einblenden der Referenzen (in den Bindecode) in maschinenspezifische Adreßdarstellung umgeformt werden.

Die Folge ist, daß jetzt der Binder auch Kenntnis von der Adreßart (s.O.) erhalten muß.

Das Struktogramm in Abb. 3.3.2-2 gibt einen Überblick über die "datcod"-Bearbeitung; im Anschluß wird auf die fünf verschiedenen Konversionsfälle, für Adressiereinheiten, Bitketten, ganze Zahlen, rationale Zahlen und zuletzt Adressen, eingegangen.

"datcod"	CASE optcod
<pre> SPECIFY(länge,typ,datobj) Befehlsfeldart INVARIANT, (Bindecode) Datei, (Lesen) FUNCTION(Befehlsfeldart) RESULT(Befehlsfeldinhalt), (Allgemeine Konversion) PROCEDURE(Datentyp, Datenobjekt, Länge in Adressiereinheiten, Zielcodefeld); DECLARE(Ziellänge) Pegelwert LOCAL, (LgName) Identifikator LOCAL, (Objekttyp) Datentyp LOCAL, (Datum) Datenobjekt LOCAL, (Zielcode) Zielcodefeld ARRAY_OF BIT(\$ _AdrEinh _Länge) LOCAL;</pre>	...
<pre> /*Lesen des ersten Namens und Aufsuchen der Objektlänge im Namensbuch: */ SET LgName BY CALL Lesen(länge) ; SET Ziellänge BY NaB(LgName) ;</pre>	
<pre> /*Lesen des Typs und des CODE-Datenobjekts: */ SET Objekttyp BY CALL Lesen(typ) ; SET Datum BY CALL Lesen(datobj) ;</pre>	
<pre> /*Löschen des Arbeitsbereichs: */ ZERO Zielcode(1:Ziellänge) ; /*Erzeugung des Zielcodes durch Aufruf der "Allgemeinen Konversionsprozedur" (s. Abb.3.3.2-3) : */ CALL Allgemeine Konversion (Objekttyp,Datum,Ziellänge,Zielcode) ;</pre>	
<pre> /*Pegelzählung und Bindecode-Ausgabe: */ INCREMENT NaB(&&_aktueller_Pegel) BY Ziellänge ; EXTEND Bindecode BY Zielcode(1:Ziellänge) ;</pre>	

Abb. 3.3.2-2: Grobstruktur der Datencode-Erzeugung

Die "Allgemeine Konversionsprozedur"

Bei Aufruf der Konversion wird als drittes Argument die Länge (in \$\$_Adressiereinheiten!) übergeben, in der das Ergebnis erzeugt werden soll. Dies geschieht im Hinblick auf die Aufweitung, bei der definierter Bindecode erzeugt werden muß (siehe "Keine Vergleiche" in 3.2.4.3).

```

DECLARE( Allgemeine Konversion )
PROCEDURE( Objekttyp,Datum, Ziellänge,Zielcode ) ;
BY_VALUE( Objekttyp )Datentyp
    STRUCT_OF(Darstellung,Länge_in_Bit),
    ( Datum ) Datenobjekt,
    ( Ziellänge ) Länge_in_Adressiereinheiten;
BY_REFER( Zielcode ) Zielcodefeld
    ARRAY_OF BIT($ AdrEinh_Länge);

```

CASE Objekttyp.Darstellung				
\$\$_Adressier einheit	\$\$_Bit			
...	...	\$\$_Ganz		
		...	\$\$_Ratio	\$\$_Adresse
		

Abb. 3.3.2-3: Grobstruktur der "Allgemeinen Konversion"

Die beiden ersten Fälle sind sehr einfach zu behandeln:

CASE Objekttyp.Darstellung	
\$\$_Adressiereinheit	
SPECIFY(Datum) BIT(\$_AdrEinh_Länge) ;	...
/* Da der Objekttyp \$\$_Adressiereinheit ist, muß die Ziellänge gleich eins sein */	
SET Zielcode(Eins) BY Datum ;	

Abb. 3.3.2-4: Ablage von Adressiereinheiten

Hier, wie auch im zweiten Fall, wird vorausgesetzt, daß CODE-Bitkettenobjekte nach dem Lesen im CG in der selben Form vorliegen, wie Bitkettenobjekte, die im CG-Quellprogramm vereinbart werden, und daß bei Zuweisung (SET) in der Länge des kürzeren Transferpartners (Quelle bzw. Ziel) linksbündig übertragen wird.

CASE Objekttyp.Darstellung	
\$\$_Bit	
SPECIFY(Datum) BIT(Objekttyp.Länge_in_Bit) ;	...
SET Zielcode(1:Ziellänge) BY Datum ;	

Abb. 3.3.2-5: "Umwandlung" von Bitketten in Zielform

Aufweiten (vierter Teil)

Daß das Zielcodefeld vorher gelöscht wird, hat zur Folge, daß rechts mit binären Nullen aufgefüllt wird. Diese Konvention macht beim Rechts-Schieben und Komplementieren von Bitketten (s. 3.3.3.1) besondere Vorkehrungen notwendig, damit der Vergleich durch Ganzzahlsabtraktion (s. 3.2.4.3) erfolgen kann.

Konversion ganzer Zahlen

Im Fall von Ganzzahlen taucht zum ersten Mal ein Generierparameter auf, der nicht Stellvertreter einer Zahl ist. Vor der Generierung sehe das CG-Gerüst an dieser Stelle so aus, wie Struktogramm 3.3.2-6 zeigt:

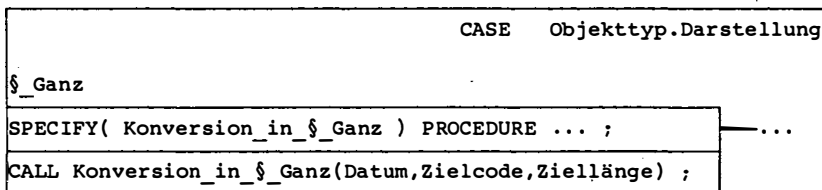


Abb. 3.3.2-6: Aufruf der Konversion ganzer Zahlen

Danach sei der CG-Rahmen z.B. folgendermaßen vervollständigt:

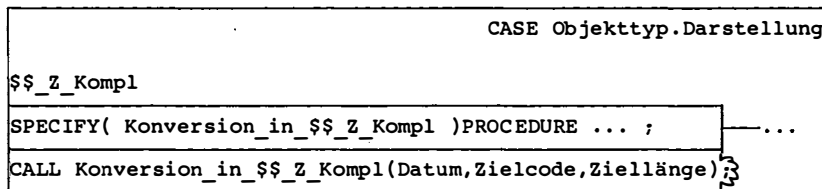


Abb. 3.3.2-7: Aufruf der Ganzzahl-Konversion nach Generierung

Die Wandelroutinen für ganze Zahlen hängen ihrerseits nur noch vom Generierparameter \$_AdrEinh_Länge ab und zwar in der gleichen trivialen Art, wie die allgemeine Konversionsfunktion (s. Struktogramm 3.3.2-3). Der Algorithmus ist jeweils zielrechnerunabhängig programmierbar.

Alle vier Prozeduren gehören zum CG-Gerüst. Bei der Generierung wird die entsprechende ausgewählt und vervollständigt, während die übrigen extrahiert werden.

In einer Erweiterung des vorgeschlagenen Konzepts könnte bei Generierung zunächst die Anzahl der gewünschten Ganzzahl-Darstellungsarten angegeben werden. Der Generator müßte die entsprechende Anzahl CASE-Zweige im Gerüst einsetzen, wonach Zweig für Zweig das bisherige Verfahren sinngemäß zur Anwendung käme.

Der CASE-Zweig für rationale Zahlen

Dieser Ast ist analog zur Ganzzahlalternative (nach der Generierung). Es sei deshalb nur auf "Rationale Zahlen" in 3.2.2 und in Kap. 4 verwiesen. Damit verbleibt noch der fünfte Zweig, die Bearbeitung von Adressen, zu behandeln.

Die Bearbeitung von Adreßobjekten

Im Abschnitt "Bearbeitung von 'datcod'" (s.O.) wurde schon festgestellt, daß das Umformen von Adreßdifferenzwerten in Zieldarstellung dem Binder übertragen werden kann, wenn diesem die `$_Adreßart` mitgeteilt wird. Werden dann noch die Tabellen der Extern- und Lokalreferenzen (s. Abb. 3.2.2-1 und 3.2.4.1-1) zu einer Tabelle vereinigt, deren Einträge die in Abb. 3.3.2-8 skizzierte Form haben,

⋮			
/ n /	"Peter"	/ "refer" /	
/ m /	"Abstand"	/ "differ" /	
⋮			

Abb. 3.3.2-8: Einträge in der Referenztabelle

dann können alle Adreßobjekte im CG so behandelt werden, wie in Abb. 3.3.2-9 gezeigt:

CASE Objekttyp.Darstellungsart	
\$\$_Adresse	
<pre> SPECIFY(Datum) Adreßobjekt STRUCT_OF(Adreßart,Identifikator), (Referenztabelle) Datei; </pre>	...
<pre> /*Erzeugung undefinierten Bindecodes: */ SET Zielcode(1:Ziellänge) BY UNSPEC(Ziellänge) ; /*Eintragung in die Referenztabelle: */ EXTEND Referenztabelle BY(NaB(&& aktueller Pegel), Datum.Identifikator, Datum.Adreßart); </pre>	

Abb. 3.3.2-9: Übersetzung von Adreßdaten im CG

Bemerkung 1

Es sind nun CGen denkbar, die nur Pseudobefehle und Platzreservierungen bearbeiten. Es wird sich zeigen, daß sie z.B. für die Erzeugung der \$\$_Speicherregister (s. 3.2.1) eingesetzt werden können (s. Kap. 4).

Bemerkung 2

Die Darstellung der Operationscodes und die Wortwahl "Name", "Bezeichner" usw., könnten zu einem Mißverständnis führen: es wird nicht vorgeschlagen, CODE-Programme Übersetzerintern in Zeichenform darzustellen; die Zeichenform wurde hier aus Darstellungsgründen gewählt und um zu demonstrieren, daß sich CODE auf einfache Weise mnemotechnisch darstellen läßt.

Es ist vielmehr beabsichtigt, alle Befehlsfeldinhalte als ganze Zahlen zu verschlüsseln, um die Verzweigungsstruktur der CGen einfach implementieren zu können (siehe besonders "Befehlsentschlüsselung" in 3.3.3).

3.3.3 Konkretisierung der ausführbaren Befehle

3.3.3.1 Grobeinteilung

Der direkte Weg führt nicht zum Ziel

Bei der Übersetzung eines ausführbaren Befehls soll der CG zunächst versuchen, Zielcode anzulegen, der speziell für die vorliegende Kombination aus CODE-Befehlsschlüssel und (je nach Befehl) Schieberichtung, Schiebeschrittzahl, Sprungbedingung, Operationstyp und Adressiermodus geeignet ist. Die Muster solcher Zielcodes in die Daten des CGs einzubringen, wird die Hauptaufgabe des Implementierers bei der Generierung sein. (Da ein Zielcode i.a. nicht nur aus einem einzigen realen Maschinenbefehl bestehen kann, wird hier auch von "Zielcodesequenz" oder einfach "Sequenz" gesprochen.)

Im weiteren werden CODE-Befehle als verschieden bezeichnet, wenn sie sich in mindestens einem ihrer Befehlsfelder unterscheiden. Würde nun einfach jeder Befehlsfeld-Kombination ein eigenes Sequenz-Muster zugeordnet (etwa mit dem Ziel, die Fähigkeiten der realen Maschine weitestgehend zu nutzen), ergäbe sich selbst dann eine viel zu große Zahl von Mustern, wenn für die Längen der CODE-Typen vernünftige Obergrenzen festgelegt würden.

Das gegenteilige Extrem wäre, einheitlich für jeden Befehl den Aufruf einer entsprechenden Bibliotheksroutine abzusetzen und ihr die Auswertung der Information zu überlassen, die in den CODE-Befehlen steckt. Der Übersetzer würde dadurch weitgehend zu einem Interpretierer werden, dessen Ausführungs-Komponenten je nach Quellprogramm aus der Bibliothek des Übersetzers/Interpretierers neu zu binden wären.

Um die Konsequenzen beider Extreme zu vermeiden, wird die Menge aller verschiedenen Befehle in Gruppen eingeteilt, für deren Mitglieder jeweils einheitliche Bearbeitung möglich ist. Die Anzahl dieser Gruppen wird möglichst niedrig gehalten, um eine einfache Verzweigungsstruktur der CGen zu erreichen.

