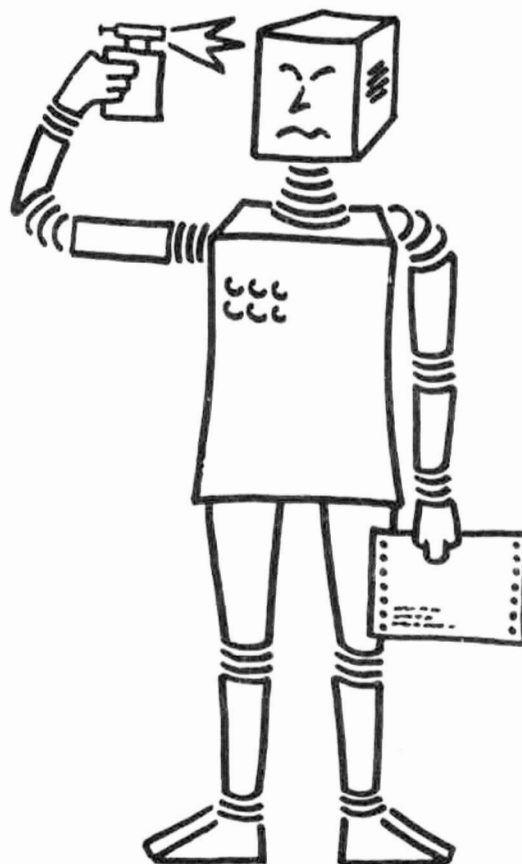


SEKI-Projekt

**SEKI
MEMO**

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



Entwurf und Implementierung von CSSA..

Teil A: Konzepte

MEMO-SEKI-82-03-A

Christian Beilken,
Friedemann Mattern,
Michael Spenke

Entwurf und Implementierung von C S S A
– Beschreibung der Sprache, des Compilers
und des Mehrrechnersimulationssystems

Teil A : Konzepte

Christian Beilken
Friedemann Mattern
Michael Spénke

Hofstadter's Law

It always takes longer than
you expect, even when you take
into account Hofstadter's Law.

Design and Implementation of C S S A
- Description of the Language, the Compiler
and the Multiprocessor Simulation System

Volume A:
Concepts

Abstract:

CSSA (Computing System for Societies of Agents) is an interactive programming language for asynchronous multiprocessor systems. Computations are done by concurrently working sequential modules, called agents which implement objects of data and control abstractions.

Communication is done by passing messages to acquainted agents. The receiving agent creates an instance of an operation capability referred to in the message. Since agents can be created during the computation and acquaintances can be transmitted in messages the heterarchical agent-net may dynamically change.

The CSSA-language has been successfully implemented for a multiprocessor simulation system running on a general purpose computer. This system allows the execution of CSSA-applications on a wide range of simulated multicomputer configurations.

This volume gives an overview of the language concepts and design criteria of CSSA. Some major concepts of compiler construction used in the implementation are also discussed. A practical automatic error recovery method for LL(1)-parsers is described in the appendix.

After a short description of the user-interface, the menu-driven program development system and the interactive debugger follows a detailed discussion of the multiprocessor simulation system.

Other volumes of the CSSA-documentation contain the language reference manual, the user's guide, programming examples and the system documentation.

Vorwort

CSSA (Computing System for Societies of Agents) ist eine Programmiersprache für autonome asynchron parallele Prozesse, die untereinander durch Übersenden von Nachrichten kommunizieren. Eine sequentielle Vorversion (CSSA-S) wurde 1979 implementiert, seit 1981 steht eine stark revidierte und erweiterte Fassung (CSSA-0) für Mehrprozessorsysteme zur Verfügung.

Die Sprache CSSA wurde im Rahmen mehrerer Diplomarbeiten des Instituts für Informatik III der Universität Bonn definiert und implementiert, diese Arbeiten wurden von Prof. Raulefs sowie von H.L. Fischer und H. Voss betreut. Die Implementierung des hier beschriebenen Systems konnte jedoch letztlich nur dank der freundlichen Unterstützung der Gesellschaft für Mathematik und Datenverarbeitung durch Bereitstellung der notwendigen Ressourcen in den Jahren 1980-82 gewährleistet werden.

Der im Rahmen dieser Arbeit realisierte CSSA-Übersetzer ist in ein Multiprozessor-Simulationssystem integriert und erzeugt Code, der von diesem Simulationssystem ausgeführt wird. Das System erlaubt nicht nur die realitätsgetreue Simulation einer Vielzahl unterschiedlicher Rechnerkonfigurationen, sondern ermöglicht auch eine systemglobale Ablaufüberwachung und Protokollierung mit wählbarem Detaillierungsgrad. Zur Unterstützung des Anwenders werden außerdem ein Programmentwicklungssystem und ausführliche quelltextbezogene Debugging- und Tracingmöglichkeiten zur Verfügung gestellt. Erste Anwendungen und Erfahrungen mit dem auf der Siemens 7760 der GMD unter BS2000 implementierten System liegen bereits vor.

Die vorliegende Arbeit beschreibt in mehreren Teilen die CSSA-Sprache mit ihren Konzepten, den Compiler, das Mehrrechnersimulationssystem, das menügesteuerte Programmentwicklungssystem und die weiteren Systemkomponenten bestehend aus der interaktiven Benutzerschnittstelle mit dem CSSA-Kommandointerpreter ("interface-Agent"), dem lokalen Betriebssystemkern mit Laufzeitsystem und virtueller Zwischencodemaschine sowie dem interaktiven debugger.

Die Beschreibung des Gesamtsystems wurde unter Berücksichtigung der unterschiedlichen Interessenschwerpunkte in fünf Teile aufgeteilt: Teil B beschreibt weitgehend implementierungsunabhängig die CSSA-Sprache; Hinweise zur Benutzung des Compilers und der weiteren Systemkomponenten zur Entwicklung und Ausführung von CSSA-Programmen werden in Teil C gegeben. Einer Sammlung von Beispiel- und Testprogrammen ist ein eigener Band gewidmet. Die Programmdokumentation findet sich in dem aus zwei Bänden bestehenden Teil E.

Der vorliegende Teil A behandelt die globaleren Aspekte der Arbeit. Er beinhaltet u.a. eine kurze Diskussion einiger wesentlicher CSSA-Sprachkonzepte sowie eine Beschreibung des realisierten Mehrrechnersimulationssystems und einiger interessanter Aspekte des Compilers und des interface-Agenten. Dabei wird auch auf die zur Anwendung kommende automatische Syntaxfehlerbehandlungsmethode und das Prinzip der Codegenerierung über Zwischencodeschichten eingegangen. Eine ausführliche kommentierte Literaturliste beschließt diesen Teil.

BMS, im September 1982

Inhalt

1. Überblick	2
2. Inhaltsübersicht aller Teile	7
3. Rahmenbedingungen und Ziele der CSSA-Entwicklung	9
4. CSSA-Sprache	14
4.1 Sprachkonzept	14
4.1.1 Höhere Sprachmechanismen	15
4.1.2 Sequentielle Sprachmechanismen	17
4.1.3 Benutzerintegration	19
4.2 Ein Beispiel: Euklidischer Algorithmus	20
4.3 Vergleich mit anderen message-passing Konzepten	48
4.3.1 Formen der Kommunikation über Nachrichten	48
4.3.2 Das Philosophenproblem in message-passing Systemen	51
4.3.3 Vergleichskriterien für die verschiedenen Konzepte	52
4.3.4 PLITS	55
4.3.5 CSP	57
4.3.6 CSSA	60
4.3.7 DP	62
4.3.8 Ada	65
4.3.9 Zusammenfassung	69
5. Sprachentwurf	71
5.1 Syntaktische Aspekte	71
5.2 Lokale Kontrollstrukturen	74
5.3 Relationen	77
6. Compilerbautechnik	80
6.1 Symboltabellenorganisation	80
6.2 Parser	89
6.3 Fehlerbehandlung	92
6.3.1 Fehlermeldungen	93
6.3.2 Fehlerbehandlungsmethoden	95
6.3.3 Anpassung an den rekursiven Abstieg	96
6.4 Semantische Überprüfungen	97
6.5 Codegenerierung	101
6.6 Laufzeitorganisation	109
7. Benutzerschnittstelle	114
7.1 Das Entwicklungssystem	114
7.2 Der Compiler	118
7.3 Die interaktive CSSA-Sitzung	122
8. Das Simulationssystem	131
8.1 Simulation der hardware	133
8.2 Simulation des Betriebssystems	136
8.3 Steuerung des Simulationssystems	148
8.4 Beispiele	150
9. Literaturverzeichnis und Dokumentation	156
Anhang: Ein sprachunabhängiges Verfahren zur Fehlerbehandlung für LL(1)-Sprachen, Vortrag WASC0-Tagung Budapest	

1. Überblick

Das **Ziel der vorliegenden Arbeit** bestand in der Definition der Sprache CSSA und der Realisierung eines Übersetzers.

CSSA ist eine Programmiersprache für **autonome asynchron parallele Prozesse**, die untereinander ausschließlich über Nachrichten kommunizieren. 1979 lag bereits eine **sequentielle Vorversion** und eine auf LISP beruhende Implementierung vor ([FI77], [AUG79], [GLÜ79]). Dieser Entwurf diente als Ausgangsbasis für die zu definierende Parallelversion der Sprache, wobei jedoch nur die grundsätzlichen Konzepte (wie scripts, Agenten und Agentennetz, interface-Agent, Facettierungsmechanismus, Unteilbarkeit von Operationen) von der sequentiellen Vorversion in angepaßter Form übernommen wurden, die restlichen Sprachmechanismen jedoch völlig neu entwickelt wurden.

Beim **Sprachentwurf** wurden die neueren Erkenntnisse des Programmiersprachen-Designs berücksichtigt (vgl. bspw. [DOD77] und [WEI78]), sowohl was die sequentiellen Sprachkonstrukte betrifft, als auch was die Kommunikationsmechanismen und das rein syntaktische Layout angeht. So wurde etwa auf einen Sprungbefehl ganz verzichtet, statt dessen jedoch einige Konstrukte zum kontrollierten Verlassen von Strukturen und eine ereignisgesteuerte Schleife hinzugenommen (vgl. [KNU74]); es wird zwischen Prozeduren mit eventuellem Seiteneffekt und Funktionen ohne Seiteneffekt unterschieden, und nicht zuletzt wurde ein strenges Typkonzept eingeführt (vgl. [WSH77]).

Beim Entwurf der **Kommunikationsmechanismen** wurden eine Reihe neuerer Sprachentwürfe untersucht, die ebenso wie CSSA auf dem module-message-Paradigma beruhen (u.a. [BRI78], [HOA78], [DOD80], [FEL79]). Allerdings ist hier die "öffentliche Diskussion" noch in vollem Gange (vgl. etwa [WEL81], [SIL81a]), die Ansätze sind z.T. recht unterschiedlich (synchronisierte bzw. asynchrone Kommunikation, handshaking, dynamische bzw. statische Prozeßmenge etc.), und eine Bewährung und Bewertung in der Praxis steht noch aus. Daher wurden die CSSA-Kommunikationsmechanismen weitgehend unbeeinflußt von anderen Konzepten entwickelt. Um so interessanter sind allerdings einige neuere Arbeiten, in denen unabhängig ganz ähnliche Konzepte vorgeschlagen werden (vgl. etwa [BPS81], [COOK81], [GEN81], [HAR81], [HUNT79], [MAY80]) oder die Schwachstellen alternativer Kommunikationsmechanismen aufgezeigt werden (z.B. [KIS79], [STR80]).

Auf ein gutes **syntaktisches Layout** wurde besonderer Wert gelegt. Äußerlich orientiert sich CSSA an der bewährten ALGOL/PASCAL-Tradition: Die Sprache enthält eine zu den Bezeichnern disjunkte Menge von Schlüsselwörtern. Allerdings existiert etwa im Unterschied zu diesen Sprachen zu jedem Schlüsselwort, das eine Klammerstruktur einleitet, ein spezifisches "schließendes" Schlüsselwort (siehe auch [KOV78]); vor jedem Statementteil einer Klammerstruktur können durch ein explizites Trennsymbol abgesetzt Deklarationen stehen, und das Semikolon fungiert als Terminator und nicht als Separator. Die Vorteile dieser und weiterer Eigenschaften des syntaktischen Layouts von CSSA für

Benutzer und Compiler werden an anderer Stelle dieser Arbeit diskutiert.

In Anbetracht der Tatsache, daß CSSA für Multi-Mikrocomputersysteme Verwendung finden soll, wo die Ressourcen beschränkt sind und noch kaum hochentwickelte Hilfsmittel für den Softwareentwurf zur Verfügung stehen, wurde beschlossen, den Compiler nach der Methode des rekursiven Abstiegs zu programmieren und anstelle von Maschinencode **Zwischencode** für eine virtuelle stack-Maschine zu generieren. Die gelungene Implementierung von PASCAL nach dieser Methode (→ [PED82], [HII77], [WIR77]) rechtfertigt diese Vorgehensweise. Es wurde daher eine **LL(1)-Grammatik** für CSSA entworfen und der **one-pass-Übersetzbarkeit** dadurch Rechnung getragen, daß das define-before-use-Prinzip als Bedingung der statischen Semantik formuliert wurde.

Anfangs war geplant, CSSA auf der Multi-Mikrocomputermaschine SMS von Siemens oder einer Mehrrechnerkonfiguration aus Prozessoren der Serie /1 von IBM zu implementieren. Da die hardware jedoch nicht unmittelbar zur Verfügung stand, wurde auf einem Großrechner zunächst ein **Testcompiler** für eine Teilmenge von CSSA realisiert, der in SIMULA programmiert war und CSSA-Programme in äquivalente SIMULA-Programme übersetzte. Dieser Testcompiler lieferte in mehrerlei Hinsicht wertvolle Erfahrungen: Zum einen konnten einige CSSA-Konzepte hinsichtlich ihrer Verwendbarkeit und Implementierbarkeit getestet werden, zum anderen konnte mit einigen Compilerbautechniken experimentiert werden. Vor allem aber wurde deutlich, daß eine Implementierung von CSSA in einer **simulierten Mehrrechnerkonfiguration** eine Vielzahl von Vorteilen gegenüber einer ersten Realisierung auf einem realen Mehrrechnersystem besitzt.

Das schließlich konzipierte und realisierte System besteht aus einer Reihe funktionaler Komponenten, dazu gehören:

- der **CSSA-Compiler**, der CSSA-Quellprogramme in eine Zwischensprache übersetzt. Der Compiler ist in SIMULA geschrieben, arbeitet syntaxgesteuert nach der Methode des **rekursiven Abstiegs** und erzeugt code in einer Folge **virtueller Zwischensprachschichten**. Der letztlich generierte code ist ablaufinvariant und für eine **virtuelle stack-Maschine** konzipiert, die als Teil des Laufzeitsystems diesen in SIMULA-Syntax realisierten code ausführt. Eingestreut in den eigentlichen Maschinencode sind Anweisungen für das Mehrrechnersimulationssystem und den interaktiven debugger. Bei der Entwicklung des Compilers wurde Wert auf eine "freundliche" Benutzerschnittstelle gelegt (vgl. [HOR74]): Die verwendete **automatische Syntaxfehlerbehandlungsmethode** ist unter Berücksichtigung der Rahmenbedingungen (kein backtracking, one symbol lookahead) nahezu optimal; es werden bei nicht allzugroßer Fehlerhäufung alle Fehler gefunden und so gut wie keine Folgefehler gemeldet. Im **Layout des abgedruckten Quelltextes** sind u.a. Schlüsselwörter hervorgehoben, Fehlerstellen unterstrichen und syntaktische Klammerstrukturen durch Angabe der Verschachtelungstiefe und durch senkrechte Linien zwischen öffnendem und schließendem Schlüsselwort verdeutlicht.

- das **Mehrrechnersimulationssystem** als Teil des Laufzeitsystems, mit dem verteilte Berechnungen auf simulierten Multiprozessorkonfigurationen ausgeführt werden können. Hier werden die hervorragend geeigneten **Simulationskonzepte von SIMULA** extensiv genutzt. Aktive Einheiten der Simulation sind neben **Prozessoren** und **Busen** auch die vom Compiler aus den scripts generierten **Programm-Moduln**, welche vom simulierten Mehrrechnersystem unter Mitwirkung weiterer Komponenten des Laufzeitsystems (lokaler Betriebssystemkern, virtuelle stack-Maschine) ausgeführt werden. Die Charakteristika der zu simulierenden **hardware-Konfiguration** und Parameter für die wesentlichen Aspekte eines **verteilten Betriebssystems** können spezifiziert werden (wie bspw. Anzahl der Prozessoren und Busse, Netztopologie, Übertragungszeit, Zeitscheibengröße etc.), und diverse **Statistiken** können erzeugt werden. Damit ist es möglich, Informationen über den Grad der Parallelität sowie über die Auslastung der simulierten hardware-Komponenten bei unterschiedlichen Rechnerkonfigurationen zu gewinnen.
- der **lokale Betriebssystemkern**, der auf jedem Prozessor ein unterbrechungsgesteuertes multiprogramming-Betriebssystem realisiert und damit die quasi-parallele Ausführung mehrerer autonomer Prozesse ("Agenten") auf einem Prozessor ermöglicht. Realisiert wird u.a. das **scheduling** der Agenten mit **Zeitscheibenverwaltung**, das **code-sharing** von Agenten gleichen Typs, die Behandlung von **Betriebssystemaufrufen** (wie Gründen, Aktivieren und Beenden von Agenten, Senden von Nachrichten etc.) als interne Unterbrechungen und die Behandlung **externer Unterbrechungen** (wie bspw. der Empfang einer Nachricht).
- die **virtuelle stack-Maschine** als ein weiterer Teil des Laufzeitsystems, die auf jedem simulierten Prozessor die Operationen des vom Compiler erzeugten Zwischencodes zur Verfügung stellt. Hierzu existieren im Agent-Control-Block (ACB), der jedem Agent zugeordnet ist, zu allen Datentypen ein eigener stack, der somit Teil des lokalen Speichers eines Agenten ist. Die virtuelle stack-Maschine realisiert die bei einem Datentyp jeweils definierten Verknüpfungen sowie die zugehörigen push- und pop-Operationen.
- die **Laufzeitroutinen**, die Strukturdefinitionen und Verwaltungsroutinen für komplexere Datenstrukturen (arrays, records, sets, Relationen) und diverse Kontrollblöcke bereitstellen, sowie den code der built-in Funktionen realisieren.
- der **interaktive debugger**, der es gestattet, Informationen über den Zustand einzelner Agenten und des gesamten Agentennetzes in symbolischer und quelltextbezogener Form mit wählbarem Detaillierungsgrad zu gewinnen. Vorgesehen sind sowohl Möglichkeiten zur einmaligen Ausgabe von **Zustandsinformation**, als auch ein zeitweises **tracing** und eine fortlaufende **Protokollierung**. Durch den interaktiven debugger wird die Entwicklung und der Test von CSSA-Programmen wesentlich erleichtert.

- der **interface-Agent**, der eine integrierte interaktive **Benutzerschnittstelle** darstellt. Er erkennt die Kommandos zur Steuerung des Simulationssystems und des debuggers und ist dynamisch programmierbar, indem er eine Teilmenge der CSSA-Sprache interpretativ verarbeitet. Dadurch kann der Benutzer ablaufende Berechnungen beeinflussen – er ist über den interface-Agenten selbst Teil des Agentennetzes.

Die oben erwähnten funktionalen Komponenten des CSSA-Systems bilden nicht in jedem Fall physisch differenzierbare eigenständige Einheiten; der Compiler ist als ein eigenes Programm realisiert, die restlichen Systemkomponenten sind Teil des Laufzeitsystems.

Das CSSA-System wurde auf einem **IBM/370-Großrechner** unter OS/VS2 entwickelt; es steht nun auf der **Siemens 7760** der Gesellschaft für Mathematik und Datenverarbeitung (GMD) unter dem Betriebssystem BS2000 interessierten Anwendern zur Verfügung. Dank der weitgehenden Kompatibilität der beiden SIMULA-Systeme gab es kaum Portabilitätsprobleme. Für die Benutzung des CSSA-Systems der Siemens-Implementierung wurde ein **menügesteuertes Programmentwicklungssystem** konzipiert, das alle Systemkomponenten integriert und es gestattet, alle Arbeiten, die zur Entwicklung von CSSA-Anwendungen notwendig sind, ohne Kenntnisse der BS2000-Kommandosprache durchzuführen.

Beim **Test von Compiler und Laufzeitsystem** wurde in systematischer Weise vorgegangen: Für jedes Sprachkonstrukt wurden mindestens zwei **Testprogramme** entwickelt. Im ersten werden alle Fehler der statischen Semantik hervorgerufen, die im Zusammenhang mit dem Konstrukt möglich sind; in den weiteren Testprogrammen werden evtl. mögliche Laufzeitfehler provoziert und das normale Funktionieren des Konstruktes (auch bei unüblicher Verwendung) demonstriert. Einige dieser Testprogramme sind zusammen mit einer Reihe sinnvoller **Beispielprogramme** in Teil D abgedruckt.

Diese Beispiele zeigen auch, daß die **Implementierung von CSSA** - die schließlich den Hauptteil der Arbeit ausmachte - **gelingen** ist. Erste Anwendungen und Erfahrungen externer Benutzer liegen bereits vor.

As language designers we often feel that people believe that one has "succeeded" and that a language "exists" as soon as it is defined, and that the subsequent implementation is a mere triviality, even if time consuming. The implementors very often are only briefly mentioned in a preface to some report.