Die magischen Begriffe Wortlänge und Adresse

Eine naheliegende und wirksame Vereinfachung liegt darin, alle Befehle für Operanden, die länger als ein Zielrechnerwort sind, in Aufrufe von Bibliotheksprogrammen zu übersetzen. Die Voraussetzungen dafür sind durch die "Aufweitung" (s. 3.2.4.3 und "Längenumrechnung" in 3.3.1) bereits geschaffen. Folgende Umstände dürfen dabei allerdings nicht übersehen werden:

- a) Bei 8-Bit-Prozessoren ist abzusehen, daß die Ausführung der numerischen Operationen schon weitgehend interpretative Züge erhalten wird (s.o.). Immerhin - unter Hinweis auf die Mitverantwortung des etwas kümmerlichen Prozessoraufbaus und auf die Tatsache, daß wenigstens einige andere Laufzeitbelastungen des Interpretierens (Namensbuchführung, Typprüfungen, ...) entfallen, kann diese negative Auswirkung noch hingenommen werden.
- b) Auch darauf, daß es für etliche Befehle, wie Addition, Komplementierung usw., einfache Zielcodesequenzen für Mehrwortoperationen gäbe, deren Berücksichtigung aus Effizienzgründen erwünscht wäre, wird im Rahmen dieses Konzepts im Interesse eines möglichst einfach implementierbaren Generatorprogramms keine Rücksicht genommen (siehe auch Kap. 1).
- c) Bei Adreß-Befehlen wird allerdings nicht mehr so rigoros verfahren, da bei der Definition des Datentyps `$_Adresse` ohnehin schon etliche Einschränkungen gemacht wurden (s. 3.2.2).

Werden Algorithmen in höheren Programmiersprachen realisiert, kommen häufig Zugriffe auf Komponenten zusammengesetzter Objekte /SCHN80/ vor, wobei Teile der Objektadresse zur Zeit der Programmausführung berechnet werden müssen; das Musterbeispiel dafür ist die Bezugnahme auf Feldelemente in Laufschleifen.

Obwohl Adressen bei 8-Bit-Rechnern üblicherweise doppelte Wortlänge haben (ausgenommen Sonderformen wie kurze Adressen, Register-Adressen usw.) stehen für ihre Verarbeitung immer einige Maschinenoperationen zur Verfügung; auch bei Rechnern größerer Wortlänge gibt es hier häufig Sonderbefehle, die einen effizienteren Umgang mit Adressen erlauben sollen.

Um dem CG-Implementierer die Möglichkeit zu geben, auf das Aufwandsverhältnis zwischen Ermittlung der Operandenadresse und Ausführung der jeweiligen Operation Einfluß zu nehmen, werden Adreßbefehle sowohl bei der Erstellung der CGen als auch im CG getrennt zu bearbeiten sein; bei den übrigen Befehlen aber wird alles "abgehackt, was über die \$_Wortlänge herausragt".

Damit spaltet die Übersetzung ausführbarer Befehle in den CGen im wesentlichen in 2 Äste auf: einerseits von Adreßbefehlen und Befehlen, deren Operanden in einem einzigen Zielrechnerwort darstellbar sind, andererseits die Übersetzung von "Mehrwortbefehlen". Bevor jedoch auf die Ablaufstruktur der Codegeneratoren und einige Anforderungen an die Generierung eingegangen wird, müssen in der Festlegung von CODE noch die entsprechenden Vorkehrungen getroffen werden.

Eine weitere Verkleinerung des Befehlssatzes:

Zunächst wird die Anzahl der CODE-Befehle selbst vermindert (vgl.3.2.4.3):

ß_Transfer

```

+++++
= optcod:"lade_ak"   typ:$_Akkutyp  opd:ß_ref
= optcod:"spei_ak"   typ:$_Akkutyp  opd:ß_ref
= optcod:"ak_in_ix"
= optcod:"lade_ix" /*$_Adresse*/  opd:ß_ref
+++++

```

Das Laden des Indexregisters wäre zwar durch Laden in den Akkumulator und anschließenden Transfer darstellbar, aber die Argumentation bei Punkt c) (s.o.) zeigt, daß davon kein Gebrauch gemacht werden sollte.

B_ref

```

+++++
=   amo: "d" /*direkt*/   +   name: &&_aA_Identor
=   amo: "di" /*indiziert*/ +   name: &&_aA_Identor
=   amo: "i" /*indirekt*/
+++++

```

B_Numerik

```

+++++
=   opc : B_num_op
+   typ : $_Akkutyp
+   opd : B_ref
+++++

```

B_num_op

```

+++++
=   "+" = "-" = "*" = "/"
+++++

```

B_Bitketten Verarbeitung

```

+++++
=   opc:"und"   +   typ:$_Bittyp +   opd:B_ref
=   opc:"i_oder" +   typ:$_Bittyp +   opd:B_ref
=   opc:"x_oder" +   typ:$_Bittyp +   opd:B_ref
=   opc:"kplem" +   typ:$_Bittyp
=   opc:"sch_l" +   scz:B_sch_zahl + typ:$_Bittyp
=   opc:"sch_r" +   scz:B_sch_zahl + typ:$_Bittyp
+++++

```

B_sch_zahl

```

+++++
=   &&_positive_ganze_Zahl
+++++

```

B_Sprung

```

+++++
=   opc:"rückspr" /*$$_Adresse*/
=   opc:"up_spr"  /*$$_Adresse*/ + nam:&&_aA_Identor
=   opc:"spr"     /*$$_Adresse*/
+   bed:B_beding + opd:B_ref
+++++

```

B_beding

```

+++++
=   "fel" = "neg" = "nul"
+++++

```

Hier sind zwei Konkretisierungsschritte zusammengefaßt (vgl. 3.3.1 und 3.3.2), da davon ausgegangen werden kann, daß die Bedeutung der Befehle nach den Ausführungen in 3.2.4.3 hinlänglich bekannt ist. Zu sehen ist auch, daß besonders bei den Sprüngen weitere Verluste - sowohl an Funktion als auch an Orthogonalität - zu beklagen sind; weitere Abstriche werden daher nicht mehr vorgenommen.

Aufweitung (letzter Teil)

Bitketten werden im Rahmen des Erzeugens von Datencode rechts mit Nullen auf die nächste Wortgrenze verlängert (siehe 3.3.2). Bei Komplementbildung und Rechtsverschieben wird die Information in diesen Erweiterungen nur in Spezialfällen erhalten bleiben. Bevor nach solchen Operationen Vergleiche in Form ganzzahliger Subtraktionen durchgeführt werden können, müssen die Operanden durch entsprechende Maskierungen wieder normiert werden.

Diese Pflicht kann (wieder einmal) dem COT aufgebürdet werden: die notwendigen Masken können mit dem "datcod"-Befehl angelegt werden und es kann offen bleiben, ob der COT diese Normierungen

- nach jedem Komplement- und Rechtsschiebe-Befehl,
- vor jedem Vergleich oder
- in optimierender Weise je nach Bedarf ins CODE-Programm einfügt.

3.3.3.2 Umgruppierung und Redundanzserhöhung

Die nächste Aufstellung der ausführbaren Befehle ist keine reine Konkretisierung sondern eine Umgruppierung und Erweiterung der vorhergehenden: die Adreßbefehle werden von den übrigen getrennt, die von nun an als "Wortbefehle" bezeichnet werden. Um diejenigen Wortbefehle, deren Operanden länger als ein Zielrechnerwort sind ("Mehrwortbefehle" im Gegensatz zu "Einwortbefehlen"), einfach identifizieren zu können, werden Umstellungen und Ergänzungen innerhalb der Befehle vorgenommen.

1) Die Angabe der Operandencodierung, d.h. wie der Operand bei Befehlsausführung zu interpretieren ist, wird in einigen Fällen in den Befehlsschlüssel aufgenommen:

bisherige Befehle	werden ersetzt durch
"lade_ak"	"l adr_ak" für Adressen und "l rgb_ak" für Objekte von einem der Arithmetiktypen (s.u. Anmerkung 1)
"spei_ak"	"s_adr_ak", "s_rgb_ak" (dito)
"+", "-", "*"	"+_adr", "-_adr", "*_adr", "+_rat", "-_rat", "*_rat", "+_gnz", "-_gnz", "*_gnz"
"/"	"/_rat", "/_gnz"

2) Im Typfeld erscheinen keine CODE-Typen mehr (außer \$\$_Adresse), sondern nur noch jene Bezeichner, die im Namensbuch-Vorlauf (siehe "Verbesserte Aufgabenverteilung" in 3.3.1) definiert werden; es handelt sich also um Information über die Operandenlänge. Außerdem erhalten ausnahmslos alle ausführbaren Befehle so ein Typlängen-Feld.

β_ausführbarer Befehl

```

*****
* = β Adreß_Befehl
*
* = β Wort_Befehl
*****

```

β Adreß-Befehl

```

*****
/*Transfer*/
*
* = opc:"lade_ix" + tlg:$$_Adresse + opd:β_ref
*
* = opc:"l_adr_ak" + tlg:$$_Adresse + opd:β_ref
*
* = opc:"s_adr_ak" + tlg:$$_Adresse + opd:β_ref
*
* = opc:"ak_in_ix" + tlg:$$_Adresse + amo:"d"
*
*
/*Sprung*/
*
* = opc:"springe" + tlg:$$_Adresse + opd:β_ref
*
* = opc:"spr_fel" + tlg:$$_Adresse + opd:β_ref
*
* = opc:"spr_neg" + tlg:$$_Adresse + opd:β_ref
*
* = opc:"spr_nul" + tlg:$$_Adresse + opd:β_ref
*
* = opc:"up_spr" + tlg:$$_Adresse + amo:"d"
*
* + nam:&&_aA_Identor
*
* = opc:"rückspr" + tlg:$$_Adresse + amo:"d"
*
*
/*Adreß-Numerik*/
*
* = opc:β_a_n_op + tlg:$$_Adresse + opd:β_ref
*
*****
β_a_n_op
*****
* = "+_adr" = "-_adr" = "*_adr"
*
*****
β_ref
*****
* = amo:"d" + nam:&&_aA_Identor
*
* = amo:"di" + nam:&&_aA_Identor
*
* = amo:"i"
*
*****

```

3) Wie zu sehen ist, erhalten die operandenlosen Befehle (genau genommen die nicht referenzierenden Befehle) ein redundantes Adressierartenfeld; Sprungbedingungen und Schieberichtungen sind nun in die jeweiligen Befehlsschlüssel integriert.

β_Wort_Befehl

```
*****
/*Transfer*/
= opc:"l_rgb_ak" + tlg:&&aA_Identor + opd:β_ref
= opc:"s_rgb_ak" + tlg:&&aA_Identor + opd:β_ref
/*Numerik*/
= opc:β_rg_n_op + tlg:&&aA_Identor + opd:β_ref
/*Bitkettenverarbeitung*/
= opc:"und" + tlg:&&aA_Identor + opd:β_ref
= opc:"i_oder" + tlg:&&aA_Identor + opd:β_ref
= opc:"x_oder" + tlg:&&aA_Identor + opd:β_ref
= opc:"kplem" + tlg:&&aA_Identor + amo:"d"
= opc:"sch_r" + scz:&&positive_ganze_Zahl
+ tlg:&&aA_Identor + amo:"d"
= opc:"sch_l" + scz:&&positive_ganze_Zahl
+ tlg:&&aA_Identor + amo:"d"
*****
β_rg_n_op
*****
= "+_g" = "-_g" = "*_g" = "/_g"
= "+_r" = "-_r" = "*_r" = "/_r"
*****
```

Anmerkung 1

Bei den Laden-/Speichern-Befehlen wird also (für Operanden gleicher Länge) nicht mehr zwischen den Arithmetiktypen unterschieden! Es wird somit vorausgesetzt, daß der CG-Implementierer in der Lage ist, jeweils ein einziges Zielcodemuster (s. 3.3.3.1, erster Absatz) anzugeben, das den Befehl für alle Arithmetiktypen zusammen verwirklicht. In der Praxis stellt diese Voraussetzung aber keine Einschränkung dar.

Anmerkung 2

Hätte die Normierung der Bitketten (s. 3.3.3.1 "Aufweitung (letzter Teil)") nicht auf den COT abgewälzt werden können, dann wäre in den Befehlen "kplem" und "sch_r" ein zusätzliches Feld für die Angabe der Kettenlänge notwendig geworden; vom CG-Implementierer hätte des weiteren verlangt werden müssen, die Codemuster für diese Befehle so aufzubauen, daß die Normierung gewährleistet wäre, und überdies wären die Optimierungsmöglichkeiten drastisch verschlechtert worden.

3.3.3.3 Übersetzung von Adreß- und Einwortbefehlen

Im Hinblick auf den Aufwand (gemessen an der Anzahl der notwendigen Codesequenzmuster) den eine CG-Generierung, bezogen auf die ausführbaren CODE-Befehle, erfordern wird, ergibt sich nun folgende Bilanz:

Befehls- schlüssel	Anzahl	Bemerkung
lade_ix	3	ein Muster je Adreßmodus
l_adr_ak	3	"-
s_adr_ak	3	"-
ak_in_ix	1	
springe	3	"-
spr_fel	3	"-
spr_neg	3	"-
spr_nul	3	"-
up_spr	1	
rückspr	1	
ß_a_n_op	9	drei je Adreßmodus
bis hier: 33 Adreßbefehle		
l_rgb_ak	3	eines je Adreßmodus
s_rgb_ak	3	"-
ß_rg_n_op	24	acht je Adreßmodus
und	3	eines je Adreßmodus
i_oder	3	"-
x_oder	3	"-
kplem	1	
sch_r	1	
sch_l	1	
dazu 42 Wortbefehle		

insgesamt 75 Adreß- und Einwort-Befehle
(+ n Mehrwort-Befehle)

Anmerkung:

Für die beiden Schiebefehle wird jeweils nur ein Muster gezählt; bei Generierung muß also jeweils ein einziges Muster angegeben werden können, in dem die Anzahl der Schiebeschritte (ggf. nach Konversion; s. 4.2.2) eingeblendet werden kann.

Auf der Grundlage dieser Bilanz wird nun festgelegt, daß zur Implementierung der Adreß- und Einwort-Befehle vom Bediener bzw. Anwender des Generatorprogramms im wesentlichen 75 Codesequenz-Muster angegeben werden müssen (in 3.3.3.4 wird dieser Satz um weitere drei vergrößert werden) und im CG-Rahmen wird folgende grobe Ablaufstruktur für die Bearbeitung der ausführbaren CODE-Befehle verankert:

Operandenlänge größer als §_Wortlänge ?	
N	J
/*Einwort- oder Adreß-Befehl*/ "Befehlsdecodierung" (Auswahl des entsprechenden Zielcodemusters)	/*Mehrwort-Befehl*/ Absetzen eines Aufrufs einer Bibliotheksroutine (siehe 3.3.3.4)
Aufbau des Zielcodes anhand des Musters (s. 4.2.1 und 4.2.2)	
Erweiterung des Bindecodes und der Referenztabelle	
/*Operandenlänge = Typlänge * §_AdrEinh_Länge ; zur Typlänge siehe 3.3.3.2 */	

Abb. 3.3.3.3-1: Grobstruktur des CG-Ablaufs bei Übersetzung ausführbarer Befehle

Die sogenannte Befehlsdecodierung

Für jeden Einwortbefehl muß anhand der Information aus den Befehlsfeldern das zugehörige Codesequenz-Muster ausgewählt werden. Um diese Entschlüsselung möglichst einfach zu gestalten, wird für die Operationsschlüssel und Adressierarten (opc- bzw. amo-Feld) eine positiv-ganzzahlige Darstellung durch die zwei folgenden Tabellen definiert:

Adressierungsart	Zahlenschlüssel
direkt bzw. nicht-referenz.	0
indirekt	1
indiziert	2

CODE-Bef.- Schlüssel	Zahlen- Schl.	Bemerkungen	
+_g	1	Wortbefehle	Adreßmodus = 0, 1 oder 2
-_g	4		
*_g	7		
/_g	10		
+_r	13		
-_r	16		
*_r	19		
/_r	22		
und	25		
i_oder	28		
x_oder	31		
l_rgb_ak	34		
s_rgb_ak	37		
+_a	40	Adreßbefehle	
-_a	43		
*_a	46		
l_adr_ak	49		
s_adr_ak	52		
lade_ix	55		
springe	58		
spr_fel	61		
spr_neg	64		
spr_nul	67		
up_spr	70	Adreßbefehle	Adreßmodus=0
rückspr	71		
ak_in_ix	72		
kplem	73	Wortbefehle	
sch_r	74		
sch_l	75		

Abb. 3.3.3-2: Schlüsseltabellen für Op.-Code und Adreßmodus

Bemerkung:

Sollten mehrere Ganzzahlcodierungen zu berücksichtigen sein (siehe "Konversion, ganzer Zahlen" in 3.3.2), müßte die 2. Tabelle entsprechend erweitert werden.

Die Summe der beiden Schlüssel ergibt für die Einwort- und Adreß-Befehle die Zahlen zwischen Eins und 75. Sie wird im folgenden "Zielcode-Index" genannt. Diese Addition stellt den wesentlichen Teil der Befehlsentschlüsselung in CODE dar.

Da die Zielcodemuster verschieden lang sein können, bietet sich für sie im CG eine Datenstruktur mit folgendem Aufbau an:

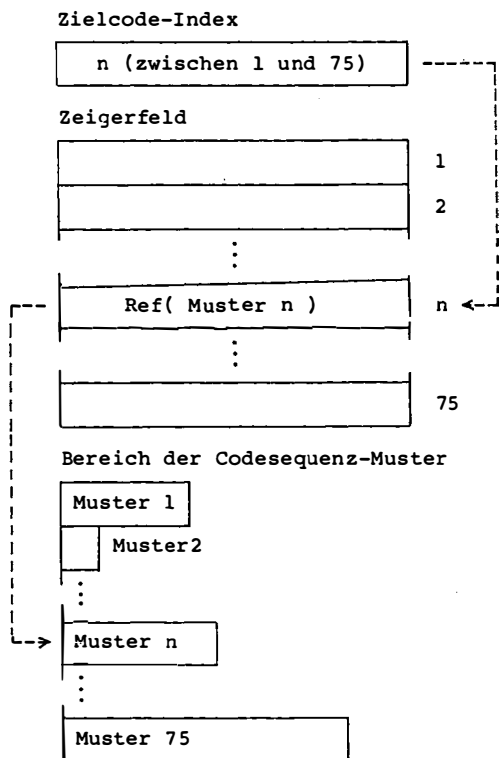


Abb. 3.3.3.3-3: Zugriff auf die Zielcodemuster

Zwei Parameter der Codemuster

In der letzten Syntax der ausführbaren Befehle ist zu sehen, daß nur noch

- 1) die Information aus dem Namenfeld und
- 2) die Schiebezahl

(soweit jeweils vorhanden) nicht in den Zielcode-Index eingehen. Da diese Information einerseits zur Generierzeit nicht bekannt ist, andererseits aber zur Übersetzung der CODE-Programme benötigt wird, um Bindecode und Referenztabelle korrekt erzeugen zu können, müssen in den Mustern Zugriffe auf diese Information (analog zur Verwendung von Parametern in Prozedur- oder Makro-Rümpfen) möglich sein; erst dann kann sie an definierter Stelle in den Bindecode bzw. die Referenztabelle (siehe Abb. 3.3.2-8) eingesetzt werden.

Damit stehen aber erst 2 mögliche Parameter von Zielcodesequenz-Mustern fest; sie dienen dem Zugriff auf Information aus den CODE-Befehlen zur Generierzeit. In 4.2.1 und 4.2.2 werden weitere Parameter einzuführen sein, die dann im wesentlichen dazu dienen werden, auf CG-interne Daten (z.B. &&_aktueller_Pegel) Bezug zu nehmen. Dort werden dann auch -Struktur der Muster und -Anforderungen an ihre Verarbeitung in CG und Generatorprogramm erläutert.

3.3.3.4 Bearbeitung von Mehrwortbefehlen

Es bleibt somit noch zu untersuchen, was im Fall von Mehrwortbefehlen (siehe Abb. 3.3.3.3-1) zu tun ist.

Als erstes muß sichergestellt werden, daß ein Muster für die Erzeugung eines Aufrufs (nämlich einer entsprechenden "Mehrwort-Routine") überhaupt existiert.

Das 76. Zielcode-Muster

Dafür böte sich zunächst an, das gleiche Muster zu nehmen, wie für den CODE-Unterprogrammsprung (opc:"up_spr"). Um aber nicht Gefahr zu laufen, entweder die Aufrufschnittstelle zwischen CG und Laufzeitunterstützung unnötig einzuschränken, oder den Unterprogrammsprung zu überladen, wird ein weiteres Codesequenz-Muster "Aufruf einer Mehrwort-Routine" eingeführt. Dies geschieht in Form der Angabe eines weiteren ausführbaren Befehls:

β ausführbarer Befehl

```

*****
*
* = ... /* siehe 3.3.3.2 */
*
* /* Aufruf einer Mehrwort-Routine:
* =   opc:"lib_call"
*
*   + amo:"direkt"
*
*   + nam:&&_MwR_Identor
*
*       (siehe später Abb. 3.3.3.4-1)
*
*   */
*
*****

```

Ins Zwischenspracheprogramm darf der COT solche Befehle nicht einsetzen (deshalb die Kommentarschreibweise). Die Darstellung als CODE-Befehl wird im Hinblick auf die Anleitung des CG-Implementierers gewählt, der diesen Befehl durch Angabe eines Zielcode-Musters verwirklichen muß.

Um aus diesem Muster und (der Information aus) dem CODE-Befehl Zielcode erzeugen zu können, muß der CG

a) anhand von Befehlsschlüssel, Typlänge und Adressiermodus die Routine identifizieren, für die ein Aufruf abgesetzt werden muß, die aufgebaute Identifikation an den Binder weiterreichen und

b) dafür sorgen, daß die Information aus einem eventuellen Namen- bzw. Schiebeschrittzahl-Feld des CODE-Befehls der Routine nach erfolgtem Aufruf zur Verfügung steht.

Beide Aufgaben sind mit den bisher eingeführten Mechanismen nicht sinnvoll lösbar. Im Fall b) ist das klar, da es in CODE keine Parameterübergabe gibt und CODE-Register und -Speicher dafür nicht ohne weiteres eingesetzt werden dürfen. Die in a) angesprochene Identifikation kann kein üblicher CODE-Bezeichner sein, sonst müßte vom COT verlangt werden,

erstens durch Aufweitung die Typlängen zu ermitteln,
zweitens die notwendigen Mehrwortroutinen herauszufinden, die für das aktuell zu übersetzende Programm benötigt werden,
drittens die zugehörigen "import"-Befehle im CODE-Programm einzusetzen und

viertens die entsprechenden Einträge in der Import-Tabelle (siehe Abb. 3.2.4.1-1) vorzunehmen.

Voraussetzung für all dies wäre noch, daß der COT in Abhängigkeit von `$_Wortlänge` und `$_AdrEinh_Länge` generiert würde; dies aber sollte auf jeden Fall vermieden werden.

Doch noch Extern-Bezeichner

Um das Problem a) zu lösen wird daher doch eine Art externer Namen in CODE eingeführt (vgl. "Import", "Export" in 3.2.4.1). Für diese Identifikatoren ist aber keine Namensbuchführung erforderlich; sie können bei jeder Bearbeitung eines Mehrwortbefehls von neuem (und in trivialer Weise; s.u.) aus dem Inhalt der drei erwähnten Befehlsfelder aufgebaut und sofort in die Referenztabelle eingetragen werden. Anschließend werden sie nicht mehr benötigt.

Eine mögliche Struktur dieser Einträge ist in Abb. 3.3.3.4-1 skizziert (vgl. auch Abb. 3.3.2-8):

Referenztablelle	
	:
/ k / (opc:"und", tlg:"bit17", amo:"d") / "refer" /	
	:

Abb. 3.3.3.4-1: Einträge für Aufrufe von Mehrwortroutinen

Anmerkung:

Anstatt einer neuen Form von Einträgen in der bisherigen Referenz-Tablelle (siehe "Die Bearbeitung von Adreßobjekten" in 3.3.2), könnte auch einfach eine weitere Referenztablelle eingeführt werden, in der nur noch Einträge der neuen Art auftauchen.

Die Zielcode-Muster 77 und 78

Im vorgestellten Befehl "Mehrwortroutinen-Aufruf" wird natürlich nur auf die Routine selbst Bezug genommen. Die Information aus einem eventuellen Namen- bzw. Schiebeschrittzahl-Feld des CODE-Befehls ist bisher noch nicht berücksichtigt (siehe oben, Abschnitt "76. Zielcode-Muster", Punkt b)). Sie wird der Mehrwortroutine als Argument übergeben; diese "verkappte" Parameterübergabe wird durch zwei weitere Befehle dargestellt, für die bei Generierung entsprechende Codesequenz-Muster anzugeben sind. Zusätzlich wird die CODE-Maschine um zwei Register erweitert, das

%%_Operandenadreß_Register_für_MwR_Aufrufe

und das

%%_Schiebeschrittzahl_Register_für_MwR_Aufrufe.

Beide Befehle und Register sind - nach bewährtem Muster - dem COT verborgen:

β_ausführbarer Befehl

```
*****
** = ... /*siehe 3.3.3.2*/
**
**      /*Lade-unmittelbar Operandenadreßregister für
**      Mehrwortroutinen-Aufrufe:
**
** =   opc:"l_mwr_oarg"
**     + amo:"unmittelbar"
**     + nam:&&_aA_Identor   (Name aus dem Mehrwortbefehl)
**
**      Lade-unmittelbar Schiebeschrittzahl-Register
**      für Mehrwortroutinen-Aufruf:
**
** =   opc:"l_mwr_sszr"
**     + amo:"unmittelbar"
**     + scz:&&_positive_ganze_Zahl   (auch aus dem Mehr-
**                                     wortbefehl)
**
**     */
*****
```

Analog für den Registersatz:

%_Registersatz_des Prozessors

```
*****
** = ... /*siehe Abb. 3.2.1-3*/
**
**      /*
** =   %%_MwR_Operandeadreß_Reg
**
** =   %%_MwR_Schiebeschrittzahl_Reg
**
**      */
*****
```

Es wird dem CG-Implementierer dringend empfohlen, diese neuen Register als %%_Speicherregister zu verwirklichen (siehe "CODE-Register und Prozeßwechsel" in 3.2.1 und siehe 4.1.3).