K.Nygaard, O-J. Dahl [NYD78]

Einige der im Rahmen dieser Arbeit entwickelten Methoden und Konzepte sind ausbaufähig und von eigenständigem Wert. Dazu gehören:

- Eine effiziente Methode zur automatischen **Syntaxfehlerbehandlung** für LL(1)-Sprachen. Die Methode wurde nicht nur theoretisch untermauert und mit anderen Verfahren verglichen, sondern zur empirischen Bewertung auch in einen PASCAL-Parser implementiert. Im Anhang von Teil A ist das Manuskript eines auf der diesjährigen WASC0-Tagung in Budapest hierüber gehaltenen Vortrages abgedruckt. Gegenwärtig wird an der GMD ein Zerteilergenerator auf der Basis dieser Methode entwickelt.
- Das **Prinzip der Simulation** von Mehrrechnersystemen mit lokalen multiprogramming-Betriebssystemkernen. Dadurch, daß der vom CSSA-Compiler generierte code genauso wie die zu simulierenden hardware-Komponenten in Form von aktiven **Simulationsobjekten** vorliegt, ist mit wenig Aufwand eine ausbaufähige und realitätsgetreue Simulation möglich, bei der im Unterschied zur gängigen Praxis die modellierte hardware nicht unter einer fiktiven Last getestet wird, sondern bei der **tatsächlichen Ausführung von Programm-Modulen**. Trotz der detaillierten Simulation ergeben sich **akzeptable Ausführungszeiten**, da der generierte code vom Simulationssystem nicht interpretiert wird, sondern als ein Gesamtprogramm in Maschinensprache weiterübersetzt wird. Der Betriebssystemkern kann als ein eigenständiges Simulationsobjekt realitätsgetreu, und falls gewünscht in Assembler-ähnlichem Stil, programmiert werden.
- Das **Relationenkonzept** als ein Konstrukt, durch das das Prinzip eines inhaltsadressierten Assoziativspeichers bereits auf Sprachebene zur Verfügung gestellt wird. Es stellt den Versuch dar, die Prinzipien des **Codd'schen Relationenmodells**, die sonst nur bei relationalen Datenbanken zur Anwendung kommen, in eine Programmiersprache zu integrieren.

Diese und einige weitere Konzepte gingen über das ursprüngliche Ziel der Arbeit hinaus. Sie stellen jedoch nicht nur Methoden und Verfahren von allgemeinem Interesse dar, sondern ermöglichten es überhaupt erst, daß das CSSA-System als ein **praktisch einsetzbares Experimentiersystem** sowohl für CSSA-Sprachkonzepte, als auch für CSSA-Anwendungen und -Implementierungen Verwendung finden kann.

The major error by designers and compiler writers is overambition. The result may be a great design but will surely be an unfinished project.

W.M. McKeeman, 1974

2. Inhaltsübersicht aller Teile

Die umfangreiche Beschreibung des Gesamtsystems mußte in mehrere Teile aufgeteilt werden; dies geschah unter Berücksichtigung der unterschiedlichen **Interessenschwerpunkte**. Für jemanden, der nur an der Sprache CSSA interessiert ist, mag Teil B genügen; ein Anwender des Systems benötigt zusätzlich Teil C. Zur Systemanpassung und Durchführung von Änderungen und Erweiterungen wird Teil E benötigt. Die im vorliegenden Teil A dargestellten Konzepte mögen nur bei einer Diskussion und Wertung der Sprache, der entwickelten Compilerbautechniken oder der Prinzipien des Laufzeit- und Simulationssystems interessieren; dagegen können die in Teil D abgedruckten Programmbeispiele auch für den lediglich an der eigentlichen CSSA-Sprache interessierten Leser von Bedeutung sein.

Die vorgenommene Aufteilung nach Anwendungsgesichtspunkten bringt naturgemäß einige **inhaltliche Überschneidungen** und **Wiederholungen** mit sich, jedoch wird der entsprechende Sachverhalt jeweils weitgehend differenziert unter dem dem jeweiligen Handbuch zugrundeliegenden spezifischen Aspekt diskutiert.

- **Teil A** behandelt i.w. einige **Konzepte** der CSSA-Sprache (u.a. das message-passing-Konzept im Vergleich zu anderen Programmiersprachen) und die im Compiler verwendeten Prinzipien zur automatischen Syntaxfehlerbehandlung und zur Codegenerierung über Zwischencodeschichten. Ein längeres Beispiel demonstriert nicht nur eine Reihe typischer Probleme verteilter Berechnungen, sondern erläutert auch einige Sprachkonstrukte in exemplarischer Weise und zeigt die Nützlichkeit der Programmentwicklungshilfen des CSSA-Systems. Es wird in diesem Band ferner auf einige Interna des Compilers eingegangen, wie die Symboltabellenorganisation, die systematische Überprüfung der statischen Semantik bei Bezeichnern mit Hilfe von Attributen und die Parsersteuerroutinen, die eine systematische Umsetzung der LL(1)-EBNF-Grammatik in einen fehlerkorrigierenden Parser gestatten. Weitere wesentliche Themen bilden das Mehrrechnersimulationssystem und das Konzept des lokalen Betriebssystemkerns; aber auch das Design der Benutzerschnittstelle wird behandelt. Eine ausführliche kommentierte Literaturliste und ein Vortragsmanuskript, in dem die im CSSA-Compiler zur Anwendung kommende Fehlerbehandlungsmethode theoretisch untermauert wird, beschließen diesen Teil.
- **Teil B** stellt eine vollständige und systematische Beschreibung der vom Compiler akzeptierten **CSSA-Sprache** dar. Das Handbuch soll einerseits einen leichten Zugang zu CSSA vermitteln, ist andererseits aber auch als ein Nachschlagewerk konzipiert. Es enthält neben einer detaillierten Diskussion aller Sprachkonzepte einen Sprachüberblick, kurze Programmfragmente zu allen Sprachkonstrukten und ein Schlagwortverzeichnis. Die Syntax von CSSA ist im Anhang in einer erweiterten BNF-Grammatik angegeben.

- **Teil C** beschreibt in ausführlicher Form den Umgang mit dem unter Siemens BS2000 laufenden **CSSA-System**. Dazu gehören das in Form einer Kommandoprozedur realisierte Entwicklungssystem, in das alle anderen CSSA-Komponenten integriert sind, der Compiler und das interaktive CSSA-System, bestehend aus interface-Agent, Mehrrechnersimulationssystem und interaktivem debugger. Auf die vom interface akzeptierte CSSA-Teilsprache einschließlich der Steuerkommandos und auf den Ablauf einer CSSA-Sitzung wird detailliert eingegangen. Das Handbuch enthält u.a. eine kommentierte Liste der Fehlermeldungen und im Anhang ein Beispiel-Sitzungsprotokoll sowie die EBNF-Grammatik der interface-Sprache.
- **Teil D** enthält **CSSA-Beispielprogramme und Testprogramme**. Die Beispielprogramme zeigen, wie kleine Probleme mit CSSA programmiert werden können, sie werden ergänzt durch eine kurze Problembeschreibung und ein Ausführungsprotokoll. Dabei werden auch die Möglichkeiten zur Steuerung und Überwachung des Systems durch interface-Kommandos demonstriert. Die Testprogramme zu einzelnen Sprachkonstrukten zeigen sowohl Semantikfehler als auch Laufzeitfehler und demonstrieren die Funktion des Sprachkonstrukts.
- **Teil E** beinhaltet in zwei Bänden die **Dokumentation des BMS-CSSA-Systems**. Dazu gehören neben einer allgemeinen Beschreibung vor allem die kommentierten Quellprogramme des Compilers und des Laufzeitsystems mit seinen Komponenten sowie die zur Ausführung von CSSA-Programmen benötigte Hilfssoftware. Ferner ist zu einem CSSA-Programm der generierte code mit Erläuterungen zur Repräsentation von CSSA-Konstrukten in der Zielsprache abgedruckt.

Designing an acronym is the first step in systems design. Many design teams manage without a resident full time acronymist. This is fatal pennypinching and explains the current low standards in dp acronymity.

Datamation, May 1981

3. Rahmenbedingungen und Ziele der CSSA-Entwicklung

Der rasante Fortschritt der VLSI-Technologie führt dazu, daß Mikroprozessoren und ganze hochentwickelte ein-Chip-Mikrocomputer als bloße Bauteile größerer Systeme angesehen werden können. Solche **Multimikrocomputersysteme** bieten durch einen hohen Grad an **Parallelität** bei relativ geringen Kosten eine enorme potentielle Leistung. Gleichzeitig entstehen auf dem Gebiet der **Kommunikationstechnik** neue Entwicklungen, wie bspw. Fiberglaslichtleiter, die eine sehr schnelle und preiswerte Breitbanddatenübertragung zwischen lokalen oder globalen Rechnernetzen ermöglichen.

Obwohl die rein technischen Voraussetzungen für derartige Mehrrechnersysteme aus tausenden von Prozessoren schon in naher Zukunft gegeben sein werden, sind damit zusammenhängende **organisatorische Probleme zur Nutzbarmachung der potentiellen Leistung** solcher Systeme noch weitgehend ungelöst. "The mere possibilities inherent in very-large-scale integration do not themselves create adequate ways of exploiting this technology", meint bspw. J.T. Schwartz dazu [SCH80].

So sind schon grundlegende Fragen der Rechnerarchitektur, wie bspw. die geeignete Topologie des Kommunikationsnetzes, nicht nur durch physikalische Gegebenheiten bedingt, sondern hängen auch in hohem Maße von den zu erwartenden **Anwendungen** und ihrem Betriebsmittelbedarf (z.B. Kommunikationsaufkommen) ab. Andererseits fällt es schwer, Anwendungen zu konzipieren, solange die grundlegenden Parameter derartiger Systeme unbekannt sind und konzeptuelle Hilfsmittel wie höhere Programmiersprachen und angemessene Berechnungsmodelle fehlen.

Jedenfalls stellen derartige **Parallelrechner in Form verteilter Systeme**, die eine starke Erweiterung des klassischen Universalrechnerprinzips bedeuten, eine große Herausforderung an viele Teilgebiete der Informatik dar:

- In den Bereich der **Rechnertechnologie** fällt die Spezifikation geeigneter VLSI-Bausteine (ein-Chip-Rechner), die mit einer relativ geringen Zahl externer Anschlüsse in flexibler Weise kombinierbar sein sollen und ein gewisses Maß an Fehlertoleranz besitzen müssen.
- Die **Rechnerorganisation** muß sich nicht nur mit Fragen der **Topologie** beschäftigen (hier existieren so unterschiedliche Vorschläge wie Ringkonfigurationen, Hyperkuben, spannende Bäume, perfect-shuffle-Strukturen etc.), sondern auch geeignete **Kommunikationstechniken** untersuchen. Auch für Multimikrocomputersysteme, die als räumlich konzentrierte lokale Rechnernetze realisiert sind, kommen bspw. Prinzipien in Frage, wie sie für Netze geographisch verteilter Großrechner oder für sogen. "local area networks" (LAN) entwickelt wurden (Breitbandkommunikation, Zeitmultiplexbetrieb, Datenpaketvermittlung, CSMA-Protokolle etc.).

- Die **Betriebssystemtheorie** muß insbesondere den **Verteilungsaspekt der Kontrolle** berücksichtigen: Eine zentrale Kontrolle des Gesamtsystems in einem einzigen Rechnerknoten oder kleinen Teilnetz würde genauso wie auf der Hardwareseite ein gemeinsamer Speicher einen untragbaren Engpaß darstellen. Die notwendigerweise **inkonsistente Systemsicht** verteilter Betriebssysteme (ein Aspekt, der bisher nur bei verteilten Datenbanken eine wesentliche Rolle gespielt hat), schafft gewisse Probleme bei der globalen Betriebsmittelverwaltung. Für das verteilte **task-Scheduling** wurden einige Verfahren (wie bspw. die "wave scheduling"-Strategie) vorgeschlagen, die jedoch erst analytisch untersucht und noch nicht empirisch erprobt wurden. Weitere Untersuchungsgegenstände sind Optimierung der Routingstrategien und dynamische Rekonfiguration der Kommunikationswege zur Minimierung der Nachrichtenlaufzeit und Erhöhung der Fehlertoleranz, sowie das Problem der Modul-Prozessor-Zuweisung in statischer ("static module assignment problem", S-MAP) oder sogar dynamischer Weise.

Die **Verwaltung des Kommunikationssubsystems** von Multimikrorechnersystemen ist eine Betriebssystemaufgabe, wie sie bei den klassischen Großrechnern nicht existierte, sondern bisher erst bei Rechnernetzen eine Rolle gespielt hat. Hierzu gehören bspw. Probleme der Sicherheit und Synchronisation (acknowledgment, handshaking, time-outs etc.), der Sequentialisierung, der Adressierung und der Fluß- und Staukontrolle (wie bspw. window-Mechanismen). Die **Fehlerbehandlung** spielt eine wichtige Rolle, da von Multimikrocomputern nicht nur eine wesentliche Erhöhung der Leistung erwartet wird, sondern auch eine Steigerung der Zuverlässigkeit: Die vorhandene Hardwareredundanz soll von der Systemsoftwareseite aus jedenfalls ausgenutzt und unterstützt werden.

- Die **theoretische Informatik** muß im Bereich der Algorithmentheorie die inhärente und maximale Parallelität von Algorithmen untersuchen. Hier stellen sich etwa folgende Fragen: Welche Verfahren lassen sich weitgehend vollständig in unabhängige Teiloperationen aufteilen (bspw. "divide et impera", branch and bound und ähnliche Prinzipien)? Wie groß und wichtig ist die Klasse der nicht parallelisierbaren Algorithmen? Wie ist ein realistisches Komplexitätsmaß zu definieren, wenn eine (theoretisch) unbegrenzte Menge von Prozessoren zur Verfügung steht?
- Dem Bereich des **Programmiersprachen-Designs** schließlich fällt die Aufgabe zu, durch Entwurf eines geeigneten semantischen Rahmenkonzeptes in Form einer Programmiersprache diejenigen höheren Strukturen bereitzustellen, die eine einfache und dem menschlichen Denken angepaßte Handhabung und **Nutzbarmachung der Parallelität** für die Anwendungsentwicklung ermöglichen. Die bei der sequentiellen Programmierung weitgehend unbekannten Aspekte der Parallelität, des Nichtdeterminismus und der Synchronisation sind konzeptuell nicht ganz einfach zu beherrschen und stellen hohe Ansprüche an die Anwendungsfreundlichkeit der Programmiersprache und damit an die Pragmatik und Semantik einzelner Sprachkonstrukte.

Eine neu entwickelte Programmiersprache sollte in jedem Fall auch die neueren Erkenntnisse des **software-engineerings** berücksichtigen und die erkannten Entwurfsfehler und Schwächen älterer Programmiersprachen vermeiden. Bei **CSSA** ist es gelungen, Modularisierungskonzepte und Datenabstraktionskonzepte in uniformer und integrativer Weise mit Verteilungsaspekten und Synchronisationsaspekten der Parallelität zu verbinden. Auf die engeren Ziele und die Sprachkonzepte von **CSSA** wird weiter unten und im folgenden Kapitel eingegangen.

Viele der genannten Aspekte beeinflussen sich gegenseitig. Insbesondere ist zu erwarten, daß Anforderungen von Programmiersprachen- und Betriebssystemseite sich direkt auf den VLSI-Entwurf auswirken. Noch ist relativ unklar, welche Anwendungen sich besser und welche sich weniger gut zur Parallelisierung eignen, und welche Rolle dabei der **Kommunikationsoverhead** spielt: Die Kommunikationskosten hängen in hohem Maße von der verwendeten Technologie ab, und es ist gegenwärtig noch nicht absehbar ob der Kommunikationsengpaß in naher Zukunft weitgehend eliminiert werden kann. Lint und Agerwala [LIA81] sind jedenfalls der Meinung, daß bei verteilten Berechnungen Kommunikationsaspekte unter Effizienzgesichtspunkten mindestens genauso wichtig wie die reinen Berechnungsaspekte sind.

Für eine Reihe **mathematischer Berechnungen**, die von relativ großer Bedeutung für physikalische und technische Anwendungen sind, wurden bereits vor längerer Zeit parallele Algorithmen entwickelt. Es handelt sich dabei jedoch vor allem um Matrizenoperationen (wie bspw. Invertierung und Multiplikation) oder einige andere numerische und rechenintensive Verfahren (Fourier-Transformation), die die Lösung großer linearer Gleichungssysteme und einiger Differentialgleichungssysteme (für Optimierungsprobleme, wie bspw. die finite-Elemente-Methode) ermöglichen. Für derartige **homogene Anwendungen** existieren schon seit einiger Zeit die nach dem pipelining-Prinzip arbeitenden Vektormaschinen (wie bspw. CYBER, CRAY) und die sogenannten **Feldrechner** (array-Prozessoren), mit denen vor allem Vektor- und Matrizenoperationen effizient berechnet werden können. Diese Rechner sind für den vorgesehenen Anwendungsraum von **CSSA** relativ **uninteressant**, da alle Einzelprozessoren ("processing elements") weitgehend synchron arbeiten und sich i.a. eine globale Rechenphase mit einer allgemeinen Datenaustauschphase abwechselt.

Im Unterschied dazu wurde **CSSA** zur Programmierung verteilter Berechnungen auf **asynchron** arbeitenden Mehrrechnersystemen entworfen. Es wird von schwach gekoppelten Systemen ohne gemeinsamen Speicher ausgegangen, wobei die Kommunikation über Nachrichtenaustausch erfolgt. Da von der Sprachseite aus keine Annahmen über die Nachrichtenlaufzeit gemacht werden, und sich Nachrichten insbesondere auch überholen können, kann **CSSA** in idealer Weise auf Systemen mit einem intelligenten Kommunikationssystem aus passiven oder aktiven Bussen mit einer gewissen Speicherkapazität (wie bspw. store-and-forward-Systeme, Paketvermittlungssysteme etc.) implementiert werden.

CSSA eignet sich besonders zur Programmierung algorithmischer Verfahren für **heterogene Systeme** durch **kooperierende asynchrone Prozesse**. Hierzu gehören insbesondere viele Anwendungen aus dem Bereich der künstlichen Intelligenz wie verteilte Expertensysteme, automatische Beweiser, natürlich-sprachliche Verstehenssysteme etc.; also Systeme, in denen – meist in Verbindung mit heuristischen Suchverfahren – die "lockere" Zusammenarbeit verschiedener autonom arbeitender Instanzentypen gefordert wird und weder die Kommunikationsintensität noch die Dauer lokaler Berechnungen von vornherein exakt abgeschätzt werden können und das System sich im Laufe der verteilten Berechnung vergrößern oder verkleinern kann.

Es scheint, daß **schwach gekoppelte verteilte Mehrrechnersysteme**, wie sie vom CSSA-Konzept unterstützt werden, für viele Anwendungen (und nicht nur für die oben genannten "task-force"-Anwendungen heterogener Systeme) ein sinnvolles Rechnerkonzept darstellen, u.U. mit geeigneter Systemsoftware sogar ein Mehrbenutzer-Universalrechnerkonzept: Derartige Rechner bieten nicht nur eine hohe Leistung mit kostengünstigen Mitteln und durch die große Redundanz und Dezentralisierung der Systemkontrolle ein hohes Maß an Ausfalltoleranz, sondern auch eine gegenüber anderen Rechnerstrukturen einfacher konzipierte Systemsoftware [--> GIL81] und Möglichkeit der dynamischen Systemerweiterbarkeit. Allerdings liegen bisher noch keine praktischen Erfahrungen vor, so daß eine Bewertung der vielen vorgeschlagenen Modelle und Realisierungsvorstellungen zum gegenwärtigen Zeitpunkt noch nicht möglich ist.

Die **fehlende praktische Erfahrung** macht sich nicht nur bei Rechnerorganisationsfragen negativ bemerkbar: Auch im Programmiersprachenbereich werden vielfältige Konzepte vorgeschlagen und – hier insbesondere die Kommunikations- und Synchronisationsmechanismen und ihr Abstraktionsgrad – ausführlich diskutiert (bezüglich ADA etwa [WEL81], [SIL81]), ohne daß ein Überblick über mögliche Anwendungen oder größere Anwendungsbeispiele vorhanden wären.

Anwendungen wiederum können erst dann in größerem Maßstab erstellt werden, wenn – wie oben schon erwähnt – die grundlegenden Parameter bekannt sind und die konzeptionellen Hilfsmittel (Programmiersprachen, Modelle) und praktischen Hilfsmittel und Werkzeuge (hardware, Compiler, Entwicklungssysteme, Simulatoren, Analysatoren etc.) zur Verfügung stehen.

Die vorliegende Arbeit sollte einen Anstoß zur Überwindung dieser "Verklemmung" darstellen. Das realisierte CSSA-System bildet gleichermaßen ein portables, allgemein verfügbares und änderungsfreundliches **Experimentiersystem** für die CSSA-Sprache selbst, für die Anwendungsentwicklung und Algorithmenkonzeption mit CSSA-Sprachmitteln, und für mögliche hardware-Konfigurationen – wobei insgesamt Wert auf eine gute Benutzerunterstützung (Ereignisprotokollierung, Testhilfen) gelegt wurde. Es kann somit die Entwicklung, Erprobung, Untersuchung und Anwendung paralleler Algorithmen, adäquater Beschreibungssprachen und geeigneter Maschinenkonfigurationen "parallel" oder schrittweise iterativ durchgeführt werden.

Es besteht so die Hoffnung, daß die "beispiellose" und erfahrungsarme Diskussion über geeignete Rechnerstrukturen, Programmstrukturen und Sprachkonzepte für parallele Algorithmen damit angereichert werden kann, und das Hauptziel der CSSA-Entwicklung - einen wesentlichen Beitrag zu dieser Diskussion zu liefern und weitere Erkenntnisse zu gewinnen über eines der gegenwärtig wichtigsten Prinzipien und Probleme der Informatik, der **Organisation und Beherrschung der Parallelität** - schließlich erreicht werden kann.