Verfeinerter Ablauf

Mit den nun zur Verfügung stehenden Hilfsmitteln läßt sich der rechte Zweig des Struktogramms 3.3.3.3-1 etwas detaillierter darstellen:

/*Mehrwortbefehl*/

mit Schiebeschrittzahl- oder Namen-Feld	
N	J
?	
	Parameterversorgung für die Mehrwortroutine erzeugen (siehe unten)
Aufbau der Zielcodesequenz für Mehrwortroutinen-Aufruf aus Muster 76	
Erweiterung des Bindecodes und der Referenztabelle	

/*Parameter-Versorgung der Mehrwortroutine aufbauen*/

J	mit Namenfeld ?	N
Aufbau Zielcode aus Muster 77 (für "l_mwr_oarg"-Befehl)	Aufbau Zielcode aus Muster 78 (für "l_mwr_sszz"-Befehl)	
Erweiterung des Bindecodes und der Referenztabelle		

Abb. 3.3.3.4-2: Bearbeitung von Mehrwortbefehlen

Schlußbemerkung zu 3.3.3.4

Mit der Einführung der beiden letzten Sonderbefehle wird erreicht, daß im Fall von Mehrwortbefehlen Schiebeschrittzahl- bzw. Namen-Feld genauso behandelt werden können, wie bei Einwortbefehlen. Damit ist die Konstruktion der CODE-Maschine abgeschlossen und gleichzeitig eine wichtige Voraussetzung für das Vorgehen in 4.2.1 geschaffen; dort und in 4.2.2 werden die Elemente geschildert, aus denen die Aktionen aufgebaut sind, die hier summarisch mit "Erzeugung von ..." bzw. "Aufbau von ..." angesprochen werden.

4. Das Generatorprogramm

Das Generatorprogramm hat im wesentlichen die Funktion, den CG-Rahmen, der markierte Lücken aufweist, so zu komplettieren, daß ein übersetzbares CG-Programm entsteht. Die erwähnten Lücken definieren die Generierparameter; die entsprechenden Argumente erhält der Generator vom CG-Implementierer. Der Generiervorgang ist also einfach ein Edierlauf auf der Quellsprachebene des CG-Programms.

In den Kapiteln 3.3.1 bis 3.3.3 war schon zu erkennen, daß es Generierparameter unterschiedlicher Komplexität gibt.

4.1 "Primitive" Generierparameter

4.1.1 Numerische Parameter

Im Laufe der CODE-Definition und der Charakterisierung der Anforderungen, die an den CG bzw. das CG-Gerüst zu stellen sind, tauchten schon drei der vier trivialen Generierparameter auf:

\$_Wortlänge (siehe "Das Aufweiten (erster Teil)" in 3.2.4.2,
Abb. 3.3.1-4,
"Die magischen Begriffe ..." in 3.3.3.1,
Abb. 3.3.3-1 und
"Das 76. Zielcode-Muster" in 3.3.3.4),
\$_Adreßlänge (siehe Abb. 3.3.1-1),
\$_AdrEinh_Länge (siehe Abb. 3.3.1-4 und 3.3.2-2 bis 3.3.2-4).

Dazu kommt nun noch

\$_BefRef_Länge,
mit dem der CG-Implementierer angibt, wieviel Platz in den zu erzeugenden Codesequenzen zur Aufnahme jeweils einer Referenz dient (siehe 4.2.2).

Für diese Werte werden im CG-Rahmen Konstanten-Deklarationen vorgesehen, deren Initialwerte offen bleiben, z.B.:

```
DECLARE (§_Wortlänge) ganze Zahl
```

```
INVARIANT
```

```
INITIAL(
```

```
-----§_WL-----  
);
```

Die geeignet gekennzeichneten Lücken werden vom Generatorprogramm geschlossen. Was geeignet ist, hängt natürlich von der Programmiersprache ab, in der der CG(-Rahmen) verfaßt wird.

4.1.2 Konversionsroutinen

Ein Mechanismus, der sehr ähnlich arbeitet, wurde für die Konversionsroutinen in den Abbildungen 3.3.2-6 und 3.3.2-7 exemplarisch vorgestellt. Diese Funktionen werden deshalb hier nur noch kurz aufgezählt: Die

Konversion_in_§_Ganz

kann aus einer vorliegenden Sammlung entnommen und in den Rahmen eingesetzt werden; der eingesetzte Teil ist nur noch von §_AdrEinh_Länge abhängig (siehe 3.2.2 und "Konversion ganzer Zahlen" in 3.3.2).

Im Gegensatz dazu müssen die folgenden Routinen komplett vom CG-Implementierer zur Verfügung gestellt werden:

Konversion_in_§_Ratio (siehe 3.2.2)

Der Aufruf der

Konversion_in_§_Adresse

erfolgt erst im Binder, wo sowohl die Adreßdaten als auch die Referenzen in Befehlen erzeugt werden (siehe auch "Die Bearbeitung von Adreßobjekten" in 3.3.2 und siehe 4.3).

Auch für die Schiebeschrittzahl aus "sch_1"- und "sch_r"-Befehlen (siehe Wortbefehle in 3.3.3.2, sowie Anmerkung zur Bilanz und "Zwei Parameter ..." in 3.3.3.3) wird mindestens eine Konversionsfunktion

Konversion_in_§_Schiebeschrittzahl

benötigt (siehe aber auch 4.2.2).

Der Einfachheit halber wird festgelegt, daß auch diese Routinen vom Generator auf CG-Quellprogrammebene in das Gerüst einzufügen sind.

Anmerkung:

Der Einsatz unterschiedlicher Bezeichner für definierendes und angewandtes Auftreten durch Einschieben eines zusätzlichen Identifizierungsschrittes (wie in 4.1.1 mit Hilfe der Konstantenvereinbarung) ist natürlich auch hier möglich und empfehlenswert. Im Abschnitt "Konversion ganzer Zahlen" in 3.3.2 wurde aus Gründen einer einfacheren Beschreibung davon kein Gebrauch gemacht.

4.1.3 Daten-Codegeneratoren und Speicherregister

Der Generator wird in zwei Teilprogramme gegliedert, die vom CG-Implementierer sequentiell zu benutzen sind: Der erste Teil übernimmt die Generierparameter, die zur Erzeugung eines Daten-CGs benötigt werden. Es handelt sich dabei um die ersten drei numerischen Parameter `$_Wortlänge`, `$_Adreßlänge` und `$_AdrEinh_Länge` (siehe 4.1.1), sowie um die ersten drei Konversionen für ganze Zahlen, rationale Zahlen und Adressen (siehe 4.1.2). Diese Parameter werden sowohl in einen normalen CG-Rahmen als auch in ein eigenes Daten-CG-Gerüst eingesetzt; im Daten-CG-Gerüst werden keine Ablaufzweige für die Bearbeitung ausführbarer CODE-Befehle vorgesehen. Der Daten-CG wird übersetzt und kann dann seine Arbeit im Übersetzungssystem aufnehmen.

Der CG-Implementierer erstellt nun ein "Deklarationsprogramm", in dem die Datenbereiche vereinbart werden, die er als `$_Speicherregister` benötigt. Durch Übersetzung dieses Programms (im normalen Übersetzungsablauf) entsteht der benötigte Bibliotheksmodul. Durch Übersetzung eines "Spezifikationsprogramms", in dem auf alle Objekte des neuen Moduls Bezug genommen wird (angewandtes Auftreten), entsteht eine Folge von "import"-Befehlen und eine Import-Tabelle (siehe 3.2.4.1).

Beide zusammen bilden

- a) einen Standard-Vorlauf für CG und Binder, der (z.B. vom COT) bei allen folgenden Übersetzungen vorneweg einzuspielen ist, und
- b) den Teil der Betriebssystem-Schnittstelle, der den Prozeßwechsel betrifft (siehe "CODE-Register und Prozeßwechsel" in 3.2.1).

Damit wird auch klar, daß der CG-Implementierer von dieser Möglichkeit, den CODE-Registersatz beliebig zu erweitern, nur diszipliniert Gebrauch machen sollte.

Eine Konsequenz

Der Daten-CG gibt dem CG-Implementierer die Möglichkeit, auf sichere Art und Weise Datenbereiche anzulegen, auf die aus den Zielcodesequenzen heraus zugegriffen können werden soll. Für diesen Zugriff genügt es nicht, gewissermaßen anonym aus einem nicht bekannten Bezeichner aus dem Namenfeld eines CODE-Befehls reale Referenzen erzeugen zu lassen. Statt dessen muß der CG nicht nur Identifikatoren bearbeiten können, die aus Zwischensprachebefehlen kommen, sondern auch solche, die als Argumente in Elementen von Zielcodesequenz-Mustern auftreten (siehe "Pegelwert direkt identifiziert" in 4.2.2).

4.2 Die Zielcodesequenz-Muster

4.2.1 Sequenzaufbau und Einwortroutinen-Aufrufe

Alle Zielcodemuster bestehen aus einer Folge von Elementen. Jedes Element ist von folgender Form:

Parameterkennung	Ziellänge	Argument
------------------	-----------	----------

Bei der Übersetzung eines CODE-Befehls müssen die Elemente des zugeordneten Sequenzmusters (siehe 3.3.3.3 und 3.3.3.4) vom CG als Codeerzeugungskommandos abgearbeitet werden.

Dabei bestimmt die Parameterkennung, was zu tun ist, und woher die Information kommt; die Ziellänge gibt an, in welcher Länge Zielcode zu erzeugen ist. Das Argument ist eine von mehreren möglichen Informationsquellen. Ein einfaches Beispiel für so ein Element ist

Festes Muster	6	'FOFO'H
---------------	---	---------

Der erzeugte Bindecode würde aus den ersten sechs Bit des Arguments bestehen.

Die Zielcodeteile, die aus der Bearbeitung der einzelnen Sequenzelemente entstehen, hat der CG fortlaufend zu konkatenieren, bis das Sequenzende erreicht ist. Da die Anzahl der Sequenzelemente von Muster zu Muster unterschiedlich sein kann, wird das Sequenzende durch ein eigenes Element gekennzeichnet:

Muster-Ende		
-------------	--	--

Ein Spezialfall ist die (fast) leere Sequenz; sie besteht lediglich aus der Parameterkennung

Kein Muster		
-------------	--	--

und zeigt an, daß der Implementierer kein Codemuster angeben kann oder will; in diesem Fall soll der CG bei Übersetzung des entsprechenden Einwort- bzw. Adreß-Befehls so vorgehen, als ob ein Mehrwortbefehl vorläge (siehe 3.3.3.4).

Ein Beispiel wären die zwei Muster für die Schiebebefehle, wenn für einen Zielrechner implementiert werden soll, der nur Einzelschritt-Verschiebepfehle kennt.

Es ist klar, daß der CG-Implementierer bei der Angabe der Codesequenz-Muster 76, 77 und 78 (siehe 3.3.3.4) von der Verweigerungs-Möglichkeit keinen Gebrauch machen darf, da sonst der erzeugte CG bei Ablauf in eine Endlos-Schleife bzw. -Rekursion geraten würde.

4.2.2 Die Sequenzelement-Arten

Für die einfachste Form eines Sequenzelements wurde schon in 4.2.1 ein Beispiel gegeben:

Festes Muster	Länge	Bitkette
---------------	-------	----------

Damit werden Bitmuster angegeben, die schon zur Generierzeit festliegen, z.B. Befehlsschlüssel des Zielrechners.

Zwei Parameter, deren Argumente zur Generierzeit noch nicht angegeben werden können, wurden schon im letzten Abschnitt von 3.3.3.3 vorgestellt. Der eine bezieht sich auf das Namenfeld des aktuell zu übersetzenden CODE-Befehls und bewirkt einen Zugriff auf den entsprechenden Namensbucheintrag:

Pegelwert via Namenfeld		
-------------------------	--	--

Die Bearbeitung dieses Elements im CG ist zunächst identisch mit der Erzeugung von Adreßobjekten; zusätzlich wird eine Längenangabe ausgewertet, die es ermöglicht, im Fall eines sogenannten "wortorientierten" Rechners den gewünschten Teil aus Adreßobjekten auszusondern und in die Befehlsobjekte als Referenzteil einzublenden. Diese Längenangabe `$_BefRef_Länge` ist wiederum Generierparameter (siehe 4.1.1).

Beim zweiten Parameter geht es um die Schiebeschrittzahl aus dem aktuellen CODE-Befehl. Für ihre Darstellung im Zielcode ist (wie z.B. für Adressen auch) eine entsprechende Konversion notwendig.

Sollen Zielrechner berücksichtigt werden, die nur einen einzigen Schiebebefehl aufweisen, und die Schieberichtung aus der Befehlsversorgung entnehmen (weitere Steuerbits außerhalb des eigentlichen realen Befehlsschlüssels, vorzeichenbehaftete Schriebebschrittzahlen, oder ähnliches), dann müssen mindestens zwei Umwandlungsfunktionen (für jede Schieberichtung eine) vorgesehen werden. Ein Beispiel eines realen Rechners, für den dies erforderlich wäre, ist der TR-440.

Schiebeschrittzahl	Länge	
--------------------	-------	--

Bei der Abarbeitung eines solchen Elements muß der CG(-Rahmen) in Abhängigkeit von der Schieberichtung die zugehörige Konversion aufrufen. Diese Konversionen sind selbst Generierparameter; in 4.1.2 wurde allerdings der zweite hier geschilderte Fall gar nicht erst berücksichtigt.

Eine weitere Sequenzelement-Art ist unter anderem für die Implementierung von Speicherregistern wichtig; wie schon in 4.1.3 erwähnt, muß der CG-Implementierer die Möglichkeit haben, Referenzen zu erzeugen, die sich auf ein Speicherregister beziehen. Dazu muß dessen Identifikator angegeben werden können:

Pegelwert direkt identifiziert		&&_aA_Identor
--------------------------------	--	---------------

Die Bezeichner der Speicherregister sind dem CG-Implementierer bereits von der Erzeugung der Speicherregister (siehe 4.1.3) her bekannt.

Darüberhinaus gestattet dieses Element natürlich den Zugriff auch auf alle anderen Namensbuch-Einträge, insbesondere also auch auf &&_aktueller_Pegel; dies ist z.B. wichtig, wenn in Sequenzen lokale Verzweigungen aufgebaut werden sollen.

Wieviel Information aus dem CG-Zustand insgesamt dem CG-Implementierer durch derartige Sequenzelemente zugänglich gemacht wird, ist letztlich eine Frage des Komforts, allenfalls noch der Effizienz.

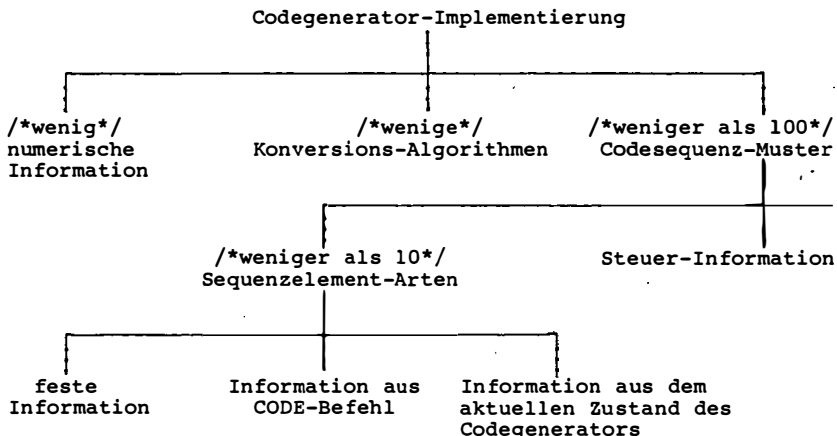
4.3 Nachtrag zum Binder

In Kapitel 4.1.3 wurde unterschlagen, daß zur Erzeugung der %%_Speicherregister auch der Binder benötigt wird. Was jedoch die Anforderungen angeht, die von der Seite der Codegenerierung an den Binder gestellt werden, kann festgehalten werden, daß die Tätigkeit des Binders nur noch über die Generierparameter \$_Adreßlänge, \$_BefRef_Länge und Konversion_in_\$_Adresse vom Zielrechner abhängt.

Somit darf vorausgesetzt werden, daß auch er in Gerüstform vorliegt und vom Generator gleichzeitig mit dem Daten-CG komplettiert wird.

5 Zusammenfassung

Auf der Grundlage der Zwischesprache-Definition wurde der Einfluß des Codegenerator-Implementierers auf die Gestaltung des Codegenerator-Quellprogramms auf ein Schema reduziert, das folgendermaßen veranschaulicht werden kann:



Diese Schematisierung darf nicht als Mechanisierung oder gar Automatisierung mißverstanden werden: gerade der höher-qualifizierte Anteil der Codegenerator-Implementierung, nämlich die Architektur-Abbildung von der virtuellen CODE-Maschine auf reale Rechner, wird von dem vorgeschlagenen Konzept am wenigsten unterstützt.

Dagegen ist zu erwarten, daß der einfache Aufbau des skizzierten Implementierungs-Werkzeugs, sowie die Revision der Aufgabenverteilung innerhalb des Übersetzungssystems, dazu beitragen, den Implementierungsaufwand für Codegeneratoren drastisch zu senken.

Bei der erwähnten Revision wurden an mehreren Stellen Absprachen zwischen Codegenerator einerseits und Compileroberteil, Binder, Prozeßverwalter und Bibliotheksroutinen andererseits notwendig. Auf die "Auslagerung" der Bearbeitung zielrechnerabhängiger Komplexe (Schlagwort PEARL-Systemteil!) in den Compileroberteil sei noch einmal besonders hingewiesen.

Insgesamt muß deshalb betont werden, daß die angestrebten Erleichterungen nicht isoliert durch die Verwirklichung des vorgestellten Codegenerierkonzepts erreicht werden können, sondern nur unter Einbeziehung aller anderen wesentlichen Komponenten des Übersetzungssystems. Dabei allerdings könnte die sehr "niedrig" angesiedelte CODE-Schnittstelle den Codegenerator zu einem einfach verfügbaren Bootstrap-Werkzeug machen.

Literatur

- AHUL78 Aho A.V., Ullman J.D.:
Principles of Compiler Design
Addison Wesley Publishing Company, Reading Mass., 1978
- APIE76 Gruber, Inderst, Piche:
Programmieranleitung für das ASME 1-PEARL-Subset
PDV-Bericht KfK-PDV 100, Gesellschaft f. Kernforschung
Karlsruhe, 11 (1976)
- BAGO71 Bauer F.L., Goos G.:
Informatik - Eine einführende Übersicht
Heidelberger Taschenbücher, Bd.80, Berlin: Springer, 1971
- BENE71 Bell C.G., Newell A.:
Computer Structures: Readings and Examples
McGraw-Hill, 1971
- BROW72 Brown P.J.:
Levels of Languages for Portable Software
Communications of the ACM 15, 12 (1972)
- CATT77 Cattell R.G.G.:
A Survey and Critique of Some Models of Code Generation
Deptm. of Computer Science, Carnegie-Mellon University,
Schrift CMU-CS-78-???, Pittsburgh Pen., 11(1977)
- CATT78 Cattell R.G.G.:
Formalization and Automatic Derivation of Code Generators
Deptm. of Computer Science, Carnegie-Mellon University,
Schrift CMU-CS-78-115, Pittsburgh Pen., 4 (1978)
- CAMA72 CAMAC, A Modular Instrumentation System for Data Handling
Commission of the European Communities, EUR 4100e, revised
version 1972, Joint Nuclear Research Centre Ispra Estab-
lishment - Italy, ESONE Committee
- CIMC76 Eichenauer B.F.:
Spezifikation der Zwischensprache CIMIC/C
Ges. f. Prozeßrechnerprogrammierung mbH, Balanstr. 138/I,
8 München 90, 9(1976)
- CIME76 Mühlhahn:
Spezifikation CIMIC-1
PDV-Bericht KfK-PDV 75, Gesellschaft f. Kernforschung
Karlsruhe, 5(1976)
- CIMP77 Eichenauer B.F., Henn R., Lucas K., Zeh A.:
Spezifikation der Zwischensprache CIMIC/P
Ges. f. Prozeßrechnerprogrammierung mbH, Balanstr. 138/I,
8 München 90, 5(1977)

- DINP81 Programmiersprache PEARL - Basic PEARL
Vornorm DIN 66253 Teil 1
Beuth Verlag GmbH, Berlin - Köln 1981
- EBER79 Eberlein W.:
Implementation eines Zwischensprache-Transformators
Diplomarbeit in Informatik, Universität Erlangen, 1979
- EPDV73 Timmesfeld K.H., et al.:
PEARL, A proposal for a process- and experiment automation
realtime language
PDV-Bericht KfK-PDV 1, Ges. f. Kernforschung Karlsruhe,
4(1973)
- FLEI79 Fleischmann A.:
Anpassung eines rechnerunabhängigen PEARL-Betriebssystems
für die Siemens 310
Studienarbeit in Informatik, Universität Erlangen, 1979
- FRAS77 Fraser C.W.:
A Knowledge-Based Code Generator Generator
SIGPLAN Notices 8 (1977) 126-129
- GRAH80 Graham S.L.:
Table-Driven Code Generation
Computer Vol.13, 8 (1980)
- GRIE71 Gries D.:
Compiler Construction for Digital Computers
John Wiley & Sons, New York, 1971
- GÜSC72 Güntsch F.R., Schneider H.-J.:
Einführung in die Programmierung digitaler Rechenautomaten
Walter de Gruyter Verlag, Berlin - New York, 1972
- HOLL76 Holleczeck P.:
Stufe 2 - ASME-PEARL-Compilersystem - Grundideen
Entwicklungsnotiz PDV-E 89, Gesellschaft f. Kernforschung
Karlsruhe, 10 (1976)
- LAMP78 Lampe K.:
Erstellung der Standard-E/A-Laufzeitroutinen des PEARL-
ASME-Stufe-1-Compilersystems in PEARL
Diplomarbeit in Informatik, Universität Erlangen, 1978
- LIND81 Lindstedt W.:
Ein Verfahren zur Erstellung portabler algorithmischer
Laufzeitprogramme
Dissertation in Informatik, Universität Erlangen,
Arbeitsbericht Band 14 Nr. 5 des Instituts für Mathema-
tische Maschinen und Datenverarbeitung, 6 (1981)

- PETE79 Petereit R.:
A Specification Language for Code Generators
Tagungsband der 9. GI-Jahrestagung, 10 (1979)
- POWA73 Poole P.C., Waite W.M.:
Portability and Adaptability
Lecture Notes in Economics and Mathematical Systems,
Advanced Course on Software Engineering, 1973
- PRES82 Prester F.-J.:
Ein portabler Treiber für graphische Geräte
Dissertation in Informatik, Universität Erlangen, 1982
- RÖS178 Rössler R.:
ERLAN-Handbuch
Interner Bericht am Physikalischen Institut der Universität Erlangen, 4 (1978)
- RÖS278 Rössler R.:
PEARL-Betriebssystem für den Z80-Rufbus-Subset
Entwicklungsnotiz PDV-E 119, Gesellschaft f. Kernforschung
Karlsruhe, 6 (1978)
- SCHN75 Schneider H.-J.:
Compiler - Aufbau und Arbeitsweise
Walter de Gruyter Verlag, Berlin - New York, 1975
- SCHN81 Schneider H.-J.:
Problemorientierte Programmiersprachen
Leitfäden der angewandten Informatik, Teubner,
Stuttgart, 1981
- TERM78 Terman C.J.:
The Specification of Code Generation Algorithms
Massachusetts Institute of Technology, Laboratory for Computer Science, MIT/LCS/TR-199, 1 (1978)
- TRAU80 Trautner M.:
Codegenerator- und Systembeschreibung der Siemens 310-PEARL-Implementation
Entwicklungsnotiz PDV-E 142, Gesellschaft f. Kernforschung
Karlsruhe, 6 (1980)
- TRAU81 Trautner M.:
Spezifikation des Codegenerators für PEARL auf einem KONTRON-System
Interner Bericht am Physikalischen Institut der Universität Erlangen, 1981

- VEPA77 Berner H.B., et al.:
VEPAS - Verfahren zur Erstellung portabler Prozeßautoma-
tisierungs-Software
Entwicklungsnotiz PDV E-101, Gesellschaft f. Kernforschung
Karlsruhe, 1977
- WAIT70 Waite W.M.:
The Mobile Programming System - Stage 2
Communications of the ACM 13, 7 (1970) 415
- WETT77 Wettstein H.:
Assemblierer und Binder
Scriptum zur gleichnamigen Vorlesung, Informatik III der
Universität Karlsruhe, 10 (1977)
- WIMM78 Wimmer W.:
Ein allgemeiner Cross-Assembler für unterschiedliche Rech-
nertypen
Deutsches Elektronen-Synchrotron, Schrift DESY DV-78/04,
Hamburg, 5 (1978)

Anhang 1: Erläuterungen zur Syntaxnotation

Produktionen beginnen - wie üblich - mit der zu produzierenden Syntaxvariablen. Im darauf folgenden Kasten stehen die Alternativen, die von jeweils einem Gleichheitszeichen eingeleitet werden. Verschiedene Bestandteile innerhalb einer Alternative werden entweder durch Minus- oder Pluszeichen getrennt.

Minuszeichen bedeuten, daß die Reihenfolge der getrennten Bestandteile irrelevant ist, während das Pluszeichen die textuelle Reihenfolge impliziert. Jeder Bestandteil einer Alternative kann aus zwei Elementen bestehen: einer Charakterisierung, die von einem Doppelpunkt abgeschlossen wird, und der eigentlichen Elementdarstellung. Die Charakterisierung ermöglicht eine (knappe) Kommentierung des Elements und kann (zusammen mit dem Doppelpunkt) auch entfallen.

Bezeichner von Syntaxvariablen beginnen mit einem der vier Fluchtsymbole

Prozent, Dollar, Paragraph oder kaufmännisches Und.

Terminale sind entweder Zeichenketten, die in Anführungszeichen eingeschlossen sind, oder es handelt sich um "Quasi-Terminale", dh. sie sehen fast wie Syntaxvariable aus, beginnen aber mit Paaren gleicher Fluchtsymbole.

Die Kastenränder bestehen entweder aus Minuszeichen, Pluszeichen oder Sternen; damit wird die Abstraktionsstufe gekennzeichnet, auf der die Produktion zu sehen ist (siehe auch folgende Beispiele).

Auf der nächsten Seite sind die Bedeutungen der Symbole noch einmal zusammenfassend dargestellt. Dahinter folgen noch einige Beispiele.