Der fachmännische Theoretiker wählt ein Modell,
der Techniker nimmt die Daten und experimentiert.
Parallel dazu entwirft der Mathematiker, um Lösungen
und Antworten zu bekommen, einen Logarithmus.
Und schließlich kann die eigentliche Berechnung
von einem Informatiker durchgeführt werden.
Computer, das farbige Bildsachbuch zu Themen der Zeit,
rororo 7147, 1978

4. CSSA-Sprache

Dieses Kapitel ist der CSSA-Sprache gewidmet. Nach einem Überblick über die Sprachkonzepte (4.1) werden an einem längeren Beispiel (4.2) die Verwendung einiger Sprachkonstrukte und die Möglichkeiten des interaktiven CSSA-Systems erläutert. In Unterkapitel 4.3 werden die Kommunikationsmechanismen denen anderer Sprachen vergleichbarer Zielsetzung gegenübergestellt.

In jedem Unterkapitel mußte schon aus Zeit- und Platzgründen eine **Beschränkung auf einige wesentliche Aspekte** erfolgen: Die Sprachkonzepte können i.w. nur genannt werden, auf eine ausführliche Diskussion und Rechtfertigung einzelner Designentscheidungen mußte weitgehend verzichtet werden. Einige Aspekte werden allerdings in Kapitel 5 herausgegriffen; es sei diesbezüglich jedoch insbesondere auf Teil B dieser Arbeit mit der vollständigen Sprachbeschreibung und den Bericht [FRV81] hingewiesen. Auch im Beispiel 4.2 konnte nur auf einige wenige Konzepte der Sprache und des CSSA-Systems eingegangen werden. Schließlich würde ein fundierter Vergleich von CSSA mit anderen modernen Programmiersprachen, der über eine Schilderung und kurze Wertung der Kommunikationsmechanismen hinausgeht, eine eigenständige Arbeit darstellen - zweifellos von hohem Wert, wenn dabei auch auf Aspekte der Daten- und Kontrollabstraktion und der Modularisierung eingegangen werden kann, jedoch verbunden mit einem Aufwand, der hier nicht getrieben werden konnte.

4.1 Sprachkonzept

CSSA ist eine Programmiersprache für **asynchron parallele Prozesse**, die untereinander über **Nachrichten** kommunizieren. Beim Sprachentwurf wurde Wert darauf gelegt, die neueren Erkenntnisse und Ergebnisse des Programmiersprachen-Designs und des software-engineerings (Modularisierungs- und Abstraktionskonzepte) in homogener Weise mit Konzepten zu verbinden, die eine strukturierte Nutzung der Parallelität verteilter Systeme ermöglichen; dies geschah insbesondere auch unter Berücksichtigung der Verifizierbarkeit von CSSA-Programmen. Eine stichpunktartige Nennung der wesentlichen Sprachaspekte mag zunächst genügen:

- Die Sprachmechanismen für sequentielle und für nebenläufige Berechnungen sind weitgehend **orthogonal** zueinander konzipiert.
- Die **sequentiellen Sprachmechanismen** orientieren sich am heutigen Stand höherer algolartiger Programmiersprachen (PASCAL, Ada etc.).
- Die verteilten Berechnungen werden von **nebenläufig** arbeitenden, **autonomen**, abgeschlossenen sequentiellen Modulen (den **Agenten**) ausgeführt.
- Agentensysteme bilden eine **Heterarchie** gleichrangiger Berechnungseinheiten, nicht notwendigerweise eine Hierarchie.

- Die Zustände der einzelnen Agenten sind voneinander vollständig **entkoppelt**, es gibt keinen globalen Zustand und keine globale Umgebung oder gemeinsam benutzbare Speicherbereiche.
- Agenten kommunizieren untereinander nur durch **Nachrichten**; über die **Nachrichtenlaufzeit** wird keine Annahme gemacht, insbesondere können sich Nachrichten auch **überholen**. Ein Agent kann nur dann einem anderen Agenten eine Nachricht zusenden, wenn er dessen **Bekanntschaft** besitzt.
- Die Bekanntschaftsrelation zwischen Agenten ist reflexiv, asymmetrisch und nicht transitiv. Der Graph dieser Relation wird **Agentennetz** genannt. Da Agenten dynamisch gegründet werden, eigenverantwortlich terminieren und Bekannschaften selbst kommunizierbare Objekte sind, ist das Agentennetz **dynamisch** veränderbar.
- CSSA stellt eine in das System integrierte, **interaktive Benutzerschnittstelle** zur Verfügung: Der Benutzer ist über den **interface-Agenten** Teil des Agentennetzes.
- Der **Synchronisationsgrad** zwischen Agenten ist **programmierbar**. Es ist sowohl eine völlig **asynchrone Kommunikation**, als auch eine **handshaking-nahe Kommunikation** möglich. Durch die flexible Synchronisationsmöglichkeit ist ein hoher Grad an **Nebenläufigkeit** gewährleistet.

4.1.1 Höhere Sprachmechanismen

Berechnungen in CSSA werden wie oben beschrieben von **Agenten** ausgeführt, diese werden mittels eines Generatorausdrucks ('new') aus **Typdefinitionobjekten** erzeugt, die **scripts** genannt werden. Scripts sind also **Agentenschemata** und bilden wie Agenten (oder genauer: Verweise auf Agenten, sogen. **acquaintances**) einen eigenständigen **Datentyp**. Sowohl Verweise auf Agenten, als auch scripts als Moduldefinitionen, sind **kommunizierbare Objekte**.

Aus einem script können beliebig viele äquivalente Agenten erzeugt werden, ein script stellt die **Typdefinition** der aus ihm erzeugten Agenten dar und legt damit das Verhalten der Agenten fest. Da Agenten autonome Objekte sind, die sich keinen gemeinsamen Speicher teilen, bilden scripts **eigenständige Module**: Es sind keine außerhalb des scripts deklarierten Bezeichner sichtbar. Scripts sind die syntaktisch obersten Einheiten für den CSSA-Übersetzer, können aber auch lokal in anderen scripts definiert werden. Die aus **lokalen scripts** erzeugten Agenten stehen allerdings gleichwertig neben allen anderen Agenten im Agentennetz, es gibt in diesem Sinne keine lokal verkapselten Teilnetze.

Jeder Agent stellt eine Menge von **Operationen** (auch operation-capabilities genannt) zur Verfügung. Eine Operationsdefinition besteht aus einem **Operationsbezeichner**, einem **entry-pattern** und einem Rumpf, der eine Folge von Anweisungen enthält. Operationen definieren eine **unteilbare** Folge von sequentiellen Aktionen, die

aufgrund einer **Nachricht** ausgeführt werden. Operationen legen also fest, in welcher Weise ein Agent auf eine bestimmte Nachricht reagiert. Nachrichten (**messages**) bestehen aus einem Operationsbezeichner und einer linearen Liste von Werten verschickbarer Datentypen, diese Liste wird als **envelope** bezeichnet.

Kommunikation zwischen Agenten ist nur durch **Nachrichten-Übermittlung** möglich, über die **Dauer** einer Übertragung kann keine Annahme gemacht werden. Folglich können sich Nachrichten insbesondere auch **Überholen**. Jeder Agent sammelt die an ihn gerichteten Nachrichten in einem privaten Puffer, **mailbox** genannt. Die Eintragung von Nachrichten in die mailbox eines Empfängeragenten geschieht asynchron nebenläufig zum lokalen Ablauf von Sender und Empfänger, insbesondere können Nachrichten in die mailbox eingetragen werden, während der Empfänger gerade eine (unteilbare) Operation ausführt: Die Kommunikation impliziert keine Ablaufkopplung zwischen Sender und Empfänger während der Nachrichtenübertragung.

Wenn ein Agent nicht gerade mit der Verarbeitung einer Nachricht beschäftigt ist, also keine Operation ausführt, inspiziert er seine mailbox, um eine zur Verarbeitung geeignete Nachricht auszuwählen. Von der Auswahlstrategie wird lediglich **Fairness** gefordert.

Die Menge der zu einem Zeitpunkt zur Verfügung gestellten Operationen und die Semantik dieser Operationen bestimmen den **Zustand** eines Agenten. I.a. wird ein Agent stets nur eine **Teilmenge** aller Operationen zur Verfügung stellen (sogen. **aufrufbare Operationen**). Diese Teilmenge wird in **Facetten** spezifiziert. In einem CSSA-script können mehrere Facetten definiert sein, ein Agent befindet sich stets in genau einer Facette, der sogen. **aktuellen Facette**.

Während der Ausführung einer Operation kann durch bestimmte Anweisungen eine **Facettierung** des Agenten bewirkt werden: Eine andere Facetteninstanz wird zur neuen aktuellen Facette. Dabei kann eine **neue** Facetteninstanz **geschaffen** werden oder in eine bereits existierende Instanz **zurückgekehrt** werden. Auch **rekursives** Facettieren ist möglich. Ein Agent kann also zu verschiedenen Zeitpunkten verschiedene Operationen zur Verfügung stellen (oder gleichbenannte mit unterschiedlicher Wirkung).

Der sogenannte **Ablaufzyklus** eines Agenten wird solange ausgeführt, bis der Agent eigenständig terminiert. Dieser Zyklus sieht so aus: Nachdem eine Nachricht für eine (in der gegenwärtigen Facette aufrufbare !) Operation aus der mailbox ausgewählt wurde, wird überprüft, ob die Nachricht das entry-pattern der benannten Operationsdefinition matcht und ob eine sogenannte **entry-assertion** erfüllt ist. Falls dies der Fall ist, wird die zugehörige Operation mit den durch das pattern spezifizierten Variablenbindungen ausgeführt und die Nachricht aus der mailbox entfernt. Nach Beendigung der Operation (ein Terminieren kann natürlich nicht garantiert werden !) oder im Falle eines Mißlingens des matches wird die mailbox weiter inspiziert. Falls keine passende Nachricht gefunden wird, so ist der Agent blockiert, oder aber es wird eine sogenannte **idle-Operation** ausgeführt, sofern eine solche in der aktuellen Facette definiert ist. Entry-

patterns von Operationsdefinitionen stellen also sowohl **Bindungsplätze für übermittelte Werte**, als auch (zusammen mit dem Operationsbezeichner und der assertion) **Selektionskriterien für Nachrichten** dar.

Neben der Durchführung rein lokaler, sequentieller Berechnungen und der Gründung neuer Agenten kann während der Ausführung einer Operation facettiert werden (das impliziert die Beendigung der Operation) und mit anderen Agenten kommuniziert werden. Die **Kommunikation** kann auf zwei Arten geschehen:

Der erste Mechanismus ermöglicht völlig **asynchrone Kommunikation** ohne implizite Kopplung zwischen Sender und Empfänger: Es handelt sich um den **einfachen send-Befehl**, durch den eine Nachricht in Form eines Operationsaufrufes mit einem envelope "aktueller Parameter" versandt wird. Sender und Empfänger werden nicht verzögert, insbesondere führt der Sender sofort das nächste Statement aus, während die Nachricht noch unterwegs ist.

Der zweite Mechanismus erlaubt eine programmierte Kopplung zwischen Sender und Empfänger, es handelt sich um die **Kommunikation durch Auftragsbeziehung** (send/receive-Mechanismus). Hierbei wird dem Auftragnehmer bei einem Operationsaufruf implizit eine **Antwortverpflichtung** übergeben. Der Empfänger kann überprüfen, ob eine Antwortverpflichtung vorliegt und dieser Verpflichtung durch 'reply' nachkommen (ohne eine explizite Bekanntschaft auf den Auftraggeber zu besitzen !) oder sie an einen anderen Agenten weitervererben, der nun seinerseits der Antwortverpflichtung nachkommen muß. Der Auftraggeber muß die **Antwortnachricht** noch in der gleichen Operation empfangen. Hierzu stehen ihm zwei Konstrukte zur Verfügung: Das **wait-Statement** verzögert die Ausführung der Operation so lange, bis eine Antwortnachricht für den bei der Vergabe des Auftrages angegebenen **port** empfangen wird; mit dem unsynchronisierten **receive-Konstrukt** ist ein selektives und aktives Warten flexibel programmierbar.

Kommunizierbare Größen sind Werte der Datentypen INT, REAL, BOOL, STRING; aber auch Verweise auf Agenten (sogen. acquaintances), also Werte vom Typ AGENT, scripts (nämlich Werte des Datentyps SCRIPT) und Operationsbezeichner und -litterale (als Werte vom Datentyp OPER) sind kommunizierbare Objekte. Auch arrays und records von diesen Objekten sind als ganzes verschickbar, sie werden dabei automatisch linearisiert. Aufzählungstypen und sets sind nicht direkt verschickbar.

4.1.2 Sequentielle Sprachmechanismen

CSSA enthält auf lokaler Ebene i.w. die üblichen Daten- und Kontrollstrukturen.

Die **einfachen Datentypen** INT, REAL, BOOL und STRING mit den üblichen Operatoren und Möglichkeiten Ausdrücke zu bilden, dienen zur Repräsentation von ganzen Zahlen, rationalen Zahlen, Wahrheitswerten und Zeichenketten. Zur Zeichenkettenmanipulation werden eine Reihe von **built-in Funktionen** zur Verfügung gestellt, ebenso zur Datenkonvertierung. Die einfachen Datentypen

AGENT, SCRIPT und OPER sind CSSA-spezifisch; sie existieren, weil Agenten, scripts und Operationen manipulierbare und verschickbare Objekte sind. Es besteht ebenfalls die Möglichkeit, **Aufzählungstypen** zu definieren.

Von den oben genannten acht Datentypen können **arrays**, **records** und **sets** als komplexe Datentypen gebildet werden, sie werden wie auch die Aufzählungstypen üblicherweise in einer **Typdefinition** spezifiziert, bevor Variablen, Konstanten und Funktionen von ihnen deklariert werden. Anonyme Typdefinitionen sind möglich, bezüglich Typgleichheit beachte man jedoch, daß CSSA ein **strenges Typkonzept** hat.

Eine Besonderheit stellen die **Relationen** dar: Bei ihnen handelt es sich um **dynamische**, sehr komfortable Datenstrukturen, die das Prinzip des **inhaltsadressierten Assoziativspeichers** bereits auf Sprachebene zur Verfügung stellen. Eine Relation ist eine Menge von records, die durch Schlüsselfelder identifiziert werden. Mit ihnen kann auf einer hohen Abstraktionsstufe die Verwaltung von Information bearbeitet werden, wofür in anderen Programmiersprachen i.a. 'dynamische Listen und andere pointer-Strukturen erst programmiert werden müssen. Gleichzeitig können damit beliebige dynamische pointer-Strukturen innerhalb eines abgeschlossenen Adressraumes simuliert werden.

Die **lokalen Kontrollstrukturen** entsprechen denen bekannter algolartiger Programmiersprachen. Im wesentlichen sind hier zur Steuerung des Kontrollflusses auf Modulebene das **if-Statement**, das **select-Statement** (zur Auswahl zwischen mehreren Alternativen) und das **loop-Statement** zu nennen. Alle kontrollstrukturartigen Statements bilden lokale scopes, in denen Deklarationen stehen können, eine begin/end-Klammerung ist daher i.a. nicht notwendig.

Das **Schleifenkonzept** ist sehr mächtig, die verschiedenen Schleifentypen (while-Schleife, Zählschleife, ereignisgesteuerte Schleife etc.) können in uniformer und übersichtlicher Weise beliebig kombiniert und verschachtelt werden. Kontrollierte Abbruchbedingungen (escape/continue) und exit-Blöcke ersetzen einen in CSSA nicht vorhandenen goto-Befehl. Zum Durchlaufen von sets und Relationen werden ebenfalls Möglichkeiten bereitgestellt.

Prozeduren und Funktionen können definiert werden. Die Parameterübergabe erfolgt "by value" für in-Parameter, bzw. "by result" für in- und out-Parameter. Wechselseitige Rekursion wird durch eine Vorwärtsdeklaration ermöglicht. Funktionen dürfen im Gegensatz zu Prozeduren keinerlei Seiteneffekt haben und werden bei der Auswertung von Ausdrücken implizit aufgerufen; Prozeduren werden durch 'call' explizit aufgerufen.

In CSSA gilt das übliche **Blockkonzept**; um die one-pass-Compilierbarkeit zu ermöglichen ist allerdings auf ein strenges **"declare before use"-Konzept** geachtet worden. Zu beachten ist auch, daß wie oben begründet (lokale) scripts Sichtbarkeitsschranken bilden.

4.1.3 Benutzerintegration

Das CSSA-Konzept sieht eine integrierte, interaktive **Benutzerschnittstelle** vor, den **interface-Agenten**. Der interface-Agent ist ein initial im System existierender Agent, der allen Agenten standardmäßig bekannt ist. Im Unterschied zu diesen verfügt er nicht über ein fest definiertes script, statt dessen ist er dynamisch programmierbar: Die interaktiv eingegebenen Instruktionen werden interpretativ ausgeführt.

Durch den interface-Agenten ist der Benutzer über ein Bildschirmterminal Teil des Agentennetzes, er kann Agenten erzeugen, lokale Größen deklarieren und Zuweisungen an lokale Größen vornehmen; vor allem aber kann er mit den anderen Agenten des Netzes mit den üblichen CSSA-Sprachmechanismen (send/receive) kommunizieren: Die **Steuerung und Kontrolle des Systems** erfolgt weitgehend auf Sprachebene. Für spezielle Eingriffe in die ablaufende Berechnung sind allerdings zusätzliche privilegierte Anweisungen vorgesehen.

Der Benutzer hat vom interface aus Zugriff auf gewisse scripts und Agenten. Er kann aus scripts, die er mit dem Editor interaktiv programmiert (oder – was in der vorliegenden Implementierung allerdings nicht vorgesehen ist – von einer Scriptbibliothek dynamisch hinzulädt) Agenten gründen. Zugreifbaren Agenten kann er acquaintances auf andere Agenten (oder sich selbst) bei der initialen creation-message oder bei einem operation-request übergeben. Schließlich kann er (bspw. durch einen geeigneten send-Befehl) eine verteilte Berechnung starten und sodann ihren Ablauf überwachen und steuern.

Da Agenten dynamisch gegründet werden und sowohl scripts als auch Bekanntschaften kommunizierbare Objekte sind, ist das **Agentennetz** dynamisch veränderbar.

Für Verwaltungsaufgaben und häufig wiederkehrende ähnliche Berechnungen wird sich der Benutzer eigene Agenten definieren (**Spezialisten** oder **Server**), die dedizierte Aufgaben übernehmen können. Auch ganze spezialisierte Teilnetze können so als verteilte Expertensysteme fungieren.

Da der Benutzer durch den interface-Agenten i.w. gleichrangig neben allen anderen Agenten des Netzes steht, bildet – bedingt durch die unvollständige und inkonsistente Systemsicht – die **Ablaufüberwachung** der parallelen Berechnung ein gewisses Problem. Hier helfen im konzipierten CSSA-Simulationssystem debugging- und tracing-Konzepte und andere Methoden zur Ablaufsteuerung und zum Eingriff in laufende Berechnungen, die über die eigentliche CSSA-Sprache hinausgehen, jedoch sinnvollerweise Bestandteil von Test- und Entwicklungssystemen sein müssen.

4.2 Ein Beispiel: Euklidischer Algorithmus

Im folgenden soll exemplarisch der Weg von einem bekannten **Problem** zu seiner **Lösung** durch ein **CSSA-Programm** aufgezeigt werden. Dabei werden einige der verwendeten CSSA-Sprachkonstrukte und einige interface-Kommandos erläutert, so daß der Leser einen Eindruck von der CSSA-Sprache und den Möglichkeiten des konzipierten CSSA-Systems erhält. Selbstverständlich kann bei einem kurzen Beispiel wie diesem nur in unvollständiger Form auf wenige Aspekte eingegangen werden, es sei daher hier nochmals ausdrücklich auf die vollständige Sprachbeschreibung (Teil B), das Handbuch zur Systembenutzung (Teil C) und die Beispielsammlung (Teil D) hingewiesen. Das Beispielprogramm wird in Kapitel 7 und 8 wieder aufgegriffen.

Als Beispiel wurde das Problem der Bestimmung des **größten gemeinsamen Teilers** von n Zahlen gewählt; der zugrundeliegende **Euklidische Algorithmus** stellt dabei nicht nur einen oft zitierten "Uralgorithmus" von einer gewissen Bedeutung dar, er ist trotz seiner Kürze nicht unmittelbar einsichtig oder trivial (Komplexitätsanalyse!) und liefert in seinen unterschiedlichen Varianten und Erweiterungen viel Spielmaterial als Gegenstand informatischer Untersuchungen (wie bspw. Verifikation, automatische Transformation rekursiver Programme etc.). Besonders reizvoll scheint, daß der bekannte Algorithmus hier in einem neuen Gewande erscheint; es zeigt sich, daß eine verteilte Berechnung des ggT ein interessantes Maß an Nichtdeterminismus beinhaltet und sich auch zur Illustration typischer Probleme bei Berechnungen durch kooperierende Prozesse (Terminierungsproblematik, Verifikation) hervorragend eignet.