Kennzeichnungen von Syntaxvariablen und Quasi-Terminalen

% _ % _	Architektur der CODE-Maschine
\$ _ \$ _	Datentypen in CODE
ß _ ß _	CODE-Befehlssatz
& _ & _	allgemeine Syntaxvariable

Rechte Seiten von Produktionen

-----	+++++	*****
/* abstrakt */	/* teilweise	/* fast
	konkretisiert */	konkret */
-----	+++++	*****

Symbole innerhalb der Kästen:

=	Einleitung einer Alternative
-	Trennung der Elemente einer Alternative ohne Reihenfolge-Festlegung
+	... mit Reihenfolge-Festlegung
	Elements
/*	Kommentar-Anfang
*/	Kommentar-Ende
"	Terminale werden in Anführungszeichen gesetzt

Beispiele β _Platzreservierung

/* zunächst abstrakt*/

```

-----
/* 1. Alternative */
=
/* Charakterisierung */
Befehlsschlüssel :
/* 1. Element (der 1. Alternative) */
 $\beta\beta$ _Pegelverstellung /* Quasi-Terminal */
/* Element-Trenner */
-
/* 2. Element (der 1. Alternative) */
Argument :  $\beta\beta$ _Pegeldifferenz
-----

```

/* schon etwas konkreter */

```

+++++
=   Befehlsschlüssel :  $\beta\beta$ _Pegelverstellung
    + Befehlsrest_Länge :  $\beta\beta$ _Anzahl_der_folgenden_
                        Komponenten
    + Adreßausdruck :  $\beta\beta$ _additive_Pegelkomponenten
+++++

```

/* ziemlich konkret */

```

*****
* =   optcod : "level"
*   + brlang : && ganze Zahl
*   + adrexp : /*Folge der*/ $\beta$ _Längenkomponenten
*****

```

 β _Längenkomponente

```

*****
* =   einheit :  $\beta\beta$ _Identifikator_aA /*angewandtes
*                                   Auftreten*/
*   + mult    : && ganze Zahl
*****

```

Anhang 2: Produktionenliste (alphabetisch sortiert)

Adreßart(3.3.2)

```
*****
* = "refer"      /* Adresse eines Objekts */
* = "differ"     /* Adreßdifferenz */
*****
```

B_Adreß_Befehl(3.3.3.2)

```
*****
/*Transfer*/
* = opc:"lade_ix" + tlg:$$_Adresse + opd:ß_ref
* = opc:"l_adr_ak" + tlg:$$_Adresse + opd:ß_ref
* = opc:"s_adr_ak" + tlg:$$_Adresse + opd:ß_ref
* = opc:"ak_in_ix" + tlg:$$_Adresse + amo:"d"
/*Sprung*/
* = opc:"springe" + tlg:$$_Adresse + opd:ß_ref
* = opc:"spr_fel" + tlg:$$_Adresse + opd:ß_ref
* = opc:"spr_neg" + tlg:$$_Adresse + opd:ß_ref
* = opc:"spr_nul" + tlg:$$_Adresse + opd:ß_ref
* = opc:"up_spr" + tlg:$$_Adresse + amo:"d"
*                               + nam:&&_aA_Identor
* = opc:"rückspr" + tlg:$$_Adresse + amo:"d"
/*Adreß-Numerik*/
* = opc:ß_a_n_op + tlg:$$_Adresse + opd:ß_ref
*****
```

B_Adreß_Rechnung(3.2.4.3)

```
-----
=   Bß Addit - $$ Adresse - B_Referenz
/* 1.Operand muß ebenfalls eine Adresse sein */
=   Bß Subtr - $$ Adresse - B_Referenz

=   Bß Multi - $$ Adresse - B_Referenz
/* Akkuinhalt muß vom Typ
   $$ Ganztyp($$ Adresse)
   sein */
-----
```

B_Adreß_Rechnung(3.3.3.1)

```
+++++
* =   opc : B_adr_op
*   /* $_Adresse */
* + opd : B_ref
+++++
```

\$_Adreßtyp(3.2.2)

```
=====
=   $$ Adresse
=====
```

B_adr_op(3.3.3.1)

```
+++++
* =   "+" = "-" = "*"
+++++
```


\$_Akkutyp(3.2.2)

```

-----
=  $_Adreßtyp
=  $_Arithmetiktyp
-----

```

\$_a_n_op(3.3.3.2)

```

*****
* =  "+_adr"  =  "-_adr"  =  "*_adr"
*
*****

```

\$_Arithmetiktyp(3.2.2)

```

-----
=  $_Bittyp
=  $_Numeriktyp
-----

```

\$_ausführbarer_Befehl(3.2.4, 3.2.4.3)

```

-----
=  $_Transfer
=  $_Numerik
=  $_Adreß_Rechnung
=  $_Bitketten_Verarbeitung
=  $_Sprung
-----

```

\$_ausführbarer_Befehl(3.3.3.2, 3.3.3.4)

```

*****
* =  $_Adreß_Befehl
* =  $_Wort_Befehl
*
*      /*Aufruf einer Mehrwort-Routine:
* =  opc:"lib_call"
*   + amo:"direkt"
*   + nam:&&_MwR_Identor
*
*      Lade-unmittelbar Operandenadreßregister für
*      Mehrwortroutinen-Aufrufe:
* =  opc:"l_mwr_oarg"
*   + amo:"unmittelbar"
*   + nam:&&_aA_Identor
*
*      Lade-unmittelbar Schiebeschrittzahl-Register
*      für Mehrwortroutinen-Aufruf:
* =  opc:"l_mwr_sszr"
*   + amo:"unmittelbar"
*   + scz:&&_positive_ganze_Zahl
*   */
*****

```

B_beding(3.3.3.1)

```

+++++
= "fel" = "neg" = "nul"
+++++

```

B_Bedingung(S.38)

```

-----
= BB_unbedingt
= BB_wenn_Fehler           = BB_wenn_kein_Fehler
= BB_wenn_positiv          = BB_wenn_negativ
= BB_wenn_null             = BB_wenn_ungleich_null
-----

```

B_Befehlsdarstellung(3.3)

```

+++++
= BB_Befehlsschlüssel_Feld
+ BB_Folge_der_Versorgungs_Felder
+++++

```

%_Befehlsregister(3.2.1)

```

-----
= %%_Befehlsschlüssel
- %%_Typ
- %%_Adressiermodus
- %%_Sprungbedingung
- %%_Schiebe_Richtung_und_Schrittzahl
-----

```

%_Befehls_und_Operanden_Adresse(3.2.1)

```

-----
= %%_Referenzbasis-Register
- %%_Indexregister
-----

```

B_Befehlszeile(3.3)

```

+++++
= BB_Zeilenlänge
+ B_Befehlsdarstellung
+++++

```

B_Bitketten_Verarbeitung(3.2.4.3)

```

-----
= Bef.Schl.      : B_Dyadische_Bitketten_Operation
- Operationstyp : B_Bittyp
- Operand       : B_Referenz
= Bef.Schl.      : B_Monadische_Bitketten_Operation
- Operationstyp : B_Bittyp
-----

```

B_Bitketten_Verarbeitung(3.3.3.1)

```

+++++
= opc:"und" + typ:B_Bittyp + opd:B_ref
= opc:"i_oder" + typ:B_Bittyp + opd:B_ref
= opc:"x_oder" + typ:B_Bittyp + opd:B_ref
= opc:"kplem" + typ:B_Bittyp
= opc:"sch_l" + scz:B_sch_zahl + typ:B_Bittyp
= opc:"sch_r" + scz:B_sch_zahl + typ:B_Bittyp
+++++

```

\$_Bittyp(3.2.2, 3.3)

```

=====
= $$_Bit - $$_Länge
=====

```

```

+++++
= $$_Bit + $_Länge_in_Bit
+++++

```

\$_CODE_Befehl(3.2.4)

```

=====
= $_Pseudobefehl
= $_Platzreservierung
= $_ausführbarer_Befehl
/* Offenes Ende : */
= $_andere_CODE_Befehle
=====

```

\$_CODE_Datenobjekt(3.3.2)

```

*****
= $ Datenobjekt in CODE
*****

```

\$_CODE_Datentyp(3.2.2)

```

=====
= $_Datentyp
=====

```

%_CODE_Maschine(3.2.1)

```

=====
= %%_Speicher_für_Daten_und_Befehle
- %_Registersatz_des_Prozessors
/*Offenes Ende : */
- %%_andere_Komponenten_der_virtuellen_Maschine
=====

```

\$_Datencode_erzeugen(3.2.4.2)

```

=====
= Bef.Schl. : $$_Erhöhung_des_aktuellen_Pegels_
              durch_Vorbelegung
- Vorbelegung : &&_zielrechnerunabhängige_
              Darstellung_des_Objekts
=====

```

\$_Datencode_erzeugen(3.3.2)

```

+++++
= Befehlsschlüssel : $$_Datencode
+ Objektart       : $$_Länge_und_Typ_des_
                    Objekts
+ Vorbelegungswert : $$_zielrechnerunabhängige_
                    Objektdarstellung
+++++
*****
= optcod : "datcod"
*****
+ länge : &&_aA_Identor
*****
+ typ   : $_Datentyp
*****
+ datobj : $ CODE Datenobjekt
*****

```

\$_Datenobjekt_in CODE(3.3.2)

```

*****
** = $$_Bitkette /*entweder der Länge $ _AdrEin Länge
**                für Adressiereinheiten (s. 3.2.2)
**                oder der im $_Datentyp angegebenen
**                Länge*/
**
** = $$_ganze_Zahl
** = $$_rationale_Zahl
**
** /* Adreß-Objekt: */
** = $ _Adreßart + && aA Identor zwei
*****

```

\$_Datentyp(3.2.2)

```

-----
- = $$ Adressiereinheit
-
- = $ Akkutyp
-
- /* Offenes Ende : */
- = $$_andere_Datentypen
-----

```

%_Die_virtuelle_CODE_Maschine_in_der_Ausführungsphase(3.2.1)

```

-----
- = %_CODE_Maschine
-----

```

B_Dyadische_Bitketten_Operation(3.2.4.3)

```

-----
- = Bß_Und = Bß_einschl_Oder = Bß_ausschl_Oder
-----

```

B_Export(3.2.4.1)

```

-----
- = Bef.Schl. : Bß_Erzeugung_eines_Eintrags_in_der_
-               Externdefinitions_Tabelle
-
- Ident. : && angewandtes_Auftreten_eines_
-               Identifikators
-----

```

B_Export(3.3.1)

```

*****
** = optcod : "export"
** - name : && aA Identor
*****

```

\$_Ganztyp(3.2.2, 3.3)

```

-----
- = $$_E_Kompl - $$_Länge
-
- = $$_Z_Kompl - $$_Länge
-
- = $$_gep_BCD - $$_Länge
-
- = $$_ung_BCD - $$_Länge
-----

```

```

+++++
- = $$_E_Kompl + $ _Länge_in_Bit
- = $$_Z_Kompl + $ _Länge_in_Bit
- = $$_gep_BCD + $ _Länge_in_Bit
- = $$_ung_BCD + $ _Länge_in_Bit
+++++

```

B_Import(3.2.4.1)

```

=====
= Bef.Schl. : B_Spezifikation_eines_externen_
      Objekts
- Ident.    : && Definierendes Auftreten_eines_
      Identifikators
=====

```

B_Import(3.3.1)

```

*****
* = optcod : "import"
* - name   : && dA Identor
*
*****

```

S_Länge_in Bit(3.3)

```

+++++
+ = && natürliche_Zahl
+
+++++

```

B_Modulende(3.2.4.1)

```

=====
= Bef.Schl. : B_Modul_abschließen
=====

```

B_Modulende(3.3.1)

```

+++++
+ = Befehlsschlüssel : B_Modul_abschließen
+
+++++
* =
*   optcod : "modend"
*
+++++

```

/*B_Modulanfang (Grundversion siehe S.42)*/

B_Modulanfang(3.2.4.1)

```

=====
= Befehlsschlüssel : B_Modul_eröffnen
=====

```

B_Modulanfang_mit_NaB_Vorlauf_für_Arithmetiktypen(3.3.1)

```

+++++
+ = Befehlsschlüssel : B_Modul_eröffnen
+   + Befehlsrestlänge : B_Anzahl_der_folgenden_
+                         Typdefinitionen
+   + NaB.Vorlauf      : B_Folge_der_
+                         Typdefinitionen
+
+++++
* =
*   optcod : "modbeg"
*   + tdanz : && natürliche_Zahl
*   + nabvorl : /*Folge der*/ B Typdefinitionen
*
+++++

```

B_Monadische_Bitketten_Operation(3.2.4.3)

```

=====
= B_Komplex
= B_Schieben - B_Richtung - B_Schritte
=====

```

B_NaB_eintragen(3.2.4.1)

```

=====
= Bef.Schl. : BB_Definition_einer_symbolischen_
  Adresse
- Identifikation : &&_Definierendes_Auftreten_
  eines_Identifikators
- Zugeordneter Pegel : &&_Adreß_Ausdruck
=====

```

B_NaB_eintragen(3.3.1)

```

+++++
= Befehlsschlüssel : BB_Definiere_Pegelwert
+ Identifikation : BB_definierendes_Auftreten_
  eines_Identifikators
+ Befehlsrestlänge : BB_Anzahl_der_folgenden_
  Summanden
+ Adreßausdruck : BB_Folge_der_Pegelsummanden
+++++
= optcod : "define"
+ name : &&_dA_Identor
+ psanz : &&_natürliche_Zahl
+ adrexp : /*Folge der*/ 3 Pegel Summanden ver...
*****

```

B_Numerik(3.2.4.3)

```

=====
= Befehlsschlüssel : B_Numerik_Operation
- Operationstyp : $_Numeriktyp
- Operand : B_Referenz
=====

```

B_Numerik(3.3.3.1)

```

+++++
= opc : B_num_op
+ typ : $_Numeriktyp
+ opd : B_ref
+++++

```

B_Numerik_Operation(3.2.4.3)

```

=====
= BB_Addit = BB_Subtr = BB_Multi = BB_Divis
= BB_Negat
/* Besonders offenes Ende :
= BB_andere_Numerik_Operationen */
=====

```

\$_Numeriktyp(3.2.2)

```

=====
= $_Rationaltyp
= $_Ganztyp
=====

```

B_num_op(3.3.3.1)

```

+++++
= "+" = "-" = "*" = "/"
+++++

```

```
/*B Pegel_Summand (überlastete Version siehe S.43)*/
```

```
B_Pegel_Summand vereinfacht(3.3.1)
```

```
*****
** =   einheit : &&_aA_Identor                               **
** + mult      : &&_ganze_Zahl                                **
*****
```

```
B_Pegel_verstellen(3.2.4.2)
```

```
-----
=   Bef.Schl.      : B_Pegelschiebung_des_
                    aktuellen_Pegels
-   Pegelveränderung : &&_Adresse_Ausdruck
-----
```

```
B_Pegel_verstellen_relativ_vorwärts(3.3.2)
```

```
*****
** =   Befehlsschlüssel : B_Pegelverstellung                **
** + Befehlsrestlänge  : B_Anzahl_der_folgenden              **
**                       Summanden                           **
** + Adressausdruck    : B_Folge_der_Pegelsummanden          **
*****
** =   optcod : "level"                                       **
** + psanz   : &&_natürliche_Zahl                             **
** + adrexp  : /*Folge der*/B_Pegel_Summanden_ver...       **
*****
```

```
B_Platzreservierung(3.2.4, 3.2.4.2)
```

```
-----
=   B_Pegel_verstellen
=   B_Datencode_erzeugen
-----
```

```
B_Pseudobefehl(3.2.4, 3.2.4.1)
```

```
-----
=   B_Modulanfang
=   B_Modulende
=   B_NaB_eintragen /* Namensbuchführung */
=   B_Import
=   B_Export
-----
```

```
$_Rationaltyp(3.2.2, 3.3)
```

```
-----
=   $_Ratio - $_Länge
-----
```

```
*****
** =   $_Ratio + $_Länge_in_Bit
*****
```

ß_ref(3.3.3.1, 3.3.3.2)

```

+++++
= amo: "d" /*direkt*/ + name: &&_aa_Identor
= amo: "di" /*indiziert*/ + name: &&_aa_Identor
= amo: "i" /*indirekt*/
*****

```

ß_Referenz(3.2.4.3)

```

-----
= Adressiermodus : ßß_direkt
- Ident. : &&_angew_Auftreten_eines_Identifikators
= Adressiermodus : ßß_indiziert
- Ident. : &&_angew_Auftreten_eines_Identifikators
= Adressiermodus : ßß_indirekt
-----

```

%_Registersatz_des_Prozessors(3.2.1, 3.3.3.4)

```

-----
= %%_Akkumulator
- %_Statusregister
- %%_übliche_Befehlsfortschaltung
- %_Befehlsregister
- %_Befehls_und_Operanden_Adresse
- %%_Rücksprungadressen_Keller
- %%_Speicherregister /*Näheres siehe 4.1*/
- %%_Zustand_des_realen_Prozessors

/*
= %%_MwR_Operandenadresse_Reg
= %%_MwR_Schiebeschrittzahl_Reg
*/
-----

```

ß_Relative_Veränderung_des_aktuellen_Pegels(3.3.2)

```

+++++
= ß_Pegel_verstellen_relativ_vorwärts
+++++

```

ß_rg_n_op(3.3.3.2)

```

*****
= "+_g" = "-_g" = "*_g" = "/_g"
= "+_r" = "-_r" = "*_r" = "/_r"
*****

```

ß_sch_zahl(3.3.3.1)

```

+++++
= &&_positive_ganze_Zahl
+++++

```



```

/*B Sprung (Idealtendenz: siehe S.38)*/
/*B_Sprung Operation mit Ref (Idealtendenz: siehe S.38)*/
B_Sprung /*Realitätstendenz*/(3.2.4.3)

```

```

-----
= Befehlsschlüssel : B_Unterprogramm_Rücksprung
= Befehlsschlüssel : B_Unterprogramm_Sprung
- Operand          : B_Referenz
= Befehlsschlüssel : B_Betriebssystemaufruf
- Operand          : B_Referenz
= Befehlsschlüssel : B_Springen
- Sprungmodus      : B_Bedingung
- Operand          : B_Referenz
*/
-----

```

B_Sprung(3.3.3.1)

```

+++++
= opc:"rückspr" /*$$_Adresse*/
= opc:"up_spr"  /*$$_Adresse*/ + nam:&&_aA_Identor
= opc:"spr"     /*$$_Adresse*/
                      + bed:B_beding + opd:B_ref
+++++

```

%_Statusregister(3.2.1)

```

-----
= %%_Fehleranzeige
- %%_Vorzeichen
- %%_Nullanzeige
-----

```

B_Transfer(3.2.4.3)

```

-----
= B_Laden_Akku      - $_Akkutyp - B_Referenz
= B_Speichern_Akku - $_Akkutyp - B_Referenz
= B_Laden_XReg     /* $_Adresse */ - B_Referenz
= B_vom_Akku_ins_XReg /* auch $_Adresse */
-----

```

B_Transfer(3.3.3.1)

```

+++++
= optcod:"lade_ak"  typ:$_Akkutyp opd:B_ref
= optcod:"spei_ak"  typ:$_Akkutyp opd:B_ref
= optcod:"ak_in_ix"
= optcod:"lade_ix" /*$_Adresse*/ opd:B_ref
+++++

```

B_Typdefinition(3.3.1)

```

*****
* = name      : &&_dA_Identor
* + einheit: $ Arithmetiktyp
*****

```

B_Wort_Befehl(3.3.3.2)

```

*****
/*Transfer*/
= opc:"l_rgb_ak" + tlg:&&_aA_Identor + opd:B_ref
= opc:"s_rgb_ak" + tlg:&&_aA_Identor + opd:B_ref
/*Numerik*/
= opc:B_rg_n_op + tlg:&&_aA_Identor + opd:B_ref
/*Bitkettenverarbeitung*/
= opc:"und" + tlg:&&_aA_Identor + opd:B_ref
= opc:"i_oder" + tlg:&&_aA_Identor + opd:B_ref
= opc:"x_oder" + tlg:&&_aA_Identor + opd:B_ref
= opc:"kplem" + tlg:&&_aA_Identor + amo:"d"
= opc:"sch_r" + scz:&&_positive_ganze_Zahl
+ tlg:&&_aA_Identor + amo:"d"
= opc:"sch_l" + scz:&&_positive_ganze_Zahl
+ tlg:&&_aA_Identor + amo:"d"
*****

```

Anhang 3: Abkürzungen1. Allgemeinsprachliche Abkürzungen

Abb.	Abbildung
bzw.	beziehungsweise
d.h.	das heißt
ggf.	gegebenenfalls
i.a.	im allgemeinen
Kap.	Kapitel
s.	siehe
s.o.	siehe oben
s.u.	siehe unten
u.a.	unter anderem
usw.	und so weiter
vgl.	vergleiche
z.B.	zum Beispiel

2. Abkürzungen für Fachbegriffe

CG	Codegenerator
CGen	Codegeneratoren
CGg	Codegenerierung
CGs	(des) Codegenerators
CIMIC	Compiler Internal Machine Independent Code
CODE	CIMIC-Objekt-Dialekt-Erlangen
COT	Compileroberteil

3. Abkürzungen aus Syntaxregeln

&&_aA_Identor	Angewandtes Auftreten eines Identifikators
adrexp	Adreßausdruck
\$_AdrEinh_Länge	Länge einer Adressiereinheit
amo	Adreßmodus
BCD	Binary Coded Decimal
Bef.Schl.	Befehlsschlüssel
brlang	Befehlsrest-Länge
&&_dA_Identor	definierendes Auftreten eines Bezeichners
datobj	Datenobjekt
\$\$_E_Kompl	Einer-Komplement
\$\$_gep_BCD	gepackte BCD-Codierung
Ident.	Identifikation
..._Identor	Identifikator
mult	Multiplikator
..._MwR...	Mehrwortroutine
..._NaB...	Namensbuch
nabvorl	Namensbuch-Vorlauf
ß_num_op	Numerik-Operation
opc	Operationscode
opd	Operand
optcod	Operationscode
psanz	Anzahl der Pegelsummanden
ß_ref	Referenz
scz	Schiebezahl
ß_sch_zahl	Schiebezahl
tdanz	Anzahl der Typdefinitionen
tlg	Typlänge
\$\$_ung_BCD	BCD-Codierung, ungepackt
\$\$_Z_Kompl	Zweier-Komplement
XReg	Indexregister

Anhang 4: Abbildungsverzeichnis

	Seite
2-1	5
2-2	5
2-3	8
3.2-1	12
3.2.1-1	14
3.2.1-2	17
3.2.1-3	19
3.2.2-1	24
3.2.4.1-1	30
3.2.4.1-2	31
3.3.1-1	42
3.3.1-2	45
3.3.1-3	46
3.3.1-4	47
3.3.1-5	50
3.3.2-1	53
3.3.2-2	56
3.3.2-3	57
3.3.2-4	58
3.3.2-5	58
3.3.2-6	59
3.3.2-7	59
3.3.2-8	60
3.3.2-9	61
3.3.3.3-1	71
3.3.3.3-2	72
3.3.3.3-3	73
3.3.3.4-1	77
3.3.3.4-2	79

Anhang 5: Stichwortverzeichnis

Im Normalfall sind Seitennummern bzw. Seitennummern-Bereiche angegeben, in einigen Fällen auch weitere Stichwörter. Verweisen auf Kapitel bzw. Anhänge ist "K." bzw. "A." vorangestellt. Weniger wichtige Hinweise stehen in Klammern.

abstrakte ...-Maschine: virtuelle Maschine

abstrakte Syntax: 12, A.1, (9, 40, 41, A.2)

Adreßart: 54-55, 60

Adreßausdruck: 28, 32

Adreßbefehl: 67, 68, K.3.3.3.3, (63)

Adreßbuch: 11

-A.-Führung: K.3.2.4.1, K.3.3.1 ab Seite 43

-A.-Vorlauf: 48-49

\$\$_Adresse: 41, 42

Adresse: 22-23, 63

Adressiereinheit: 22, 32, 41, 42, 80, (81)

Adressiermechanismen: K.3.2.3

Adreßlänge: 23, 42, 80

Adreßobjekt: 60

Akkumulator: 14, 15

Akkutyp: 24, 36

Allgemeine Konversionsprozedur: 57

Ankunftsregister (einer Unterbrechung): 17

Anzeigeregister: 14, 16

Architektur der virtuellen CODE-Maschine: K.3.2.1

Arithmetiktyp: 24

Assembler einer virtuellen Maschine: 4, 9

Aufweitung: 32, 35, 47, 58, 66, (22, 37, 49, 57, 63, 76)

ausführbare Befehle: K.3.2.4.3, K.3.3.3

Ausnahmefall-Behandlung: 16, 17

Bedingungscode-Register: Anzeigeregister
 bedingter Sprungbefehl: 16, Sprungbefehl
 Befehlsdecodierung: 71-73
 Befehlsregister: 13, 14, 19
 Befehlssatz der CODE-Maschine: K.3.2.4, K.3.3
 Betriebssystem: Laufzeitunterstützung
 Bibliotheksroutinen: Laufzeitunterstützung
 -Portabilität von B.: 3
 Bindecode-Schnittstelle: 4, 87
 Bindefunktionen (quellsprachspezifische): 10
 CIMIC: 9, (1)
 CODE: 9
 Codegenerator-Gerüst (-Rahmen, -Skelett): 7, K.3.3, K.4
 CODE-Maschine: virtuelle CODE-Maschine
 Codesequenz-Muster: Zielcodesequenz-Muster
 Compileroberteil (COT): 1
 Crosscompiler: 2
 Daten-Codegeneratoren: K.4.1.3
 Datentypen der virtuellen CODE-Maschine: K.3.2.2
 Effizienz: 2, 3
 Einadreß-Architektur: 16
 eingebettete Systeme: 3
 Einwortbefehle: 67, K.3.3.3.3, K.4.2.1
 Erstellung von Codegeneratoren: 7, K.4, besonders K.4.1.3
 Export: 29, 51

Generierung von Codegeneratoren: Erstellung v. C.
 Generierparameter: K.4.1.1, K.4.1.2, K.4.2
 höhere Programmiersprachen: problemorientierte P.
 Import: 29, 51
 Indexregister: 14, 15, 64, 68
 Interrupt: Unterbrechung
 Janus-Architektur: 4
 Kontext in der Zwischensprache CODE: 41
 Konversionsprozedur: allgemeine K.
 Konversionsroutinen: Generierparameter
 Längenumrechnung: 46-48
 Laufzeitbefehle: ausführbare Befehle
 Laufzeitunterstützung: 1, Prozeßverwalter, Bibliotheksrou-
 tinen, Betriebssystem
 -L. und Codegenerator: 3, 10, 88
 Laufzeitsystem: Laufzeitunterstützung
 Maschinenbeschreibung: Rechnerbeschreibung
 Maschinencode einer virtuellen Maschine: 4, 9
 Mehrwortbefehl: 67, K.3.3.3.4, (63)
 Mehrwort-Routine: 75-79
 Muster: Zielcodesequenz-Muster
 Namensbuch: Adreßbuch
 Nebenläufigkeit: 10
 Numeriktyp: 24
 Operandenkeller: 15, 16
 Optimierung: 10, 16
 Orthogonalität: 10, 15, K.3.2, K.3.3