Die **erste Idee** der **verteilten Berechnung** des ggT von n natürlichen Zahlen $X_1, \dots, X_n$ ($n \geq 1$, $X_i > 0$) ist recht einfach und geht auf Francez zurück. Danach werden n parallele Prozesse **ringförmig** so angeordnet, daß jeder Prozeß mit seinem unmittelbaren linken und rechten Nachbarn kommunizieren kann. Nach anfänglicher Initialisierung der Prozesse $P_1, \dots, P_n$ mit den Zahlenwerten $X_1, \dots, X_n$ führt jeder Prozeß einen **lokalen Arbeitszyklus** durch, der darin besteht, seinen beiden Nachbarn die eigene Zahl mitzuteilen und anschließend auf eine entsprechende Nachricht von einem der Nachbarprozesse zu warten. Falls der empfangene Wert kleiner als der eigene Wert ist, wird der eigene Wert entsprechend der Euklidischen Regel (Restbildung bei der Division der beiden Werte) verändert und der neue Wert den Nachbarn mitgeteilt, andernfalls geschieht nichts.

Diese Lösung, bei der (insbes. bei großem n) Berechnungen parallel ablaufen können, hat die Eigenschaft, daß schließlich jeder Prozeß den Wert des ggT annimmt und die verteilte Berechnung in diesem Zustand **blockiert**: Es sind dann keine akzeptierbaren Nachrichten mehr unterwegs. Allerdings "weiß" kein Prozeß, ob die Berechnung global blockiert ist! Das nicht triviale Problem der sogen. **verteilten Terminierung**, das nicht so sehr in der (individuellen) Erkennung des globalen Terminierungszustandes als in der (allgemeinen) Frage nach einer automatischen Transformation derartiger blockierender Programme in äquivalente terminierende Programme besteht, soll hier jedoch nicht weiter

diskutiert werden. Es sei jedoch noch darauf hingewiesen, daß bei derartigen Überlegungen für die hier angegebenen CSSA-Lösungen selbstverständlich die **Kommunikationssemantik** von CSSA (insbesondere auch die unbestimmte Nachrichtenlaufzeit) beachtet werden muß.

Bei einer **CSSA-Lösung der ggT-Berechnung** wird man jeden Prozeß durch einen Agenten gleichen Typs realisieren. In einer **Initialisierungsphase** müssen nicht nur die n Agenten gegründet und mit der entsprechenden Zahl initialisiert werden, sondern es muß auch das ringförmige Agentennetz aufgebaut werden, indem jeweils benachbarte Agenten miteinander bekannt gemacht werden. In einer **zweiten Phase** kann ein Agent sodann seinen Zahlwert den beiden Nachbaragenten mitteilen, um anschließend in einer **dritten Phase** auf eine **akzeptierbare Nachricht** (das ist eine Nachricht mit einem Wert, der kleiner dem eigenen ist) von einem Nachbarn zu warten. Falls eine derartige Nachricht empfangen wird, kann der Agent nach Durchführung der Restberechnung wieder mit Phase zwei fortfahren.

Es bietet sich unmittelbar an, die drei Phasen durch drei **parallele Facetten** INIT, DISPLAY und ACCEPT zu modellieren, wobei die Phasenübergänge durch einen Facettenwechsel realisiert werden und die Facette ACCEPT eine Operation EUKLID enthält, durch die akzeptierbare Nachrichten verarbeitet werden. Die DISPLAY-Facette benötigt keine von außen aufrufbare Operation, hier sollen "spontan" Nachrichten an die Nachbaragenten versandt werden. In der INIT-Facette muß eine Operation KNOW vorhanden sein, die Bekanntschaftsnachrichten entgegennimmt und verarbeitet und anschließend den Übergang in den DISPLAY-Zustand veranlaßt.

Es folgt die script-Definition von Agenten des Typs PROCESS:

```

1      (* ggT-Berechnung *)
2  type PROCESS is
3  script ( int: N) assert N > 0
4      var int: NUMBER := N ;
5      var agent: LEFT,RIGHT ;
6      facethead ACCEPT ;
7
8      facet DISPLAY is
9          operation idle is
10             oper: EUKLID ;
11
12             print ("*** MY CURRENT NUMBER : ",NUMBER," ***") ;
13             send EUKLID(NUMBER) to LEFT ;
14             send EUKLID(NUMBER) to RIGHT;
15             replace by ACCEPT ;
16         endoperation
17     endfacet
18
19     facet INIT is
20         public: KNOW ;
21         operation KNOW(--> LEFT,RIGHT) is
22             replace by DISPLAY;
23         endoperation
24     endfacet
25
26     facet ACCEPT is
27         public: EUKLID;
28         operation EUKLID( int: NEIGHBOUR)
29             assert NEIGHBOUR<NUMBER is
30             functionhead MOD( int: I,J) returns int external;
31             NUMBER := MOD(NUMBER-1,NEIGHBOUR)+1 ;
32             replace by DISPLAY;
33         endoperation
34     endfacet
35
36     initial INIT ;
37 endscript

```

Einige zusätzliche **Erläuterungen** mögen zum genaueren Verständnis des CSSA-Programms beitragen: Die Variable NUMBER stellt die im Verlauf der Berechnung sich ändernde Größe dar, die anfangs (Zeile 4) mit dem Initialwert N belegt wird und schließlich den Wert des ggT annimmt. Der bei der Gründung des Agenten übermittelte initiale Wert N ist entsprechend der Eingabezusicherung des Algorithmus garantiert größer als 0, ansonsten ist der Agent wegen der assertion $N > 0$ (Zeile 3) nicht lebensfähig. Die initiale Facette wird am Ende des scripts nach 'initial' (Zeile 36) benannt. Die Variablen LEFT und RIGHT vom Typ AGENT stellen Bekanntschaften (acquaintances) auf die beiden benachbarten Agenten dar, sie erhalten ihren Wert erst bei Ausführung der KNOW-Operation der INIT-Facette. Diese Operation wird von außen (bspw. vom interface aus) durch eine Nachricht mit geeigneten Parametern aktiviert; durch den Pfeil im Pattern der Operationsdefinition wird spezifiziert, daß es sich bei LEFT und RIGHT um global definierte Größen handelt, die bei der Aktivierung der Operation durch einen patternmatch-Vorgang einen Wert erhalten.

Durch das einzige Statement 'replace by DISPLAY' der KNOW-Operation wird die initiale Facette verlassen und die DISPLAY-Facette betreten. Die hier definierte idle-Operation ($\rightarrow$ B 19.3) wird sofort ausgeführt, da in diesem Zustand keine andere ausführbare Operation existiert. Nachdem der gegenwärtige Wert der Variablen NUMBER durch PRINT ausgegeben wurde und an die linken und rechten Nachbaragenten in Form des Operationsaufrufes EUKLID versandt wurde, wird in die ACCEPT-Facette übergewechselt. Erst jetzt können zwischenzeitlich evtl. schon an den Agenten gerichtete EUKLID-Nachrichten durch die einzige Operation dieser Facette akzeptiert werden. Durch die assertion $NEIGHBOUR < NUMBER$ wird sichergestellt, daß nur solche Nachrichten akzeptiert werden, durch die ein kleinerer Wert als der eigene Wert von einem Nachbaragenten übermittelt wird. Die Neubesetzung der Variablen NUMBER wird entsprechend der Kernidee des Euklidischen Algorithmus durch die Restberechnung, hier mit der built-in Funktion MOD, vorgenommen. Man beachte, daß $MOD(NUMBER-1, NEIGHBOUR)+1$ fast immer äquivalent zum Ausdruck $MOD(NUMBER, NEIGHBOUR)$ ist, bis auf den Fall, daß letzterer 0 ergibt, wenn die Division aufgeht. Dann liefert der erste Ausdruck den Wert von NEIGHBOUR. Nachdem der Wert von NUMBER verkleinert (!) wurde, wird wieder in die DISPLAY-Facette übergewechselt. Eine evtl. zwischenzeitlich eingegangene weitere EUKLID-Nachricht (etwa vom anderen Nachbarn) kann erst zu einem späteren Zeitpunkt, wenn der Agent sich wieder in der ACCEPT-Facette befindet, empfangen werden.

Es folgen einige **Anmerkungen zu technischen Details** des Programms: FACETHEAD (Zeile 6) stellt eine Vorwärtsvereinbarung der Facette ACCEPT dar. Sie ist aufgrund des die one-pass Übersetzbarkeit garantierenden "declare before use Prinzips" der statischen Semantik von CSSA notwendig, da sich die beiden Facetten DISPLAY und ACCEPT wechselseitig aufrufen. Von außen aufrufbare Operationen einer Facette müssen zu Beginn der Facette hinter dem Schlüsselwort PUBLIC deklariert werden (Zeilen 20,27). Der in den Zeilen 13 und 14 verwendete Operationsbezeichner EUKLID stellt hier ein Literal (und nicht etwa eine Variable oder Konstante vom Typ OPER) dar, Operationsliterals werden in einer $<OPER-DECL>$ (Zeile 10) vereinbart. Der waagerechte Strich (Zeilen 11 und 30) trennt den Deklarationsteil und den Ausführungsteil von Blöcken; die Operation KNOW enthält keinen Deklarationsteil, daher kann hier der Trennstrich entfallen. Bei MOD handelt es sich um eine built-in Funktion, der Kopf solcher Funktionen wird wie bei externen Funktionen deklariert; die Namen der formalen Parameter I und J sind dabei bedeutungslos.

Nachdem das script mit dem CSSA-Compiler übersetzt wurde, kann in der **interaktiven CSSA-Sitzung** das Agentennetz aufgebaut werden. Dem interface-Agenten, über den der Benutzer Zugang zum CSSA-System hat, ist das script PROCESS bekannt. Um bspw. den ggT von fünf Zahlen zu berechnen, werden nun fünf Variablen vom Typ AGENT deklariert und die Agenten gegründet:

```
>>> 0.000 INTERFACE(1) : ENTER CSSA COMMAND -
      VAR AGENT:A1,A2,A3,A4,A5;
      A1 :- NEW PROCESS(108) ;
      A2 :- NEW PROCESS(76) ;
      A3 :- NEW PROCESS(12) ;
```

```
A4 :- NEW PROCESS(60) ;
A5 :- NEW PROCESS(36) ;
```

Diese Kommandos werden **interpretativ** vom interface-Agenten ausgeführt; sie unterscheiden sich aber nicht von den entsprechenden CSSA-Kommandos, die in scripts enthalten sind und vom Compiler **übersetzt** werden. Durch den Generator **NEW** wird ein Agent gegründet, der Typ des Agenten (hier PROCESS) wird dabei spezifiziert. Gleichzeitig werden Parameter angegeben, deren Werte mit der Gründungsnachricht versandt werden und durch einen sogen. creation-match an die im Pattern des Agenten angegebenen Variablen (hier N) gebunden werden (--> B 17.3.3,20.3). Die Variablen A1,...,A5 des interface haben jetzt als Wert jeweils einen Verweis auf den entsprechenden Agenten. Damit die Agenten sich entsprechend der Doppelringkonfiguration kennenlernen, muß die KNOW-Operation aufgerufen werden; der Benutzer wird daher im interface die folgenden Befehle ausführen:

```
>>> 0.000 INTERFACE(1) : ENTER CSSA COMMAND -
      OPER: KNOW ;
      SEND KNOW(A5,A2) TO A1 ;
      SEND KNOW(A1,A3) TO A2 ;
      SEND KNOW(A2,A4) TO A3 ;
      SEND KNOW(A3,A5) TO A4 ;
      SEND KNOW(A4,A1) TO A5 ;
```

Wie schon beim script PROCESS erläutert, muß auch im interface zunächst das Literal des aufgerufenen Operationsnamens (hier KNOW) vereinbart werden. Der Interpreter gibt bei Meldungen, die an den Benutzer gerichtet sind, jeweils den aktuellen **Simulationszeitpunkt** an, hier 0.000. Da Nachrichten im Simulationssystem eine gewisse Laufzeit besitzen (standardmäßig mindestens 0.1 Zeiteinheiten, wenn die Agenten - wie hier vorausgesetzt wird - auf unterschiedlichen Prozessoren laufen), ist das Agentennetz zu diesem Zeitpunkt noch nicht aufgebaut. Tatsächlich wird immer dann, wenn der Benutzer am interface die Kontrolle hat, die Simulationszeit angehalten. Der Benutzer kann nun durch Eingabe des Kommandos **RUN** den Ablauf der Berechnung starten. Er erhält folgende Meldungen (zur eindeutigen Identifizierung werden im Protokoll die PROCESS-Agenten mit ihrer Inkarnationsnummer gekennzeichnet):

```
>>> 0.000 INTERFACE(1) : ENTER CSSA COMMAND -
      RUN;
*** 0.300 PROCESS(1) : *** MY CURRENT NUMBER : 108 ***
*** 0.300 PROCESS(2) : *** MY CURRENT NUMBER : 76 ***
*** 0.700 PROCESS(4) : *** MY CURRENT NUMBER : 60 ***
*** 0.700 PROCESS(3) : *** MY CURRENT NUMBER : 12 ***
*** 0.700 PROCESS(5) : *** MY CURRENT NUMBER : 36 ***
*** 1.500 PROCESS(4) : *** MY CURRENT NUMBER : 12 ***
*** 1.500 PROCESS(2) : *** MY CURRENT NUMBER : 4 ***
*** 1.600 PROCESS(1) : *** MY CURRENT NUMBER : 32 ***
*** 2.500 PROCESS(1) : *** MY CURRENT NUMBER : 4 ***
*** 2.500 PROCESS(3) : *** MY CURRENT NUMBER : 4 ***
*** 2.600 PROCESS(5) : *** MY CURRENT NUMBER : 12 ***
*** 3.200 PROCESS(4) : *** MY CURRENT NUMBER : 4 ***
*** 3.500 PROCESS(5) : *** MY CURRENT NUMBER : 4 ***
>>> 4.400 SYSTEM TERMINATED
```

Zum Zeitpunkt 4.4 ist die **Berechnung blockiert**: Das Simulationssystem erkennt den Blockierungszustand und erzeugt eine entsprechende Meldung (--> Kap. 8.3). Den Meldungen, die durch das PRINT-Kommando bewirkt werden, entnimmt man, daß zum Blockierungszeitpunkt alle Agenten die gleiche Belegung ihrer Variablen NUMBER besitzen, nämlich 4. Tatsächlich gilt $ggT(108,76,12,60,36) = 4$.

Die bisher geschilderte Lösung ist noch **nicht in jeder Hinsicht befriedigend**. So stellt etwa das PRINT-Kommando (--> B 16) keinen vollwertigen E/A-Befehl dar, sondern eher einen Befehl, der nur für Testzwecke im Simulationssystem geeignet ist, da durch ihn Agenten Werte direkt auf den Bildschirm des interface ausgeben können, unter Umgehung der eigentlichen Kommunikationskanäle. Eine **Lösung ohne PRINT** ist daher vorzuziehen; allerdings genügt es nicht, Möglichkeiten zu schaffen, die ein Inspizieren der NUMBER-Variablen durch das interface zulassen, sondern es muß auch der **Blockierungszustand erkannt** werden!

Mehr **Einblick in die ablaufende Berechnung** ist wünschenswert, vor allem um die Effizienz des Verfahrens beurteilen zu können. Auch ist die **"Bedienung" des Programms** nicht ganz einfach: Für jede erneute Berechnung müssen die Agenten neu gegründet werden und das Netz neu aufgebaut werden - was mit einigem Schreibaufwand für den Benutzer verbunden ist. Schließlich sind einige **Varianten** und Erweiterungen des Algorithmus denkbar; vor allem aber möchte man sich von der **Korrektheit** des Verfahrens überzeugen und Experimente mit **unterschiedlichen hardware-Konfigurationen** durchführen. Auf diese Punkte soll im folgenden eingegangen werden.

Die nachstehende **Zustandstabelle** gibt die Belegung zu unterschiedlichen Zeitpunkten an und zeigt, wie die Agenten sich bei den Berechnungen gegenseitig beeinflußt haben; die hierin enthaltene Information läßt sich allein aus den oben angegebenen PRINT-Meldungen ableiten:

PROC (1)	108	=====>32	=>4=====						T
		"	"	"	"	"	"	"	E
		"	"	"	"	"	"	"	R
PROC (2)	76=====	=>4=====	"	"	"	"	"	"	M
		"	"	"	"	"	"	"	I
PROC (3)		12==	=>4=====	"	"	"	"	"	N
		"	"	"	"	"	"	"	A
PROC (4)		60	=>12=====	"	"	"	"	"	T
		"	"	"	"	"	"	"	E
PROC (5)		36		"	"	"	"	"	D
		"	"	"	"	"	"	"	
time	0.3	0.7	1.5	1.6	2.5	2.6	3.2	3.5	4.4

Die eingezeichneten Pfeile geben an, welcher Wert eines Nachbaragenten den jeweiligen Wert hervorgerufen hat.

Eine Untersuchung des Agenten mit der Inkarnationsnummer 5, also PROCESS(5), zeigt, daß er insgesamt 6 Nachrichten erhalten haben muß (seine Nachbarn PROCESS(1) und PROCESS(4) haben je 3 mal die DISPLAY-Facette betreten), jedoch nur einmal von seinem linken und rechten Nachbarn "beeinflusst" wurde. Daher müssen die Nachrichten EUKLID(108) und EUKLID(32) von seinem rechten Nachbarn sowie EUKLID(60) und EUKLID(4) von seinem linken Nachbarn zum Zeitpunkt 4.4 sich noch in der mailbox von PROCESS(5) befinden. Ähnliche Überlegungen ergeben, daß auch PROCESS(2) und PROCESS(3) je 4 Nachrichten und PROCESS(1) und PROCESS(4) je 3 Nachrichten nicht akzeptierten. Tatsächlich zeigt auch die folgende Tabelle, die durch das **interpreter-Kommando STATUS** erzeugt wird, diesen Sachverhalt:

```
>>> 4.400 INTERFACE(1) : ENTER CSSA COMMAND -
      STATUS;
+++ 4.400 ALL EXISTING AGENTS:
```

AGENT	FACET	OPER.	MAILBOX
P1: INTERFACE(1) *			
P2: PROCESS(1)	ACCEPT		EUKLID EUKLID EUKLID
P3: PROCESS(2)	ACCEPT		EUKLID EUKLID EUKLID EUKLID
P4: PROCESS(3)	ACCEPT		EUKLID EUKLID EUKLID EUKLID
P5: PROCESS(4)	ACCEPT		EUKLID EUKLID EUKLID
P6: PROCESS(5)	ACCEPT		EUKLID EUKLID EUKLID EUKLID

Aus der Tabelle geht auch hervor, daß keiner der Agenten gegenwärtig eine Operation ausführt, und sie sich alle in der Facette ACCEPT befinden. Die Namen P1...P6 benennen die simulierten hardware-Prozessoren, auf denen die Agenten laufen.

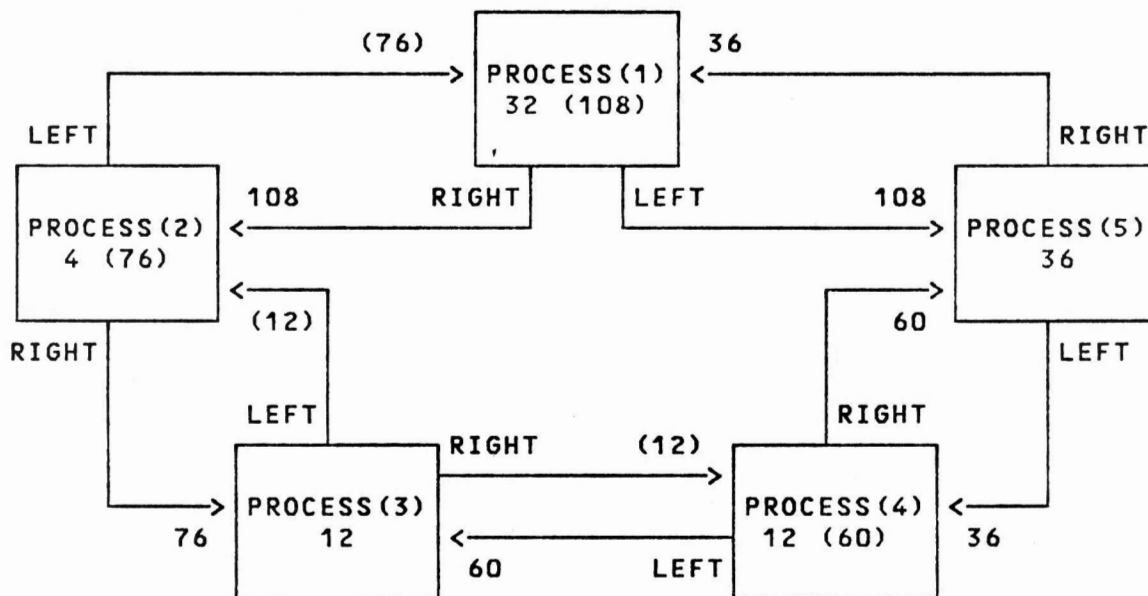
Um einen Einblick in die ablaufenden Berechnungen zu bekommen, können verschiedene **Protokollierungsmöglichkeiten** des CSSA-Systems benutzt werden (--> Kap. 7.3). Lediglich eine davon, die **observe-Option** (--> C 5.4), soll hier verwendet werden. Sie bewirkt, daß die wichtigsten Aktionen aller Agenten gemeldet werden. Dazu gehören das Versenden und Empfangen von Nachrichten, das Facettieren, das Gründen neuer Agenten, das Starten von Operationen und das Warten auf eine akzeptierbare Nachricht. Die observe-Option wird durch das interface-Kommando OBSERVE eingeschaltet. Von den erzeugten Meldungen werden hier der Übersichtlichkeit wegen nur diejenigen abgedruckt, die sich auf PROCESS(3) beziehen:

```
+++ 0.500 PROCESS(3) RECEIVES KNOW(AGENT:PROCESS(2),AGENT:
      PROCESS(4)) FROM INTERFACE(1)
+++ 0.500 PROCESS(3) STARTING OPERATION KNOW(AGENT:
      PROCESS(2),AGENT:PROCESS(4))
+++ 0.600 PROCESS(3) PERFORMS FACETTING : INIT --> DISPLAY
+++ 0.600 PROCESS(3) STARTING OPERATION IDLE
*** 0.700 PROCESS(3) : *** MY CURRENT NUMBER : 12 ***
+++ 0.800 PROCESS(3) SENDS EUKLID(INT:12) TO PROCESS(2)
+++ 0.900 PROCESS(3) SENDS EUKLID(INT:12) TO PROCESS(4)
+++ 1.000 PROCESS(3) PERFORMS FACETTING : DISPLAY --> ACCEPT
+++ 1.000 PROCESS(3) IS IDLE
+++ 1.000 PROCESS(3) RECEIVES EUKLID(INT:60) FROM PROCESS(4)
+++ 1.000 PROCESS(3) IS IDLE
+++ 1.300 PROCESS(3) RECEIVES EUKLID(INT:76) FROM PROCESS(2)
+++ 1.300 PROCESS(3) IS IDLE
+++ 1.900 PROCESS(3) RECEIVES EUKLID(INT:12) FROM PROCESS(4)
+++ 1.900 PROCESS(3) IS IDLE
+++ 2.200 PROCESS(3) RECEIVES EUKLID(INT:4) FROM PROCESS(2)
+++ 2.200 PROCESS(3) STARTING OPERATION EUKLID(INT:4)
+++ 2.400 PROCESS(3) PERFORMS FACETTING : ACCEPT --> DISPLAY
+++ 2.400 PROCESS(3) STARTING OPERATION IDLE
*** 2.500 PROCESS(3) : *** MY CURRENT NUMBER : 4 ***
+++ 2.600 PROCESS(3) SENDS EUKLID(INT:4) TO PROCESS(2)
+++ 2.700 PROCESS(3) SENDS EUKLID(INT:4) TO PROCESS(4)
+++ 2.800 PROCESS(3) PERFORMS FACETTING : DISPLAY --> ACCEPT
+++ 2.800 PROCESS(3) IS IDLE
+++ 3.500 PROCESS(3) RECEIVES EUKLID(INT:4) FROM PROCESS(4)
+++ 3.500 PROCESS(3) IS IDLE
```

Das Protokoll zeigt, wie der Agent die KNOW-Nachricht (mit den Werten der übergebenen Parameter) empfängt und ausführt (Zeitpunkt 0.5), in die DISPLAY-Facette überwechselt, dort die idle-Operation startet (Zeitpunkt 0.6) und den Wert seiner Zahl (12) ausgibt und an seine beiden Nachbarn verschickt (0.8, 0.9). Kurz nachdem er in die ACCEPT-Facette übergewechselt ist (Zeitpunkt 1.0), empfängt er eine EUKLID-Nachricht seines rechten Nachbarn mit dem Zahlwert 60, die er jedoch nicht akzeptiert, ebenso wenig wie die Nachricht seines linken Nachbarn (Zeitpunkt 1.3). Zwischen den Zeitpunkten 1.3 und 2.3 wartet der Agent auf eine akzeptierbare Nachricht, die zum Zeitpunkt 1.9 eintreffende

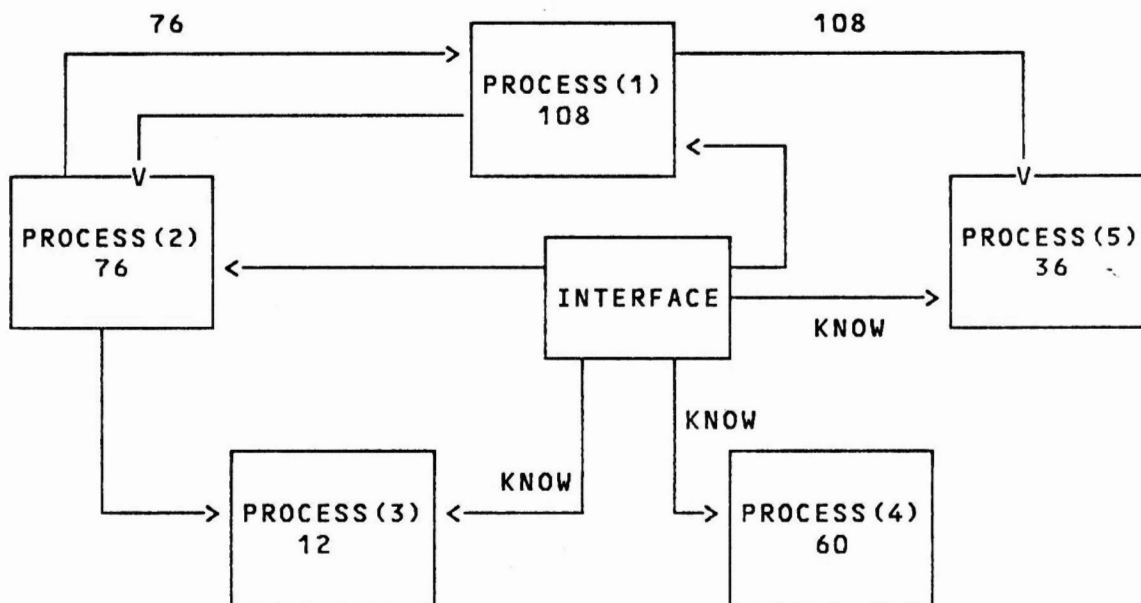
Nachricht mit dem Wert 12 von PROCESS(4) wird nicht akzeptiert. Erst die zum Zeitpunkt 2.2 empfangene Nachricht löst einen weiteren Zyklus aus, der zum Zeitpunkt 2.8 beendet ist und dazu führt, daß die Variable NUMBER den Wert 4 annimmt, der nicht mehr verkleinert werden kann.

Das Verfolgen der observe-Information ermöglicht es, eine globale Sicht vom Ablauf der verteilten Berechnung zu gewinnen – eine Möglichkeit, die nur in einem simulierten System gegeben ist! So läßt sich bspw. zum Zeitpunkt 1.6 folgendes **globale Zustandsdiagramm** skizzieren (zu diesem Zeitpunkt nicht mehr relevante Werte sind eingeklammert):



Der gleiche "Systemschnappschuß" läßt sich dadurch erreichen, daß zu einem bestimmten Zeitpunkt die Information ausgewertet wird, die einige Kommandos des interaktiven debuggers (→ C 5.4) liefern. So wird bspw. beim **dump-Statement** der Wert der lokalen Variablen eines Agenten gezeigt (hier etwa von Interesse für die Variable NUMBER), das **mailbox-Statement** zeigt die von einem Agenten empfangenen, aber noch nicht akzeptierten Nachrichten und mit dem **sysstatus-Kommando** kann bei Kenntnis der simulierten hardware-Konfiguration festgestellt werden, welche Nachrichten noch unterwegs sind. Diese Möglichkeiten werden in Kapitel 7.3 demonstriert.

Eine Analyse des Systemzustandes zum **Zeitpunkt 0.4**, die hier nicht nachvollzogen werden kann, da nicht das gesamte observe-Protokoll abgedruckt wurde, liefert unter Einbeziehung des interface-Agenten folgende Situation:



Man beobachtet, daß bereits zu einem Zeitpunkt, wo das Agentennetz **noch nicht vollständig aufgebaut** ist, in Teilbereichen des Netzes die EUKLID-Berechnungsoperationen gestartet werden. Nach anfänglichen Zweifeln, ob eine derartige einem "aus den Händen laufenden" Berechnung in jedem Fall (auch bei den unterschiedlichsten Nachrichtenlaufzeiten) zu einem korrekten Ergebnis führt, stellt man fest, daß sich das beobachtete Phänomen sogar sinnvoll nutzen läßt und zu einer Programmvereinfachung führt, die gleichzeitig eine flexiblere und dynamischere Gestaltung des Berechnungsablaufes ermöglicht. Allerdings scheint hierzu eine eingehendere Untersuchung der Kernidee des Algorithmus und eine "Korrektheitsargumentation" notwendig zu sein, worauf erst weiter unten eingegangen werden soll.

Zunächst soll die **Handhabung des Programms vereinfacht** werden. Man stellt sich vor, daß etwa ein Befehl `SEND GGT(X1,...,Xn) TO...` im interface genügen sollte, um die ggT-Berechnung für eine beliebige Anzahl natürlicher Zahlen zu starten, und daß das etwas lästige Gründen von n Agenten und der stets gleichartige Netzaufbau entfällt. Hierzu definiert man sich folgendes script:


```

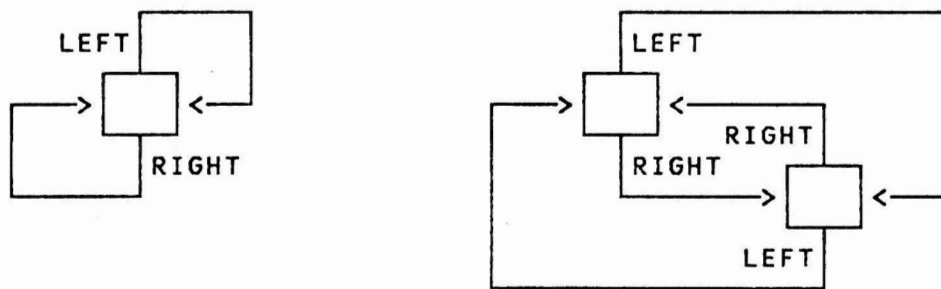
1  type SERVER is
2  script ( script: PROCESS)
3      facet COMPUTE is
4          public: GGT;
5
6          operation GGT( int: N; array (1..N) of int: NUMBERS) is
7              oper: KNOW;
8              var array (1..N) of agent: LIST;
9              functionhead MOD( int: X,Y) returns int external;
10
11             loop for I in 1..N do
12                 LIST(I):- new PROCESS(NUMBERS(I)) ;
13             endloop ;
14             loop for I in 1..N do
15                 send KNOW(LIST(MOD(I+N-2,N)+1),
16                     LIST(MOD(I,N)+1)) to LIST(I) ;
17             endloop ;
18             endoperation
19
20         endfacet
21         initial COMPUTE ;
22     endscript

```

Der SERVER besteht - etwas untypisch für CSSA - nur aus einer einzigen Facette und einer einzigen Operation. In dieser Operation werden in zwei Schleifen die PROCESS-Agenten gegründet, initialisiert und entsprechend der Doppelringtopologie miteinander bekannt gemacht. Das Pattern der Operation ist variabel lang ($\rightarrow$ B 20.3); die erste Komponente stellt die obere Grenze des arrays NUMBERS dar. Eine Nachricht muß, damit der patternmatch erfolgreich ausgeführt werden kann, zur Berechnung des ggT von n Zahlen eine Liste von n+1 Werten enthalten, wobei das erste Element den Wert n hat.

Im creation-pattern des SERVER ist ein Feld PROCESS vom Typ SCRIPT vorhanden; dieses ist aufgrund der Sichtbarkeitsgrenze von scripts notwendig, falls der SERVER und das ggT-Berechnungsprogramm in Form eigenständiger paralleler script-Moduln realisiert werden: Erst beim Gründen eines SERVER-Agenten kann diesem eine Denotation auf ein anderes - im interface lokal bekanntes - script übergeben werden. Alternativ hierzu ist es auch möglich, das PROCESS-script als **lokales script** ($\rightarrow$ B 17.3.1) im SERVER zu deklarieren. Es wird dann entweder zwischen den Zeilen 2 und 3 des SERVER oder in einem anderen Deklarationsbereich (in der Operation zwischen den Zeilen 6 und 10, auf Facettenebene oder auch am Anfang der ersten loop-Schleife) notiert; das Pattern "(script: PROCESS)" in Zeile 2 entfällt damit.

Durch eine etwas ungewöhnliche modulo-Berechnung wird in den Zeilen 15 und 16 der jeweilige Index des linken und rechten Nachbarn berechnet. Auch für die **Sonderfälle N=1 und N=2** werden die ggT-Berechnungen korrekt durchgeführt, es ergeben sich dann die folgenden Agentennetze (ohne interface-Agent), die entartete Doppelringkonfigurationen darstellen:



Im interface wird nun einmalig ein SERVER-Agent deklariert und initialisiert und ein Operationsliteral vereinbart, etwa:

```
VAR AGENT: S:-NEW SERVER(PROCESS);
OPER: GGT;
```

Dann kann durch

```
SEND GGT(n,I1,...,In) TO S;
```

die Berechnung des ggT von n Zahlen angestoßen werden. Nach Beendigung der Berechnung kann durch einen weiteren derartigen Befehl die ggT-Berechnung erneut angestoßen werden - allerdings werden dabei jedesmal neue PROCESS-Agenten gegründet!

Die Deklaration der AGENT-Variablen S ist nicht notwendig, es ist auch möglich, durch

```
SEND GGT(...) TO NEW SERVER(PROCESS);
```

die ggT-Berechnung von einem **anonymen** SERVER ausführen zu lassen.

Falls das PROCESS-script lokal im SERVER deklariert wird, vereinfacht sich die Deklaration bzw. der zuletzt angegebene send-Befehl zu:

```
VAR AGENT: S:-NEW SERVER;
```

bzw.

```
SEND GGT(...) TO NEW SERVER;
```

Das folgende **Ablaufdiagramm** zeigt den Verlauf der Berechnung für den Aufruf `SEND GGT(5,108,76,12,60,36) TO S;`. Es sei nochmals daran erinnert, daß der erste Parameter 5 die Anzahl der nachfolgenden eigentlichen Parameter angibt!

P1	108	=====>32==	"	=====>4	"					T
		"		"	"					E
P2	76=====	"		"	"					R
		"		"	"					M
P3		12==	"	=====>4	"					I
		"		"	"					N
P4					60	=>12	"	=>4		A
							"	"		T
P5					36		"	"		E
							"	"		D
t	1.4	1.5	2.1	2.5	2.9	3.0	3.5	3.6	4.3	5.4

Im Vergleich zum weiter oben angegebenen ersten Ablaufdiagramm fällt zunächst auf, daß die ggT-Berechnung etwas länger dauert (5.4 statt 4.4 Zeiteinheiten), und daß die Agenten PROCESS(4) und PROCESS(5) recht spät in Erscheinung treten. Der zusätzliche Zeitbedarf von ca. einer Zeiteinheit wird vom SERVER hervorgerufen, der die 5 Agenten erzeugt und initialisiert (pro CSSA-Statement werden 0.1 Zeiteinheiten Simulationszeit verbraucht). Die Tatsache, daß die beiden Agenten PROCESS(4) und PROCESS(5) erst relativ spät Berechnungen durchführen, liegt an einem Engpaß der simulierten hardware: Die Nachrichten vom SERVER an diese Agenten muß über den interface-Prozessor geroutet werden, dieser ist jedoch sehr stark ausgelastet und findet nur selten Zeit, Aufgaben des Kommunikationssystems zu übernehmen. Die Situation kann an dieser Stelle nicht genauer analysiert werden, da die notwendigen Voraussetzungen hierzu erst in Kapitel 8 (Simulationssystem) diskutiert werden.

Auch das **observe-Protokoll** des Agenten PROCESS(5) zeigt diesen Sachverhalt sehr deutlich:

```

+++ 0.800 PROCESS(5) IS IDLE
+++ 1.600 PROCESS(5) RECEIVES EUKLID(INT:108) FROM PROCESS(1)
+++ 1.600 PROCESS(5) IS IDLE
+++ 2.700 PROCESS(5) RECEIVES EUKLID(INT:32) FROM PROCESS(1)
+++ 2.700 PROCESS(5) IS IDLE
+++ 2.800 PROCESS(5) RECEIVES KNOW(AGENT:PROCESS(4),AGENT:
      PROCESS(1)) FROM SERVER(1)
+++ 2.800 PROCESS(5) STARTING OPERATION KNOW(AGENT:
      PROCESS(4),AGENT:PROCESS(1))
+++ 2.900 PROCESS(5) PERFORMS FACETTING : INIT --> DISPLAY
+++ 2.900 PROCESS(5) STARTING OPERATION IDLE
*** 3.000 PROCESS(5) : *** MY CURRENT NUMBER : 36 ***
+++ 3.100 PROCESS(5) SENDS EUKLID(INT:36) TO PROCESS(4)

```

```

+++ 3.200 PROCESS(5) SENDS EUKLID(INT:36) TO PROCESS(1)
+++ 3.300 PROCESS(5) PERFORMS FACETTING : DISPLAY --> ACCEPT
+++ 3.300 PROCESS(5) STARTING OPERATION EUKLID(INT:32)
+++ 3.500 PROCESS(5) PERFORMS FACETTING : ACCEPT --> DISPLAY
+++ 3.500 PROCESS(5) STARTING OPERATION IDLE
*** 3.600 PROCESS(5) : *** MY CURRENT NUMBER : 4 ***
+++ 3.700 PROCESS(5) SENDS EUKLID(INT:4) TO PROCESS(4)
+++ 3.800 PROCESS(5) SENDS EUKLID(INT:4) TO PROCESS(1)
+++ 3.800 PROCESS(5) RECEIVES EUKLID(INT:60) FROM PROCESS(4)
+++ 3.900 PROCESS(5) PERFORMS FACETTING : DISPLAY --> ACCEPT
+++ 3.900 PROCESS(5) RECEIVES EUKLID(INT:4) FROM PROCESS(1)
+++ 3.900 PROCESS(5) IS IDLE
+++ 4.100 PROCESS(5) RECEIVES EUKLID(INT:12) FROM PROCESS(4)
+++ 4.100 PROCESS(5) IS IDLE
+++ 4.700 PROCESS(5) RECEIVES EUKLID(INT:4) FROM PROCESS(4)
+++ 4.700 PROCESS(5) IS IDLE

```

Interessanterweise erhält der Agent bereits zu den Zeitpunkten 1.6 und 2.7 EUKLID-Nachrichten mit den Werten 108 und 32 von seinem rechten Nachbaragenten PROCESS(1), noch bevor er die KNOW-Operation zum Zeitpunkt 2.8 ausgeführt hat. Da er sich jedoch in der INIT-Facette befindet, kann er diese Nachrichten nicht verarbeiten. Erst zum Zeitpunkt 3.3, wenn die ACCEPT-Facette betreten wird, kann eine der bereits empfangenen EUKLID-Nachrichten sofort verarbeitet werden. Zum Zeitpunkt 3.8 ist ein weiteres interessantes Phänomen zu beobachten: Während der Agent in der Facette DISPLAY die idle-Operation ausführt, wird asynchron eine Nachricht von seinem rechten Nachbarn in die mailbox eingetragen.

Eine genauere Betrachtung der beiden Zustandstabellen zeigt noch **weitere Unterschiede**: In der ersten Tabelle nimmt PROCESS(5) nacheinander die Werte 36 (Initialwert), 12 und 4 an; in der zweiten Tabelle geht der Wert von 36 direkt auf 4 über. Zwar wird dieser Wert auch von PROCESS(1) hervorgerufen, jedoch nicht wie in der ersten Tabelle durch den Wert 4, sondern durch den Wert 32! Schließlich bekommt in der zweiten Tabelle PROCESS(4) seinen Wert 4 durch PROCESS(5) und nicht durch PROCESS(3). Dies zeigt deutlich, daß es sich bei diesem ggT-Algorithmus um ein **nichtdeterministisches Verfahren** handelt: Zwar wird das gleiche Ergebnis berechnet, jedoch auf unterschiedliche Weise. Die Ursache liegt darin, daß wegen des SERVER die Agenten zu anderen Zeitpunkten gegründet werden und das Betriebssystem sie anders auf die baumförmig angeordneten hardware-Prozessoren verteilt, was zu unterschiedlichen Nachrichtenlaufzeiten führt. Die Einflüsse unterschiedlicher hardware-Konfigurationen werden in Kapitel 8 genauer diskutiert.

Bei Ausführung von SEND GGT(2,111,81) TO S; wird ein Agenten-netz aus **zwei PROCESS-Agenten** in der Form einer oben skizzierten leicht entarteten Doppelringkonfiguration aufgebaut. Es werden folgende Meldungen erzeugt:

```

*** 1.200 PROCESS(1) *** MY CURRENT NUMBER : 111 ***
*** 2.800 PROCESS(2) *** MY CURRENT NUMBER : 81 ***
*** 3.900 PROCESS(1) *** MY CURRENT NUMBER : 30 ***
*** 4.800 PROCESS(2) *** MY CURRENT NUMBER : 21 ***
*** 5.900 PROCESS(1) *** MY CURRENT NUMBER : 9 ***

```

```

*** 6.800 PROCESS(2) *** MY CURRENT NUMBER : 3 ***
*** 7.800 PROCESS(1) *** MY CURRENT NUMBER : 3 ***

```

Die beiden Agenten können aufgrund des Algorithmus nur abwechselnd arbeiten, eine parallele Ausführung von Berechnungen ist nicht möglich. Daß dennoch in einem gewissen Sinne eine **verteilte Berechnung des ggT** stattfindet, zeigt die Gegenüberstellung des Zustandsdiagramms mit dem üblichen sequentiellen Berechnungsschema: PROCESS(1) durchläuft die Folge 111,30,9,3 und PROCESS(2) die bis auf den letzten gemeinsamen Wert disjunkte Komplementärfolge 81,21,3.