Parallelität: Nebenläufigkeit

Parameter bei der Erstellung von CGen: Generierparameter

Pegelverstellung: 32

Platzreservierung: K.3.2.4.2, K.3.3.2

Portabilität: K.1

problemorientierte Programmiersprachen (Übersetzung von ...): 4

Prozeß: Rechenprozeß

Prozeß-Programmiersprachen: 1, 2

-komplexe Elemente von P.: 10

Prozeßrechner-Verbund: 3

Prozeßverwalter: Laufzeitunterstützung

Prozeßwechsel: 18

Pseudobefehle: K.3.2.4.1, K.3.3.1

Rechenprozeß: 10

Rechnerbeschreibungs-Sprache: 1

Referenzstufen: 25

Registersatz der virtuellen CODE-Maschine: K.3.2.1

Rückkehradressen-Keller: 14, 18

Sequenzelement(-Art): K.4.2, Zielcodesequenz-Muster, Generierparameter

Speicher: 14, 15

Speicherregister: 18, 61, K.4.1.3, (86,87)

Sprungbefehl: 38, 39, 65, 68, (70, 72)

Statusanzeigen: 14, 16

symbolische Adressierung: 11, 28

Tabellensteuerung: 1

Transferbefehle: 34, 64, 68, 69

Typbindung: 18, (11)

Typprüfung: Typbindung

Übersetzung:

- Ablauf der Ü.: K.2
- Verteilung der Ü.-Aufgaben: 10
- Ü. von Rechenprozessen: 10

Unterbrechungen: 10, 16, 17

- Ankunftsregister von Ü., Maskierung von Ü.: 17

Verarbeitungsregister: Akkumulator

virtuelle Maschine: 4, 6

- die v.M. CODE: K.3
- Architektur der virtuellen CODE-Maschine: K.3.2.1
- abstrakte Form der virtuellen CODE-Maschine: K.3.2
- Konkretisierung der " " : K.3.3
- Quellsprachunabhängigkeit der Zwischensprache CODE: 7

Vorbelegung: 32

Wolkentechnik: 14

Wortbefehl: K.3.3.3.2-3.3.3.4

Wortlänge: 63, 64, 76, 80, Aufweitung

Zeichenketten: 21

Zielcode-Index: 73

Zielcode(sequenz)-Muster: 62, K.3.3.3.3 ab S.71, K.3.3.3.4, (18)

Zielcodesequenz-Elementarten: K.4.2.2

Zielrechner: (6, 7)

- Abstraktion von Z.n: 4
- Spektrum der Z.: 2

Zielrechnerunabhängigkeit:

- Z. der Zwischensprache: 7, 10
- Z. des Compileroberteils: 10, 11

Zwischensprache:

- die Z. CODE: K.3
- Sprachebene einer Z.: 1, 4
- Überarbeitung der Z.-Schnittstelle: 6

In letzter Zeit sind folgende Arbeitsberichte des Institutes für
Mathematische Maschinen und Datenverarbeitung erschienen:
=====

Band 11
=====

- Nr. 1 Körber, K., Werzinger, G.:
 Das Modell SIM-LAB.
 Beschreibung des Grundmodells.
 (Februar 1978)
- Nr. 2 Körber, K., Werzinger, G.:
 Das Modell SIM-LAB.
 Beschreibung der Modellerweiterung und der Strategien.
 (Februar 1978)
- Nr. 3 Körber, K., Werzinger, G.:
 Das Modell SIM-LAB.
 Listing der Quellprogramme.
 (Februar 1978)
- Nr. 4 Beth, Th.:
 On Resolutions of Steiner Systems.
 (März 1978)
- Nr. 5 Leeb, K.:
 Salami-Taktik beim Quader-Packen.
 Riess, W.:
 Zwei Optimierungsprobleme auf Ordnungen.
 (April 1978)
- Nr. 6 Händler, W., Henning, W., Kleinöder, W., Kneissl, J.,
 Volkert, J.:
 EPIK - Erlanger Projekt Interaktive Kartographie.
 (April 1978)
- Nr. 7 Schreiber, H.:
 Hardware-Messung und Analyse des Ablaufgeschehens in
 Rechnerkernen
 (April 1978)

- =====
- Nr. 8 Weber, D.:
Datengraphen und deren Transformation:
Ein Konzept zur Spezifikation von Datentypen.
(Juni 1978)
- Nr. 9 Musielak, H., Schmidt, B., Stössel, M.:
Vergleich von Simulationssprachen:
GPSS-FORTRAN - GPSS - GASP IV.
(Juli 1978)
- Nr. 10 Solymosi, A.:
Synthese von analysierenden Automaten auf Grund von
formalen Grammatiken.
(Juli 1978)
- Nr. 11 Meyer, K., Schaller, K., Schmidt, B.:
SIM-QUEUE.
Beschreibung eines Simulationsmodells für ein kom-
plexes Warteschlangensystem, das mit Hilfe von Eingabe-
daten zusammengestellt werden kann.
(Juli 1978)
- Nr. 12 Bodendorf, F., Eckardt, T., Mertens, P., Schrammel, D.:
Benutzung und Beurteilung von Kleinrechnersystemen in einer
Hochschule - Ein Beitrag zur Dezentralisierungsdiskussion
(August 1978)
- Nr. 13 Ravinovitz, M.:
An Approach to a Reconfigurable Multiprocessor System.
Bode, A., Händler, W.:
Classification d'architectures parallèles:
Introduction de la notation ECS et application au
projet EGPA
Sigmund, W.:
Parallel Machine Functions in Program Transformation
and Execution.
(September 1978)
- Nr. 14 Beth, Th., Strehl, V.:
Materialien zur Codierungstheorie
(September 1978)

- Nr. 15 Wendler, K.:
Betriebssystemaspekte in hierarchisch modularen Poly-
prozessorsystemen - Modellierungsansätze und Koordinie-
rungsmechanismen
(Oktober 1978)
- Nr. 16 Altmann, W.:
Beschreibung von Programmoduln zum Entwurf zuverlässiger
Softwaresysteme
(Oktober 1978)
- Nr. 17 Hofmann, W.:
Warteschlangenmodelle für die Parallelverarbeitung
(November 1978)
- Nr. 18 Stöth, G.:
Höhere Programmiersprachen für Kleinrechner:
Anforderungen, Entwicklungskriterien, Compileraspekte
(November 1978)
- Nr. 19 Bode, A., Händler, W.:
Rechnerarchitektur - Grundlagen und Verfahren
(November 1978)
- Nr. 20 Kneissl, F.:
Grundsoftware für die Textverarbeitung
(Dezember 1978)
- Nr. 21 Leidel, W.:
Synchronisationsprobleme in Automatennetzen
(Dezember 1978)
- Nr. 22 Bodendorf, F., Renninger, W., Schaller, K.:
Zwei neue Methodenbanksysteme für statistische Anwendungen
(Dezember 1978)
- Nr. 23 Jahresbericht 1978 der Informatik
(März 1979)

- Nr. 1 Elzer, P.:
Strukturierte Beschreibung von Prozesssystemen
(Februar 1979)
- Nr. 2 Wunderatsch, H.:
Deterministisches Zuordnen in heterogenen Prozessor-
systemen mit Pseudo-Boole Methoden
(Juni 1979)
- Nr. 3 Bunke, H.:
Sequentielle und parallele programmierte Graph-Grammatiken
(Juni 1979)
- Nr. 4 Luft, A.L.:
Zur ingenieurwissenschaftlichen Theorie von Rechner-
systemen
(August 1979)
- Nr. 5 Schmidt, B.:
Das Simulationsmodell SIM 330 - Einführung
(August 1979)
- Nr. 6 Gardill, R.:
SKOP, Syntaxnotation als Kontrollstruktur für
Programmiersprachen
(September 1979)
- Nr. 7 Keramidis, S.:
Entwurf und Implementierung eines Rechnerverbund-
netzes für die Fertigungssteuerung.
Fallstudie zur Kontrolle zuverlässiger Systeme
(September 1979)
- Nr. 8 Widjaja, H.:
Die optimale Zerlegung von Netzen und die Optimierung
des Prozeßverhaltens im Adressraum
(Oktober 1979)
- Nr. 9 Keramidis, S., Grothe, W.:
Beiträge zur Lösung des Verklemmungsproblems in prioritäts-
freien Betriebsmittelmanchinen
(Oktober 1979)
- Nr. 10 Beth, Th., Hain, M., Sagerer, G. Schäfer, N.:
Materialien zur Codierungstheorie II (Dezember 1979)
- Nr. 11 Jahresbericht 1979 der Informatik (März 1980)

- Nr. 1 Schrammel, D., Bayer, W., Gliemann, H.-P., Heigl, M., Mertens, P.:
Ein Leistungs- und Wirtschaftlichkeitsvergleich zwischen Klein- und Großcomputern
(Januar 1980)
- Nr. 2 Göttler, H.:
Semantische Modelle
(März 1980)
- Nr. 3 Fromm, H.J.:
Zur Modellierung der Speicherinterferenz bei hierarchisch organisierten Multiprozessorsystemen
(April 1980)
- Nr. 4 Jäpel, D.:
Klassifikatorbezogene Merkmalsauswahl
(Juli 1980)
- Nr. 5 Bodendorf, F., Mertens, P., Haas, R., Weber, G. Wolf, G.:
SAMBA - ein Methodenbankrahmen um das Statistik-Paket SPSS
(Juli 1980)
- Nr. 6 Bartsch, B.:
Inferenz und Analyse spezieller Graphgrammatiken für die syntaktische Mustererkennung
(September 1980)
- Nr. 7 Wurm, F.X.:
Auftragssystem für eine Mehrprozessoranlage
(Oktober 1980)
- Nr. 8 Schedelbeck, A.:
Vergleich der beiden höheren Realzeitprogrammiersprachen ADA und PEARL.
(Oktober 1980)
- Nr. 9 Jahresbericht 1980 der Informatik.
(März 1981)

- Nr. 1 Pattern Recognition
(Research at Lehrstuhl für Informatik 5)
(Februar 1981)
- Nr. 2 Bunke, H.:
Analyse elektrischer Schaltpläne
(März 1981)
- Nr. 3 Linster, C.-U.:
SYMPOS/UNIX - Ein Betriebssystem für homogene
Polyprozessorsysteme
(Juni 1981)
- Nr. 4 Akyildiz, J.F., Bolch, G.:
Analytic Solution Techniques for Queueing Network
Models of Computer Systems
(Juli 1981)
- Nr. 5 Lindstedt, W.G.:
Ein Verfahren zur Erstellung portabler algorithmischer
Laufzeitprogramme
(Juli 1981)
- Nr. 6 Vollmar, R.:
Zum Begriff des Parallelismus bei Polyautomaten
(Juli 1981)
- Nr. 7 Luft, A.L.:
Rationaler Sprachgebrauch und orthosprachliche
Standardisierung als Grundlagen zuverlässiger Software-
Entwicklung und -Dokumentation
(Januar 1982)

Band 14 Fortsetzung

=====

Nr. 8 Händler, W.:

PROCEEDINGS OF A WORKSHOP ON TAXONOMY IN COMPUTER
ARCHITECTURE

(Februar 1982)

Nr. 9 Beth, Th., Hess, P., Wirl, K.:

Materialien zur Kryptographie

(März 1982)

Nr. 10 Jahresbericht 1981 der Informatik

(März 1982)

- Nr. 1 Wittmann, A.:
Ein Mechanismus für die Synchronisation paralleler
Prozesse
(Februar 1982)
- Nr. 2 Maehle, E.:
Fehlertolerantes Verhalten in Multiprozessoren -
Untersuchungen zur Diagnose und Rekonfiguration
(März 1982)
- Nr. 3 Bathelt, P.:
Vergleich von Synchronisationsmechanismen
(Mai 1982)
- Nr. 4 Keramidis, S.:
Eine Methode zur Spezifikation und korrekten
Implementierung von asynchronen Systemen
(Juni 1982)
- Nr. 5 Fromm, H.J.:
Multiprozessor-Rechenanlagen:
Programmstrukturen, Maschinenstrukturen und Zuordnungs-
probleme
(Mai 1982)
- Nr. 6 Bley, H.:
Vorverarbeitung und Segmentierung von Stromlaufplänen
unter Verwendung von Bildgraphen
(Juli 1982)
- Nr. 7 Hohl, W.:
Informatik-Sammlung
Katalog mit historischen Erläuterungen
(August 1982)
- Nr. 8 Messerer, M.:
Ein neuer Ansatz zur Parallelisierung von
Compilern
(August 1982)

Band 15 Fortsetzung

=====

Nr. 9 Grosch, J.:

Eine Programmiersprache mit mengentheoretischen
Konstrukten und deren effiziente Implementierung
(August 1982)

Nr.10 Kleinöder, W.:

Stochastische Bewertung von Aufgabenstrukturen
für hierarchische Mehrrechnersysteme
(August 1982)

Nr.11 Kneißl, F.:

Realisierung von Makro-Datenflußmechanismen auf
hierarchischen Mehrrechnersystemen
(August 1982)

Nr.12 Brendel,W.,Fan,Z.,Klar,R.,Schmielau,W.:

ERES 82
Handbuch und Fallstudie
Handbook and Case Study
(Oktober 1982)