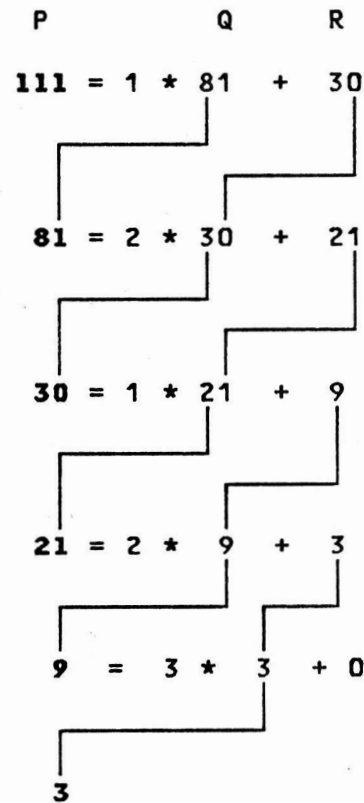
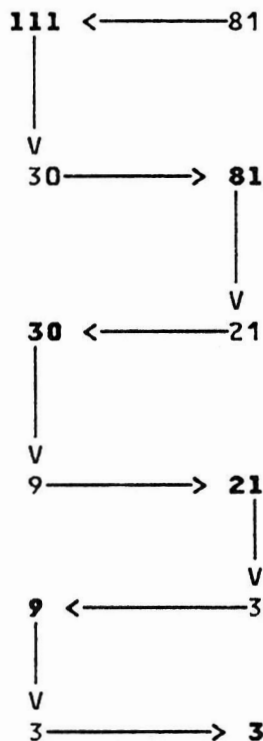
```

(* ggT-Berechnung sequentiell iterativ *)
function GGT( int: P,Q) returns int is
  var int: R;
  loop while Q /= 0 do
    R:=P-(P/Q)*Q; P:=Q; Q:=R;
  endloop;
  return P;
endfunction;

(* rekursiv ginge es natürlich auch : *)
... if MOD(P,Q)=0 then return Q;
    else return GGT(Q,MOD(P,Q));
endif;...
(* wieso nicht ... else return GGT(P,MOD(P,Q)); ?? *)

```

PROCESS(1) PROCESS(2)



Im letzten Beispiel der verteilten ggT-Berechnung besitzt jeder Agent **zwei** Bekanntschaften auf den anderen Agenten. Tatsächlich wird jeder neue Wert in zwei Nachrichten, über LEFT und RIGHT, versandt. Dies ist unnötig. Anstelle einer Doppelringkonfiguration genügt hier auch eine einfache Ringkonfiguration. Dies läßt vermuten, daß bei der Berechnung des ggT von mehr als zwei Zahlen ebenfalls eine einfache Ringkonfiguration genügt – das ist tatsächlich der Fall, wie weiter unten gezeigt wird. Es soll hier diese Vermutung zunächst experimentell an einem Beispiel überprüft werden.

Um eine **einfache Ringkonfiguration** aufzubauen, ist es nicht notwendig, das PROCESS-script zu ändern – es genügt, dafür Sorge zu tragen, daß einer der beiden "Kommunikationskanäle" LEFT oder RIGHT eines jeden Agenten anstatt in das Agentennetz ins "Leere" führt. Der SERVER kann hierfür in der vorliegenden Form allerdings nicht verwendet werden, die KNOW-Operation muß wieder explizit aufgerufen werden. Damit kein Laufzeitfehler entsteht ("SEND TO <NOAGENT> ISSUED"), wird LEFT kurzerhand auf das interface gesetzt. Im interface werden dazu folgende Kommandos ausgeführt:

```
VAR AGENT : A1,A2,A3,A4,A5;
A1:-NEW PROCESS(108);
A2:-NEW PROCESS(76);
A3:-NEW PROCESS(12);
A4:-NEW PROCESS(60);
A5:-NEW PROCESS(36);
OPER:KNOW;
SEND KNOW(SELF,A2) TO A1;
SEND KNOW(SELF,A3) TO A2;
SEND KNOW(SELF,A4) TO A3;
SEND KNOW(SELF,A5) TO A4;
SEND KNOW(SELF,A1) TO A5;
RUN;
```

Hierbei wird durch das Schlüsselwort 'self' eine Bekanntschaft auf den Agenten selbst symbolisiert, dies ist in diesem Fall das interface.

Tatsächlich wird auch hier der ggT korrekt berechnet, das folgende Zustandsdiagramm zeigt jedoch, daß die Berechnung mit 8.2 Zeiteinheiten fast **doppelt so lange** dauert wie bei der Doppelringkonfiguration (4.4 Zeiteinheiten). Die Wertabhängigkeiten wurden in das Zustandsdiagramm nicht eingezeichnet, da hier stets PROCESS(i) PROCESS(mod(i,5)+1) beeinflußt.

PROC (1)	108			36		12				4	T E R M I N A T E D
PROC (2)	76					4					
PROC (3)		12					4				
PROC (4)		60	12					4			
PROC (5)		36			12				4		
time	0.3	0.7	1.4	2.4	2.6	3.4	4.5	5.2	6.5	7.4	8.2

Da das Simulationssystem den Blockierungszustand erkennt und meldet und die Kontrolle dann an den interface-Agenten und damit an den Benutzer übergibt, kann durch das **mailbox-Kommando** (--> C 5.2) festgestellt werden, welche Nachrichten dem interface von den PROCESS-Agenten (über LEFT) zugesandt wurden. Damit **erübrigt sich das PRINT-Statement**. Das Ergebnis des mailbox-Kommandos ist die folgende Liste:

MAILBOX OF INTERFACE(1) :

```
(1) EUKLID(INT:108)
(2) EUKLID(INT:76)
(3) EUKLID(INT:60)
(4) EUKLID(INT:36)
(5) EUKLID(INT:12)
(6) EUKLID(INT:12)
(7) EUKLID(INT:36)
(8) EUKLID(INT:12)
(9) EUKLID(INT:12)
(10) EUKLID(INT:4)
(11) EUKLID(INT:4)
(12) EUKLID(INT:4)
(13) EUKLID(INT:4)
(14) EUKLID(INT:4)
```

Offenbar stellt der kleinste Wert, also 4, den ggT dar.

Bisher noch ungelöst ist das Problem, den **Blockierungszustand** und damit die Terminierung lediglich durch **CSSA-Sprachmittel** ohne Zuhilfenahme des CSSA-Simulationssystems ("SYSTEM TERMINATED") zu erkennen. Nur in einem Sonderfall, nämlich dann, wenn die Variable NUMBER den Wert 1 annimmt, könnte ein Agent dies dem interface als "Endergebnis" der verteilten ggT-Berechnung melden. Auf diese Weise wäre eine **Halbentscheidbarkeit der Teilerfremdheit** realisiert: Ein Benutzer würde im Falle der Teilerfremdheit schließlich eine Meldung bekommen, im anderen Fall jedoch nicht wissen, wie lange er noch auf "keine Meldung" warten soll!

Hier führt eine Beobachtung an den dem interface zugesandten Nachrichten weiter: Der ggT-Wert 4 wird dem interface insgesamt fünfmal zugesandt, dies ist für keinen anderen Wert der Fall (der Wert 12 wird aber immerhin viermal zugesandt !). Eine nicht ganz einfache Überlegung, die einige der weiter unten aufgeführ-

ten Korrektheitsargumente benutzt, zeigt, daß dies tatsächlich genau für den ggT-Wert gilt. Wir beschränken uns an dieser Stelle auf den eher informellen Nachweis einer wichtigen Voraussetzung zur Allgemeingültigkeit der Beobachtung:

Behauptung (EINDEUTIGKEIT):

Eine Wertbelegung, die im Laufe der Berechnung von allen Agenten (eventuell zu unterschiedlichen Zeitpunkten) angenommen wird, ist eindeutig.

Nachweis:

Falls sie nicht eindeutig ist, dann gibt es jedenfalls einen größten und einen kleineren derartigen Wert. Betrachte denjenigen Agenten, der als erster (!) den größten Wert verändert (d.h. verkleinert!). Dies ist nur durch Empfang einer Nachricht von einem anderen Agenten möglich, der bereits einen kleineren Wert als den größten angenommen hat. (Die Wertbelegung eines einzelnen Agenten ist eine mit der Zeit monoton abnehmende Zahlenfolge.) Das steht jedoch im Widerspruch zur Annahme, daß der Empfänger der erste derartige Agent ist.

Bemerkung:

Es werden hier implizit bereits einige Beobachtungen und Argumente benutzt, die erst weiter unten genauer untersucht werden. Auch ist beim Begriff "als erster" (hier in einem kausalen Sinne gebraucht) extreme Vorsicht geboten - bei Argumentationen über einem "relativistischen" System, wo schon die Existenz eines wohldefinierten Zeitbegriffs zweifelhaft ist! Es sei noch angemerkt, daß ein eher mathematisch orientierter Nachweis der Behauptung möglich ist, indem zunächst gezeigt wird, daß eine gemeinsame Wertbelegung alle im Laufe der Berechnung angenommenen größeren Belegungen der einzelnen Agenten teilt. Die noch nachzuweisende Tatsache, daß schließlich der ggT berechnet wird und die verteilte Berechnung dann blockiert, schließt damit eine andere (größere) gemeinsame Belegung aus.

Die obige Beobachtung führt zu folgender **Lösungsidee des Terminierungsproblems:** Ein MANAGER wird von allen PROCESS-Agenten über jeweils neu berechnete Werte informiert. Sobald ein Wert n mal gemeldet wurde, ist der ggT gefunden - nur das jeweilige Minimum ist dabei ein Kandidat, größere Werte können ignoriert werden. Das CSSA-script des MANAGER wird hier nicht aufgeführt, da das Programm recht trivial ist und keine wesentlichen neuen CSSA-Sprachkonstrukte enthält. Die Operationen des SERVER und des MANAGER können natürlich in sinnvoller Weise zu einem einzigen script zusammengefaßt werden.

Das PROCESS-script soll nun so abgeändert werden, daß **beliebige logische Netztopologien** konfiguriert werden können. Hierbei bietet es sich an, die Menge der erreichbaren Nachbaragenten in einem **set** zu speichern. Die Zeile mit der Nummer 5

```
var agent : LEFT,RIGHT;
```

wird dazu ersetzt durch die Definition

```
var set of agent : ADJACENTS;
```

und die beiden Zeilen 13 und 14

```
send EUKLID(NUMBER) to LEFT;
send EUKLID(NUMBER) to RIGHT;
```

werden durch die Schleife

```
loop for A of ADJACENTS do
  send EUKLID(NUMBER) to A;
endloop;
```

ersetzt. Hierbei ist A eine implizit deklarierte Konstante vom Typ AGENT, die in undefinierter Reihenfolge alle Elemente der Menge durchläuft, indem sie die entsprechenden Werte annimmt. Falls die Menge leer sein sollte, wird die Schleife nicht ausgeführt.

Die Operation KNOW bekommt ein anderes Pattern, das es ermöglicht, eine variable Zahl aktueller Parameter anzugeben:

```
operation KNOW( int: N; array (1..N) of agent: LIST) is
  loop for J in 1..N do
    put LIST(J) into ADJACENTS;
  endloop;
```

Die zusätzliche Schleife dient dazu, die übermittelten Nachbaragenten in die Menge einzufügen; ein direktes Übermitteln von Mengen in Nachrichten ist nicht möglich.

Mit dieser kleinen Änderung kann nun bspw. ein **vollständiger Graph** aufgebaut werden, bei dem jeder PROCESS-Agent jeden anderen kennt. Um die ggT-Berechnung für 5 Zahlen zu initiieren, würden im interface nach der Gründung und Initialisierung der 5 Agenten folgende send-Kommandos ausgeführt:

```
SEND KNOW(4,A2,A3,A4,A5) TO A1 ;
SEND KNOW(4,A1,A3,A4,A5) TO A2 ;
SEND KNOW(4,A1,A2,A4,A5) TO A3 ;
SEND KNOW(4,A1,A2,A3,A5) TO A4 ;
SEND KNOW(4,A1,A2,A3,A4) TO A5 ;
```

Zum Zeitpunkt 6.3 ist die Berechnung beendet, auch hier mit dem korrekten Ergebnis 4. Der relativ große Zeitbedarf im Vergleich zur Doppelringkonfiguration hat mehrere Ursachen: Zum einen dauert die Ausführung der KNOW-Operation länger und werden in der DISPLAY-Facette 4 statt 2 send-Befehle ausgeführt, zum anderen sind wesentlich mehr Nachrichten unterwegs (52 statt 26 beim Doppelring und 14 beim einfachen Ring), die die Kommunikationskanäle der simulierten hardware belasten. Von den 44 Nachrichten wurden übrigens genauso wie bei der Doppelringkonfiguration nur 8 akzeptiert!

Nachdem nun die Möglichkeit besteht, beliebige Netztopologien zu konfigurieren, stellt sich die Frage nach der **optimalen (logischen) Konfiguration**. Offenbar ist diese in hohem Maße von der zugrundeliegenden hardware abhängig: Man erwartet eine effi-

ziente Ausführung, wenn eine eins-zu-eins Beziehung zwischen den Bussen und Prozessoren einerseits und den logischen Kommunikationskanälen andererseits besteht. Sollen Experimente mit der **inhärenten Parallelität** des Algorithmus durchgeführt werden, so möchte man von der Übertragungszeit von Nachrichten abstrahieren, um verschiedene logische Konfigurationen bei unterschiedlichen Belegungen untersuchen zu können. Das CSSA-Simulationssystem ermöglicht es, derartige Experimente in einfacher Weise durchzuführen – eine Reihe davon sind in Kap. 8.4 (dort allerdings mit einer Nachrichtenlaufzeit von 1.0 statt wie hier mit 0.1 Zeiteinheiten) aufgeführt.

Der folgende Satz zeigt, daß jedenfalls nicht beliebige logische Konfigurationen von 2 oder mehr PROCESS-Agenten zu einer korrekten Berechnung des ggT führen, sondern daß an das Agentennetz gewisse Anforderungen gestellt werden:

Behauptung (ERREICHBARKEIT):

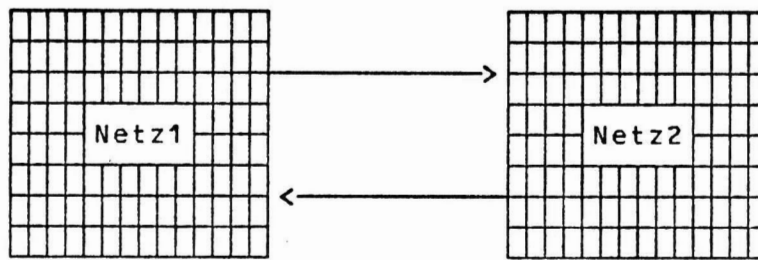
Damit bei jeder initialen Belegung der ggT berechnet wird, muß der Graph des PROCESS-Agentennetzes stark zusammenhängend sein.

Beweis durch Widerspruch:

Ein Agentennetz aus zwei oder mehr Agenten sei nicht stark zusammenhängend. Es existieren also zwei verschiedene Agenten a und b , so daß von a kein Weg nach b führt. Sei nun folgende Belegung gegeben, die zu einer Partition des Netzes führt: a selbst und alle von a erreichbaren Agenten werden mit einer natürlichen Zahl $n > 0$ initialisiert, alle anderen (also insbesondere b) mit $2n$. Dann führt keine Kante von der Klasse $[n]$ der mit n markierten Agenten in die Klasse $[2n]$ der mit $2n$ markierten Agenten. Ein Agent aus $[2n]$ kann daher niemals eine Nachricht mit dem Wert n empfangen: Das System ist bereits nach seiner Initialisierung blockiert, ohne daß sich ein Wert geändert hat. Obwohl der ggT dieser Belegung n beträgt, ändert der Agent b seine Belegung von $2n$ nicht.

Daß die Erreichbarkeit auch eine **hinreichende Bedingung** für die korrekte ggT-Berechnung bei jeder Belegung von Agenten des oben angegebenen Typs darstellt, wird erst weiter unten gezeigt.

Zwei Netze, in denen zunächst unabhängig voneinander ggT-Berechnungen durchgeführt werden, können in gewisser Weise so zusammengeschaltet werden, daß der ggT des Gesamtnetzes berechnet wird. Da offenbar zusammenhängende Komponenten zu einem Knoten abstrahiert werden können und die ggT-Berechnung folglich in derartige (zusammenhängende) Komponenten zerlegt werden kann, rechtfertigt der folgende Satz eine **dynamische Netzerweiterung** derart, daß nach der Terminierung zweier (zusammenhängender) Berechnungsnetze diese zu einem einzigen zusammenhängenden Netz erweitert werden, anschaulich:

**Satz (ZERLEGBARKEIT):**

$$\text{ggT}(X_1, \dots, X_n, Y_1, \dots, Y_m) = \text{ggT}(\text{ggT}(X_1, \dots, X_n), \text{ggT}(Y_1, \dots, Y_m))$$

Beweis:

Bezeichne $A = \text{ggT}(X_1, \dots, X_n)$, $B = \text{ggT}(Y_1, \dots, Y_m)$ und $C = \text{ggT}(A, B)$. Zu zeigen ist also $C = \text{ggT}(X_1, \dots, X_n, Y_1, \dots, Y_m)$. Der Beweis wird in zwei Schritten geführt. Zunächst wird gezeigt, daß C gemeinsamer Teiler von $X_1, \dots, X_n, Y_1, \dots, Y_m$ ist; anschließend, daß jeder gemeinsame Teiler von $X_1, \dots, X_n, Y_1, \dots, Y_m$ auch Teiler von C ist. Daraus folgt, daß C der größte gemeinsame Teiler ist. Im zweiten Schritt wird die aus der Primfaktorzerlegung sofort ableitbare Tatsache benutzt, daß jeder gemeinsame Teiler einer Menge natürlicher Zahlen den ggT dieser Menge teilt.

- 1a: $C = \text{ggT}(A, B) \longrightarrow C$ teilt A und C teilt B .
- 1b: $A = \text{ggT}(X_1, \dots, X_n) \longrightarrow A$ teilt $X_1, \dots, X_n$.
- 1c: $B = \text{ggT}(Y_1, \dots, Y_m) \longrightarrow B$ teilt $Y_1, \dots, Y_m$.
- 1d: Aus der Transitivität der Teilbarkeitsrelation folgt, daß C $X_1, \dots, X_n, Y_1, \dots, Y_m$ teilt.
- 2: Sei D ein beliebiger gemeinsamer Teiler von $X_1, \dots, X_n, Y_1, \dots, Y_m$.
 - $\longrightarrow D$ ist gemeinsamer Teiler von $X_1, \dots, X_n$ und D ist gemeinsamer Teiler von $Y_1, \dots, Y_m$.
 - $\longrightarrow D$ teilt $\text{ggT}(X_1, \dots, X_n)$ und D teilt $\text{ggT}(Y_1, \dots, Y_m)$.
 - $\longrightarrow D$ ist gemeinsamer Teiler von A und B .
 - $\longrightarrow D$ teilt $\text{ggT}(A, B)$.
 - $\longrightarrow D$ teilt C .

Wenn man nun noch zeigen könnte, daß der Zeitpunkt des Zusammenschaltens von Netzen unerheblich ist, die ablaufenden Berechnungen sich also nicht "wesentlich" stören, dann hätte man aufgrund der oben angedeuteten Analogie der verteilten ggT-Berechnung zweier Agenten zum üblichen sequentiellen Berechnungsschema vielleicht eine gute Ausgangsbasis, um die Hinlänglichkeit der Erreichbarkeitsbedingung und damit die Korrektheit des Verfahrens einzusehen...

Tatsächlich wird die Korrektheit des Algorithmus später auf eine etwas andere Art nachgewiesen, es soll zunächst die weiter oben gemachte Beobachtung, daß mit der verteilten ggT-Berechnung bereits begonnen wird, während das Netz noch aufgebaut wird, ausgenutzt werden. Es wird dazu ein script konzipiert, das eine **dynamische Netzerweiterung** erlaubt, noch während die Berechnung abläuft.

```

1  type PROCESS is
2  script ( int: N) assert N>0
3      var int: NUMBER := N ;
4      var set of agent: ADJACENTS ;
5
6      facet COMPUTE is
7          public: KNOW,EUKLID;
8
9          operation KNOW( agent: A) assert A <> noagent is
10             put A into ADJACENTS ;
11             send EUKLID(NUMBER) to A ;
12         endoperation
13
14         operation EUKLID( int: NEIGHBOUR)
15             assert NEIGHBOUR<NUMBER is
16             functionhead MOD( int: I,J) returns int external;
17
18             NUMBER := MOD(NUMBER-1,NEIGHBOUR)+1 ;
19             loop for A of ADJACENTS do
20                 send EUKLID(NUMBER) to A ;
21             endloop ;
22         endoperation
23
24         operation EUKLID is null; endoperation
25
26     endfacet
27     initial COMPUTE ;
28 endscript

```

Da die KNOW-Operation jetzt jederzeit aufrufbar sein soll, befindet sie sich in der gleichen Facette wie die EUKLID-Operation. Sobald sie aktiviert wird, sendet der Agent an seinen neuen Bekannten seinen augenblicklichen Zahlwert; die Bekanntschaft wird im set ADJACENTS vermerkt, damit bei einer späteren Ausführung der EUKLID-Operation die neue Belegung allen bekannten Agenten mitgeteilt werden kann.

Im Unterschied zur letzten Fassung des PROCESS-scripts existiert nur eine einzige Facette; auf die DISPLAY-Facette konnte verzichtet werden, das Versenden des neuen Wertes wird nun direkt in der EUKLID-Operation vorgenommen. Die Bekanntschaften müssen einem Agenten in einer Folge einzelner KNOW-Aufrufe übermittelt werden, auf ein variabel langes Pattern wurde verzichtet. Da kein PRINT-Befehl vorgesehen ist, sollte durch Aufruf der KNOW-Operation jedem Agenten eine Bekanntschaft auf einen MANAGER oder das interface übergeben werden – sobald so viele identische Werte zugesandt wurden wie PROCESS-Agenten existieren, ist der ggT gefunden.

Die Operation EUKLID ist überladen (→ B 19.3): Sie ist doppelt definiert. Die zweite Definition ist an sich nicht notwendig, sie wurde hier nur angegeben, um das Prinzip des overloadings darzustellen. Welche der beiden Operationsversionen bei Empfang einer EUKLID-Nachricht aktiviert wird, hängt vom Ergebnis des patternmatches und des Wertes der evtl. vorhandenen assertion ab. Dazu wird zunächst die textuell erste Version überprüft. Falls der patternmatch erfolgreich ausgeht und die assertion den Wert TRUE liefert, wird sie aktiviert. Ansonsten

wird die zweite Version entsprechend geprüft. Da hier kein Pattern und keine assertion angegeben ist, geht der patternmatch in jedem Fall erfolgreich aus (--> B 20.3,20.4), die Operation wird aktiviert - jedoch alleine mit dem Ergebnis, daß die (an sich nicht akzeptierbare) Nachricht aus der mailbox verschwindet! Den gleichen Effekt hätte man mit einem etwas größeren dokumentarischen Wert auch durch folgende Definition erreichen können:

```

operation EUKLID( int: NEIGHBOUR) assert NEIGHBOUR>=NUMBER is
    null;
endoperation

```

Denkübungen:

- 1) Einem neuen Bekannten kann die Menge aller eigenen Bekannten übermittelt werden, indem in die KNOW-Operation folgende Schleife eingefügt wird:

```

loop for B of ADJACENTS do
    send KNOW(B) to A ;
endloop ;

```

Sollte diese Schleife sinnvollerweise am Anfang der Operation oder erst nach dem Statement **"put A into ADJACENTS"** stehen ?

- 2) Macht es einen Unterschied, ob die assertion der KNOW-Operation um "... **and not (A in ADJACENTS)**" ergänzt wird ?
- 3) Macht es einen Unterschied, wenn außerdem gleich zu Beginn der KNOW-Operation "**put self into ADJACENTS**" ausgeführt wird ? Kann der gleiche Effekt auch durch das Statement "**send KNOW (self) to self**" erreicht werden ?

Manche Menschen würden eher sterben als nachzudenken.
Und sie tun es auch.

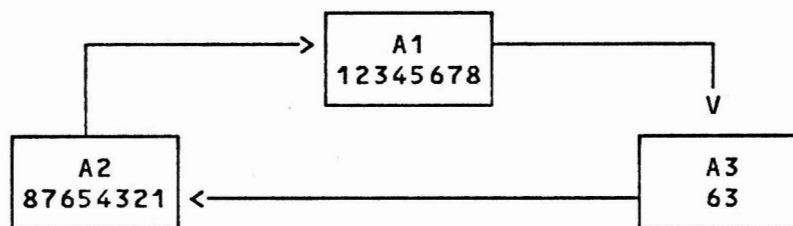
Bertrand Russell

Um ein dynamisches Netz aus PROCESS-Agenten der oben abgedruckten Form aufzubauen, werden im interface zunächst folgende Befehle ausgeführt:

```
VAR AGENT: A1,A2,A3,A4,A5;  
A1 :- NEW PROCESS(12345678);  
A2 :- NEW PROCESS(87654321);  
A3 :- NEW PROCESS(63);  
SEND KNOW(INTERFACE) TO A1;  
SEND KNOW(INTERFACE) TO A2;  
SEND KNOW(INTERFACE) TO A3;  
SEND KNOW(A1) TO A2;  
SEND KNOW(A2) TO A3;  
SEND KNOW(A3) TO A1;
```

INTERFACE ist ein Schlüsselwort der CSSA-Sprache, es stellt eine Konstante vom Typ AGENT dar. Da die Befehlsfolge im interface selbst ausgeführt wird, ist es in diesem Fall identisch zu SELF.

Nachdem die KNOW-Nachrichten akzeptiert wurden, liegt unter Vernachlässigung des interface-Agenten folgende Situation vor:

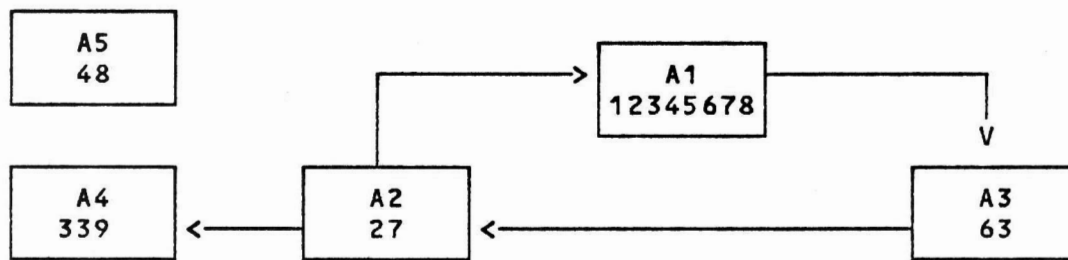


Das Netz ist stark zusammenhängend, eine ggT-Berechnung kann ablaufen.

Zum Zeitpunkt 2.3 hat Agent A2 bereits die Nachricht von Agent A3 empfangen und als neuen Wert 27 ermittelt. Im interface werden nun einige Kommandos ausgeführt, die zu einer Erweiterung des Netzes führen:

```
A4 :- NEW PROCESS(339);  
A5 :- NEW PROCESS(48);  
SEND KNOW(INTERFACE) TO A4;  
SEND KNOW(INTERFACE) TO A5;  
SEND KNOW(A4) TO A2;
```

Es ergibt sich folgende Situation:



Zum Zeitpunkt 3.7 verändert A4 seinen Wert auf 15 und A1 seinen Wert auf 9. In der mailbox des interface befinden sich nun die folgenden Nachrichten:

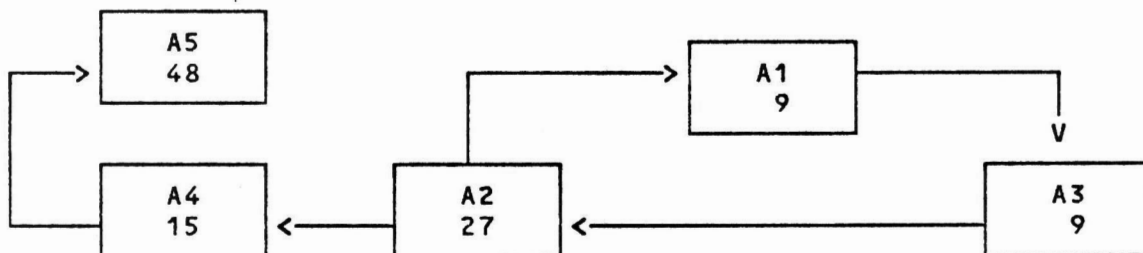
MAILBOX OF INTERFACE(1):

- (1) EUKLID:INT(87654321)
- (2) EUKLID:INT(12345678)
- (3) EUKLID:INT(63)
- (4) EUKLID:INT(27)
- (5) EUKLID:INT(339)
- (6) EUKLID:INT(48)

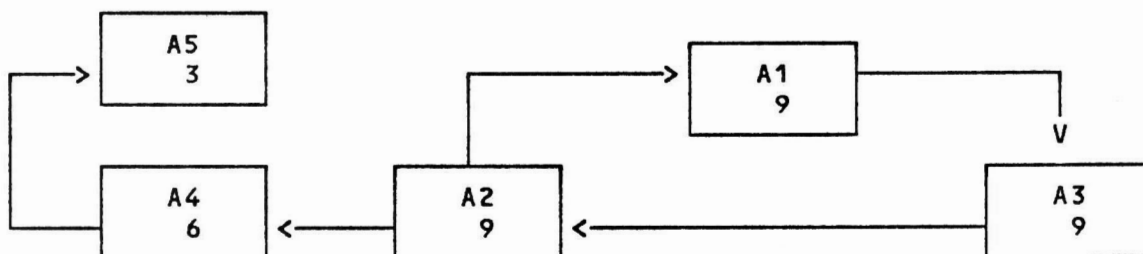
Die neuen Werte von A4 und A1 sind offenbar noch nicht eingetroffen. Das Netz wird durch

SEND KNOW(A5) TO A4;

erweitert. Zum Zeitpunkt 4.3 berechnet auch A3 einen neuen Wert 9, so daß nun folgende Situation vorliegt:



Die Berechnung schreitet weiter fort, zum Zeitpunkt 5.1 setzt A5 seinen Wert auf 3, zum Zeitpunkt 6.0 erhält A2 den Wert 9 und zum Zeitpunkt 7.2 A4 den Wert 6. In diesem Zustand blockiert das System:



Der Graph ist nicht stark zusammenhängend, die Belegung ist unterschiedlich, es ist nicht der ggT (mit dem Wert 3) der 5 Zahlen berechnet worden. In dem stark zusammenhängenden Teilnetz A1,A2,A3 ist dagegen der (lokale) ggT berechnet worden! Das Gesamtnetz kann durch Hinzufügen einer Kante von A5 nach A1 stark zusammenhängend gemacht werden:

SEND KNOW(A1) TO A5;

Nun erhalten nacheinander A1 (Zeitpunkt 9.3), A3 (9.9), A2 (11.3) und schließlich A4 (12.4) den Wert 3, was den ggT aller 5 Ausgangswerte darstellt; und tatsächlich finden sich zum Zeitpunkt 13.3 auch 5 EUKLID-Nachrichten mit dem Wert 3 in der mailbox des interface.

Es sollen nun noch zum Abschluß dieses längeren Beispiels die Argumente dafür geliefert werden, daß durch ein stark zusammenhängendes PROCESS-Agentennetz **bei jeder Belegung der ggT berechnet wird**. Hierzu ist i.w. zu zeigen, daß im Blockierungszustand die gemeinsame Belegung dem ggT der Ausgangsbelegung entspricht, daß dieser Zustand schließlich erreicht wird, und daß das evtl. Überholen von Nachrichten unerheblich ist. Die Teilargumente werden in Form einzelner Beobachtungen und Lemmata aufgeführt; bei CSSA-Programmen wird – wenn überhaupt – nur in informeller Weise argumentiert. Es wird generell ein stark zusammenhängendes Netz aus zwei oder mehr PROCESS-Agenten vorausgesetzt; der interface-Agent, ein evtl. vorhandener SERVER oder MANAGER sowie die Möglichkeit der dynamischen Netzerweiterung werden zunächst ignoriert.

Beobachtung 1:

Jeder (neue) Wert eines Agenten wird allen Nachbarn mitgeteilt.

Beobachtung 2:

Ein Agent, der die KNOW-Operation ausgeführt hat, kann höchstens dadurch blockiert sein, daß er auf eine akzeptierbare EUKLID-Nachricht wartet. (Eine EUKLID-Nachricht heißt akzeptierbar, wenn sie einen kleineren Wert als die eigene Belegung enthält.)

Beobachtung 3:

Ist die Belegung aller Agenten nicht identisch, so ist das System nicht blockiert. (Das System ist blockiert, wenn jeder Agent blockiert ist und keine akzeptierbare Nachricht mehr existiert.)

Argumentation:

Nach Voraussetzung existieren Agenten a und b mit unterschiedlicher Belegung. OBdA sei $\text{Belegung}(a) < \text{Belegung}(b)$. Dann existieren in einer Agentenfolge $a=X_1, X_2, \dots, X_n=b$ eines Weges (Zusammenhang!) von a nach b zwei benachbarte Agenten X_i, X_{i+1} mit $\text{Belegung}(X_i) < \text{Belegung}(X_{i+1})$. Agent X_{i+1} kann zumindest die Nachricht von X_i (eine solche existiert nach Beobachtung 1) akzeptieren.

Beobachtung 4:

Ist das System nicht blockiert, so kann ein Agent durch Empfang einer Nachricht seine Belegung echt verkleinern (niemals aber gleich belassen oder gar vergrößern). Die Belegung ist stets eine positive natürliche Zahl.

Nachweis:

- a) assertion NEIGHBOUR<NUMBER
- b) $x < y \longrightarrow \text{mod}(y-1, x) + 1 < y$
- c) $\text{mod}(y-1, x) + 1 > 0$
- d) assertion $N > 0$

Lemma 1 (EUKLID'SCHE REGEL):

$1 \leq X_i < X_j$ ($i, j = 1, \dots, n$) $\longrightarrow \text{ggT}(X_1, \dots, X_n) = \text{ggT}(X_1, \dots, X_k, \text{mod}(X_j - 1, X_i) + 1, X_l, \dots, X_n)$, wobei $k = j - 1$ und $l = j + 1$.

Beweisskizze:

Es genügt zu zeigen, daß die (endliche) Menge der gemeinsamen Teiler identisch ist. Man benutzt beim Nachweis, daß jeder gemeinsame Teiler der einen Menge gemeinsamer Teiler der anderen Menge ist (und umgekehrt) die Tatsache, daß eine Zahl, die Dividend und Divisor teilt, auch den Rest teilt, und sofern sie Rest und Divisor einer ganzzahligen Division mit Rest teilt, auch den Dividenden teilt (Dividend > Divisor).

Beobachtung 5:

Wird im Laufe der verteilten Berechnung eine identische Belegung aller Agenten erreicht, so ist das System blockiert.

Argumentation:

Alle Nachrichten, die noch unterwegs sind, enthalten Werte, die größer oder gleich der gemeinsamen Belegung sind ($\longrightarrow$ Beobachtung 4). Derartige Nachrichten sind dann nicht mehr akzeptierbar.

Lemma 2 (TERMINIERUNG):

$X_1 = X_2 = \dots = X_n \longrightarrow \text{ggT}(X_1, \dots, X_n) = X_1$.

Beweis klar nach Definition von "Teiler" und ggT.

Die **Gesamtargumentation**, zunächst unter Vernachlässigung der Möglichkeit, daß sich Nachrichten überholen und der Tatsache, daß die Berechnung u.U. bereits beginnt, während das Netz noch aufgebaut wird, verläuft etwa folgendermaßen: Solange die Belegung des Systems nicht identisch ist, ist es nicht blockiert (B3). Dann kann ein Agent seine Belegung verkleinern (B4), was er schließlich auch tut (endliche Nachrichtenlaufzeit und B2). Die neue Belegung hat den gleichen ggT wie die alte (Lemma 1). Nach einer endlichen Anzahl von Belegungsänderungen ist eine gemeinsame Belegung erreicht, evtl. die kleinste gemeinsame Belegung mit dem Wert 1 (B4). Dann ist das System blockiert (B5), die gemeinsame Belegung stellt den ggT dar (Lemma 2).

Beobachtung 6:

Nachrichten dürfen sich ohne Konsequenz für das Endergebnis überholen.

Bemerkung:

Überholen heißt nicht nur, daß eine zweite Nachricht schneller als eine erste über die Kommunikationskanäle transportiert wird, sondern auch, daß Nachrichten nicht nach dem FIFO-Prinzip aus der mailbox ausgewählt werden.

Nachweis:

Eine überholte Nachricht richtet keinen Schaden an, da sie einen größeren Wert als die schnellere Nachricht hat und daher nicht mehr akzeptiert werden kann: Die schnellere Nachricht hat dazu geführt, daß die Belegung des Agenten kleiner (oder höchstens gleich) ihrem Wert wird; die überholte Nachricht bleibt in der mailbox. Dies ist gleichbedeutend damit, daß die überholte Nachricht nie abgesandt wurde, etwa weil der Sender-Agent von vornherein den kleineren Wert der schnelleren Nachricht besaß. Der ggT-Wert bleibt davon unbeeinträchtigt.

Aus der Tatsache, daß sich Nachrichten überholen dürfen, folgt auch, daß es nicht schadet, wenn die Berechnung bereits vor dem vollständigen Aufbau des Netzes beginnt, und daß die **dynamische Netzerweiterung** in dem Sinne **korrekt** ist, als letztlich der ggT des Gesamtnetzes berechnet wird, sofern schließlich ein stark zusammenhängendes Netz vorliegt: Angenommen, es wird dynamisch eine zusätzliche Bekanntschaft von einem Agenten a zu einem Agenten b eingerichtet. Sowohl a als auch b können im Verlaufe der bisherigen Berechnung ihren Wert bereits verändert (vermindert!) haben. Einer neu eingerichteten Bekanntschaft wird jedenfalls der aktuelle Wert zugesandt. Dies ist gleichbedeutend damit, daß die Bekanntschaft von Anfang an bestand, die neue Nachricht jedoch alle vorherigen Nachrichten überholt hat. Sollte zu dem Zeitpunkt, wo die neue Nachricht beim Empfänger ankommt, eine "alte" Nachricht akzeptierbar sein, ist es die neue jedenfalls auch.

Damit ist nun auch der mehr oder weniger informelle Nachweis geführt, daß durch die angegebenen scripts der ggT berechnet wird, sofern das zugrundeliegende Agentennetz stark zusammenhängend ist. Diese Überlegungen beschließen das Beispiel; es sollte nicht nur einige **typische Probleme verteilter Berechnungen** aufzeigen, sondern vor allem dazu dienen, einige **CSSA-Sprachkonzepte** exemplarisch zu erläutern und die Nützlichkeit einiger **Möglichkeiten des CSSA-Systems** für die Programmentwicklung zu zeigen. Selbstverständlich konnten nicht alle Möglichkeiten des debuggers und des Simulationssystems zur Anwendung kommen (mehr davon in Kapitel 7 und 8), auch die Sprachkonzepte konnten nur in einer Auswahl gezeigt werden: So fehlen hier nicht nur eine Reihe von Datentypen und Kontrollstrukturen, sondern vor allem auch das Relationenkonzept, der Facettierungsmechanismus für statisch und dynamisch verschachtelte Facetten und die synchronisierte Kommunikation mit dem port-Konzept. Beispiele zu diesen Sprachkonstrukten finden sich in Teil D dieser Arbeit.

4.3 Vergleich mit anderen message-passing Konzepten

In diesem Kapitel, bei dem es sich um eine überarbeitete Fassung eines Seminarvortrages [SPE81] handelt, soll CSSA mit einigen anderen Sprachkonzepten verglichen werden, die ebenfalls auf dem "module-message"-Paradigma basieren:

- PLITS (Programming Language In The Sky [FEL79])
- CSP (Communicating Sequential Processes [HOA78])
- DP (Distributed Processes [BRI78])
- Ada ([DOD79], [DOD80])

Es werden lediglich die **Kommunikationsmechanismen** der einzelnen Sprachen miteinander verglichen; auf Modularisierungs- und Abstraktionskonzepte oder gar auf die rein sequentiellen Sprachkonzepte konnte nicht eingegangen werden.

4.3.1 Formen der Kommunikation über Nachrichten

Als Ausgangspunkt für alle Überlegungen kann eine beliebige sequentielle Programmiersprache dienen, die um einen SEND- und einen RECEIVE-Befehl erweitert wird, so daß Nachrichten zwischen verschiedenen Prozessen mit eigenständigen Kontrollflüssen übertragen werden können. Eine message ist dann zunächst lediglich eine Liste von Werten. Bei RECEIVE wird eine passende Liste von Variablen angegeben, an die die Werte gebunden werden:

P1	P2
.	.
.	.
SEND(5,TRUE) TO P2	RECEIVE(I,FLAG)
.	.
.	.

Meist wird aber einer message noch ein **Operationsbezeichner** (command identifier, entry name) vorangestellt, so daß messages ähnlich wie Prozeduraufrufe aussehen:

P1	P2
.	.
.	.
SEND INSERT(5,TRUE) TO P2	RECEIVE INSERT(I,FLAG)
.	.
.	.

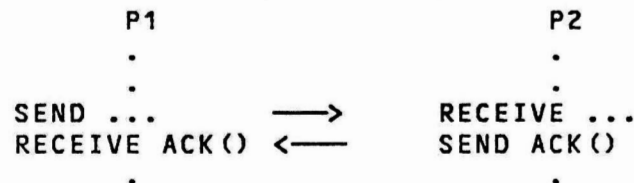
Die Prozesse P1 und P2 werden im allgemeinen nicht gleichzeitig das SEND- bzw. RECEIVE-Statement erreichen. Wird RECEIVE vor SEND erreicht, so wartet P2 sinnvollerweise. Wird aber SEND vor RECEIVE erreicht, so gibt es zwei Alternativen:

- P1 wartet, bis P2 RECEIVE erreicht hat (**synchrone** Kommunikation)

- P1 sendet die Nachricht ab und führt das nächste Statement aus. Die Nachricht wird gepuffert, bis P2 das RECEIVE erreicht hat (**asynchrone** Kommunikation, freilaufendes SEND)

Jede der Alternativen wird in einigen Konzepten gewählt. Tatsächlich kann man mit dem freilaufenden SEND die synchrone Kommunikation simulieren und umgekehrt:

- Synchronisation mit freilaufendem SEND kann erreicht werden durch Zurücksenden einer Quittung:



- Das freilaufende SEND kann durch synchronisierte Kommunikation simuliert werden, indem vor den Empfänger ein spezieller Pufferprozeß gesetzt wird, der alle Nachrichten sofort annimmt und erst später weiterreicht, wenn sie vom Empfänger angefordert werden.

Mit der Einführung von Operationsbezeichnern hat man bereits ein **selektives** RECEIVE, das nur bestimmte Nachrichten empfängt (nämlich diejenigen mit dem passenden Operationsbezeichner). Weitere Selektionsbedingungen sind üblich:

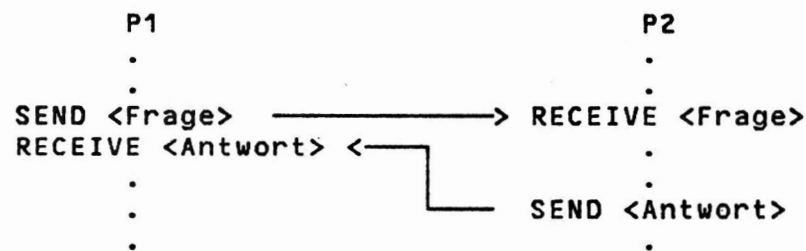
- Angabe der Quelle: RECEIVE ... FROM P1
- Passende Datentypen: RECEIVE INSERT(I) erwartet bspw., daß die Nachricht eine Integer-Zahl enthält.
- Boolesche Bedingungen an den lokalen Zustand des Empfängers
- Boolesche Bedingungen an den Inhalt der message

Mit der Möglichkeit, anzugeben, daß ganz bestimmte Nachrichten empfangen werden sollen, kommt man zwangsläufig auch zu Konstrukten, um alternativ verschiedene Nachrichten zu empfangen, etwa:.

```
RECEIVE INSERT(I) DO ...
OR
RECEIVE DELETE(I) DO ...
```

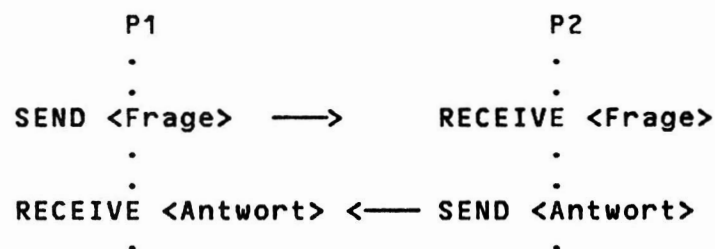
Es wird gewartet, bis entweder eine INSERT-Nachricht oder eine DELETE-Nachricht eintrifft.

Programmiert man mit den bisher eingeführten Konstrukten, so wird man bald feststellen, daß sich die versandten messages meist in zwei Klassen einteilen lassen: **Fragen** und **Antworten**. Ein typisches Kommunikationsprotokoll ist beispielsweise:

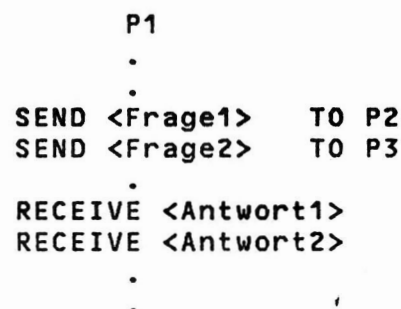


Auf Seiten von P2 steht dann üblicherweise ein alternatives RECEIVE, das verschiedene Fragen zuläßt. Da dieses Protokoll so häufig ist, werden dafür gern höhere Konstrukte eingeführt. Das Senden einer Frage und sofortiges Empfangen der Antwort wird zu einem CALL-Befehl mit IN- und OUT-Parametern zusammengefaßt. Das Verarbeiten einer Frage und Versenden einer Antwort wird zu einem prozedurähnlichen Block (Operation, Entry, Common Procedure) mit IN- und OUT-Parametern zusammengefaßt. Die Antwort wird dann automatisch an den rufenden Prozeß zurückgesandt. Anstelle des alternativen RECEIVE gibt es dann Sprachkonstrukte, die eine Reihe von Prozeduraufrufen von außen erlauben.

Mit solchen mächtigeren Konstrukten können dann einfache, gängige Probleme eleganter und kürzer programmiert werden. Bei komplizierteren Protokollen gibt es dann aber u.U. Schwierigkeiten und sehr umständliche Lösungen, wenn es nur noch die abstrakteren Konstrukte gibt. Häufig tritt nämlich auch der Fall ein, daß die Antwort erst nach einiger Zeit empfangen wird:



Sinnvoll ist auch, daß zwei Anfragen parallel bearbeitet werden:



Zwei Teilprobleme können so **gleichzeitig** gelöst werden. Gerade Probleme, die sich baumartig in Teilprobleme zerlegen lassen, bieten sich ja für Parallelarbeit an.

Mit mächtigeren **Sprachbausteinen** kann meist schneller und kürzer programmiert werden, mehr Flexibilität bieten aber kleinere Bausteine. Allerdings wird die geringere Flexibilität größerer Bausteine teilweise bewußt in Kauf genommen, da sie einen gewissen Programmierstil erzwingen und so erst die Voraussetzungen für **Korrektheitsbeweise** erfüllen.

In fast allen Anwendungsbeispielen läßt sich eine strenge Trennung zwischen **nur fragenden** und **nur antwortenden** Prozessen feststellen. Meist gibt es 'server', die gewisse Dienste zur Verfügung stellen, und 'user', die sich um diese Dienste bewerben und evtl. warten müssen, bis andere Benutzer abgefertigt sind. Aber nur CONCURRENT PASCAL (das hier nicht diskutiert wird, da es auf gemeinsamem Speicher basiert) trägt dieser Trennung explizit Rechnung durch zwei verschiedene Sprachkonstrukte. Dort gibt es nämlich **passive** Monitore (server) und **aktive** Prozesse (user). Ansatzweise erscheint die Trennung auch in Hansen's DP: Prozesse sind aktiv, bis sie terminieren, stellen dann aber noch ihre common procedures zur Verfügung. In allen Beispielen in [BRI78] terminieren die Prozesse sofort oder nie, d.h. sind **nur** server oder **nur** user.

4.3.2 Das Philosophenproblem in message-passing Systemen

Um die Unterschiede zwischen den verschiedenen Konzepten zu verdeutlichen, soll eine für message-passing Konzepte geeignete, dezentralisierte Lösung des Philosophenproblems angegeben werden und mit allen vorgestellten Sprachen programmiert werden.

Die fünf Philosophen und die fünf Gabeln sind hier parallel arbeitende Prozesse, die nur durch Nachrichten kommunizieren. Es gibt keine globale Instanz wie einen Tisch oder ein Haus, die einen Engpass bilden könnte. Die Lösung erfüllt folgende Forderungen:

- ◊ Keine zwei Nachbarn essen jemals gleichzeitig
- ◊ Ein Philosoph kann nach endlicher Zeit essen, wenn er hungrig wird.

Es gilt aber nicht:

- ◊ Wenn keiner seiner Nachbarn ißt, so kann ein Philosoph sofort essen.

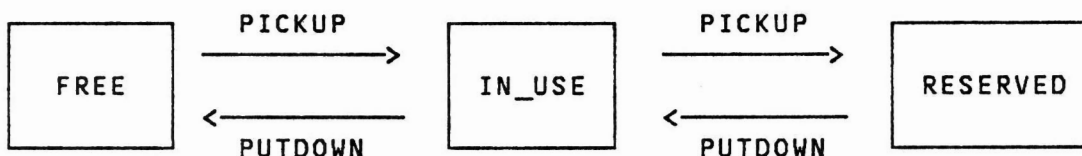
Die Philosophen wiederholen zyklisch folgende Aktionen:

1. Denken
2. Linke Gabel anfordern (durch SEND PICKUP)
3. Rechte Gabel anfordern (durch SEND PICKUP)
4. Essen
5. Linke Gabel hinlegen (durch SEND PUTDOWN)
6. Rechte Gabel hinlegen (durch SEND PUTDOWN)

Man beachte, daß beim Anfordern einer Gabel gewartet werden muß, bis eine Bestätigungsnachricht von der Gabel eingetroffen ist. Für das Anfordern eignet sich also eine Form der **synchronisierten** Kommunikation. Beim Hinlegen einer Gabel braucht jedoch nicht auf eine Bestätigung gewartet zu werden. Der Philosoph kann sofort mit dem Denken anfangen. Hier ist also die **nicht-synchronisierte** Form der Kommunikation (eine freilaufende message) angebracht.

Eine **Verklemmung** wird dadurch verhindert, daß die Gabeln (Betriebsmittel) **linear geordnet** (durchnummeriert) werden, und jeder Philosoph stets die Gabel mit der kleineren Nummer zuerst anfordert.

Die Gabel hat drei verschiedene mögliche Zustände: FREE, IN_USE, RESERVED. Die Nachrichten, die von den Philosophen gesendet werden, bewirken folgende Zustandsübergänge der Gabel:



Im Zustand RESERVED wird die Gabel noch benutzt, aber der zweite Philosoph hat sich bereits vorangemeldet und bekommt sie zugeteilt, sobald sie nicht mehr gebraucht wird. Dadurch wird verhindert, daß ein Philosoph seine Gabel hinlegt und sofort wieder aufnimmt, obwohl ein anderer Philosoph wartet.

Man beachte, daß die Gabel die Bestätigung für das 'zweite' PICKUP erst geben darf, wenn wieder ein PUTDOWN ausgeführt wurde. Die Gabel muß daher in irgendeiner Form die Identität des wartenden Philosophen speichern, bis PUTDOWN ausgeführt wurde.

In einigen Sprachkonzepten ist allerdings bereits eine **implizite** Form der Voranmeldung vorgesehen: Beispielsweise werden in Ada die vorliegenden Nachrichten grundsätzlich nach dem FIFO-Prinzip abgearbeitet, so daß ein bereits wartender Philosoph nicht übervorteilt werden kann. Der dritte Zustand der Gabel ist in diesem Falle überflüssig. Auch die CSSA-Lösung verwendet nur zwei Zustände.

Weitere Bemerkungen zum Philosophenproblem, sowie ein ausführliches Protokoll der CSSA-Lösung finden sich in Teil D dieser Arbeit.

4.3.3 Vergleichskriterien für die verschiedenen Konzepte

In diesem Kapitel sollen noch einige Kriterien diskutiert werden, anhand derer die verschiedenen Konzepte verglichen werden können.

◊ **Kontrollfluß**

In allen Konzepten arbeiten mehrere Prozesse parallel, d.h. es gibt **mehrere Kontrollflüsse**. Es ist interessant zu ver-

gleichen, wie die Kontrollflüsse entstehen und wieder verschwinden.

Die einfachste Form ist, daß zu Anfang alle Prozesse **simultan** gestartet werden und so die parallelen Kontrollflüsse entstehen. Bei Sprachen, die ein explizites Starten und Terminieren von Prozessen zulassen, können neue Kontrollflüsse hinzukommen und wieder verschwinden. Teilweise gibt es auch spezielle Parallelkommandos, die die parallele Ausführung von einigen Statements spezifizieren. Hier spaltet sich ein Kontrollfluß auf und verschmilzt wieder ('fork/join').

◦ **Nichtdeterminismus**

Da die relativen Ausführungszeiten von Prozessen auf verschiedenen Prozessoren ungewiß und die Übertragungszeiten für Nachrichten nicht vorhersagbar sind, verhalten sich Systeme paralleler Prozesse im allgemeinen **nicht-deterministisch**. Dementsprechend führen einige Konzepte auch nicht-deterministische Konstrukte für die **sequentielle** Programmierung ein: Beispielsweise das nicht-deterministische Einschlagen einer CASE-Alternative oder die zufällige Auswahl der nächsten passenden Nachricht aus dem Puffer. Es gibt Konzepte (z.B. CSSA), die ausdrücklich darauf basieren, daß die nicht-deterministische Auswahl **fair** ist, d.h. daß mögliche Alternativen nicht unendlich oft ausgelassen werden; andere sehen Programme, die nur bei fairer Auswahl richtig laufen, als **inkorrekt** an.

◦ **Das Adressierungsproblem**

In einem SEND-Befehl muß der Prozeß **benannt** werden, der die Nachricht empfangen soll. Die einfachste Möglichkeit besteht darin, daß jeder Prozeß einen systemglobalen, eindeutigen Namen bekommt, den alle anderen Prozesse kennen und verwenden können.

Oft soll es aber eine ganze Reihe von Prozessen geben, die alle nach dem **gleichen Algorithmus** arbeiten. Man führt daher **indizierte** Prozeßnamen ein.

```
PROCESS PHILO(1..5)      PROCESS FORK(1..5)
      .
      .
      .
SEND PICKUP TO FORK(3)    RECEIVE PICKUP ...
```

Es gibt also fünf Philosophen und fünf Gabeln. Im Rumpf eines Prozesses kann üblicherweise auf den eigenen Index zugegriffen werden (z.B. durch THIS). Kann der Index in einem SEND-Statement durch einen beliebigen Ausdruck gegeben sein, so kann der adressierte Prozeß nicht mehr **statisch** anhand des Programmtextes entschieden werden, sondern erst zur Laufzeit. Wenn eine solche flexible Adressierung nicht unterstützt wird, so dürfen im Index-Ausdruck nur Konstanten und THIS verwendet werden.

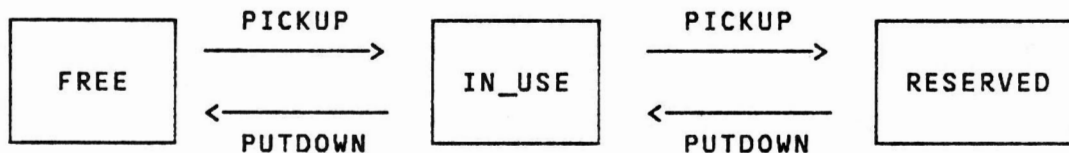
Die wohl flexibelste Form der Adressierung besteht darin, daß Prozeßdefinitionen zunächst nur einen **Typ** festlegen, von dem beliebig viele Instanzen geschaffen werden können. Die Adressierung geschieht dann über Variablen, die als Wert einen Verweis auf eine Prozeßinstanz haben. Können solche Verweise auch in Nachrichten verschickt werden, so

bilden die Prozesse mit ihren Bekanntschaften einen **dynamischen**, gerichteten Graphen (in CSSA: "Agentennetz"), in dem Knoten und Kanten entstehen und verschwinden können. Ein gewisses Problem stellen dann die Kanten ohne Endknoten dar (Verweise auf terminierte Prozesse).

◦ **Darstellung von path-expressions**

Server-Prozesse können oft als **abstrakte Datentypen** aufgefasst werden, die bestimmte Operationen zur Verfügung stellen (bereit sind, bestimmte messages zu verarbeiten). Die Menge der zur Verfügung gestellten Operationen kann sich dabei nach dem Zustand richten.

Die Gabel aus dem Philosophenproblem kann bspw. als ein solcher abstrakter Datentyp aufgefaßt werden:



- Im Zustand FREE wird nur die Operation PICKUP erlaubt.

- Im Zustand IN_USE werden PICKUP und PUTDOWN erlaubt.

- Im Zustand RESERVED wird nur PUTDOWN erlaubt.

Die Gabel akzeptiert gewissermaßen eine Sprache über dem Alphabet {PICKUP,PUTDOWN}. Wenn möglich, stellt man solche Sprachen durch reguläre Ausdrücke, die sogenannten path-expressions, dar. Die von der Gabel akzeptierte Sprache läßt sich schreiben als:

$(\text{PICKUP} (\text{PICKUP PUTDOWN})^* \text{PUTDOWN})^*$

Es ist recht aufschlußreich zu vergleichen, welche Möglichkeiten die verschiedenen Konzepte bieten, um path-expressions zu realisieren. Besonders wichtig sind dabei die Konstrukte, die das Empfangen nur bestimmter messages spezifizieren.

◦ **Implementierung von Prioritätsregelungen**

Es sind leicht Anwendungen denkbar, bei denen ein server die an ihn gestellten Anfragen nicht in der Reihenfolge ihres Eingangs beantwortet, sondern in einer beliebigen anderen Reihenfolge (vgl. Verwaltungsprozesse für Betriebsmittel). Dazu müssen beispielsweise die messages wahlfrei aus dem Puffer entnommen werden können; oder alle eine Anfrage betreffenden Informationen (gewünschte Operation, Parameter, fragender Prozeß) müssen im server gespeichert werden können. Ada z.B. läßt zunächst nur FIFO-Bearbeitung der Anfragen zu, und erlaubt es auch nicht, die Identität eines wartenden Prozesses zwischenspeichern. In DP gibt es keine Möglichkeit, einen Aufruf von außen überhaupt zu verhindern. Dennoch können in allen Konzepten Prioritätsregelungen implementiert werden – wenn auch teilweise recht umständlich.

Im folgenden werden die verschiedenen Konzepte kurz dargestellt. Natürlich können in den Kurzbeschreibungen nur die **wichtigsten** Sprachkonstrukte beschrieben werden. Es wird dann jeweils die oben angegebene Lösung des Philosophenproblems programmiert, und die hier vorgestellten Vergleichskriterien werden

untersucht.

4.3.4 PLITS

PLITS geht davon aus, daß auf mehreren parallel arbeitenden Prozessoren (zunächst) unabhängig voneinander sogenannte **Module** laufen. Hier wird davon ausgegangen, daß alle Module in PASCAL geschrieben sind. PASCAL soll nun so erweitert werden, daß die verschiedenen Module kommunizieren können. Es werden dazu Variablen vom Typ MODULE eingeführt, die als Wert die Adresse eines Moduls haben können.

Beispiel: **var** NEIGHBOUR: MODULE;

Die Module selbst werden als **Typen** deklariert:

type PHILO = **mod** (LFORK,RFORK:MODULE) **begin ... end**

Exemplare (Instanzen) werden wie records erzeugt:

P := new PHILO(...,...);

Als zweiter neuer Datentyp wird in PLITS nun die MESSAGE eingeführt. Eine message ist eine Liste von name:value Paaren, ähnlich der property-Listen in LISP.

var M: MESSAGE;
M := (NAME: "Schmitz", VORNAME: "Jupp", ALTER: 28);

In eine message können neue Paare durch PUT eingetragen werden oder durch REMOVE entfernt werden. Typ und Anzahl der Felder ist nicht fest wie bei records. Auf einzelne Felder wird mit Punktnotation zugegriffen:

M.NAME := "Schmitz"

Messages können an andere Module verschickt werden durch

send <message> **to** <module>

SEND verzögert den Sender nicht. Als Gegenstück zum SEND-Statement gibt es ein RECEIVE-Statement.

receive <message-var>

Es wird gewartet, bis irgendeine message ankommt, die dann der angegebenen Variablen zugewiesen wird. Will man unnötiges Warten vermeiden, so kann man das Standard-Prädikat 'PENDING' verwenden, das angibt, ob eine message vorliegt.

Das Philosophenproblem in PLITS

```

type PHILO= mod (LFORK,RFORK:MODULE)
var ACK:MESSAGE;
while true do
begin (* DENKEN *)
  send (REQUEST: PICKUP, CALLER: this ) to LFORK;
  receive ACK;
  send (REQUEST: PICKUP, CALLER: this ) to RFORK;
  receive ACK;
  (* ESSEN *)
  send (REQUEST: PUTDOWN) to LFORK;
  send (REQUEST: PUTDOWN) to RFORK;
end;

type FORK= mod
var WAITING_PHILO:MODULE;
  MESS      :MESSAGE;
  STATE     : (FREE,IN_USE,RESERVED);
while true do
begin receive MESS;
  case STATE of
    FREE:      (* request=pickup *)
                STATE:=IN_USE;
                send () to MESS.CALLER;
    IN_USE:    if MESS.REQUEST=PICKUP
                then begin STATE:=RESERVED;
                           WAITING_PHILO:=MESS.CALLER;
                           end
                else (* request=putdown *) STATE:=FREE;
    RESERVED: (* request=putdown *);
                STATE:=IN_USE;
                send () to WAITING_PHILO;
  end;
end;

```

- Die Gabel empfängt jede message und reagiert je nach Wert von REQUEST. Das ist das Grundprinzip von PLITS: Sollen Nachrichten nicht sofort verarbeitet werden, so können sie im Modul gespeichert werden. PLITS enthält allerdings auch einige Formen des selektiven RECEIVE.
- Da keine Annahme darüber gemacht wird, in welcher Reihenfolge Nachrichten abgearbeitet werden, ist eine explizite Voranmeldung der Philosophen nötig.
- Ein Schutz gegen unsinnige messages ist nicht implementiert.
- Die leere message wird als Signal verwendet.
- Beim Erzeugen der Philosophen müssen die beiden Gabeln so übergeben werden, daß die linke die kleinere Nummer hat.

Es ist gelungen, mit einer minimalen Anzahl neuer Konstrukte alle denkbaren Formen von message-passing programmierbar zu machen: Messages sind voll in die Sprache integrierte Datenstrukturen, so daß sie beliebig abgespeichert, inspiziert, gelöscht

oder weitergesandt werden können. SEND und RECEIVE können von beliebigen Kontrollstrukturen umgeben sein. Durch die recht primitiven Konstrukte werden PLITS-Lösungen teilweise etwas aufwendig, andererseits bieten die Konstrukte ein hohes Maß an Flexibilität.

4.3.5 CSP

CSP (von C.A.R Hoare) ist keine vollständige Programmiersprache, sondern nur ein Sprachfragment, ein Vorschlag für gewisse Kommunikationskonzepte. Hoare betont, daß er bewußt eine stark eingeschränkte Sprache entworfen hat, um den Blick für die wesentlichen Konzepte freizumachen.

Die Parallelität wird in ein zunächst sequentielles Programm durch ein **parallel-command** eingeführt:

```
[p1:: <statements> || ... || pn:: <statements>]
```

Der Kontrollfluß spaltet sich auf; es sind n Prozesse gleichzeitig aktiv. Das **parallel-command** ist beendet, sobald **alle** parallelen Prozesse beendet sind.

Möglich sind auch **indizierte** Prozeßnamen, falls mehrere **gleichartige** Prozesse parallel ablaufen sollen:

```
[fork(i:0..4)::FORK || phil(i:0..4)::PHIL]
```

FORK und PHIL stehen für Statementfolgen die noch definiert werden und in denen i benutzt werden kann (liefert den jeweiligen Index). In jedem Fall steht die Maximalzahl von Prozessen in einem CSP-Programm **statisch** fest.

Kommunikation zwischen Prozessen ist möglich, indem ein Prozeß ein **output-command** (Syntax: <process-name>!<expr>) ausführt, und ein anderer ein korrespondierendes **input-command** (Syntax: <process-name>?<variable>):

```

p1                                p2
.                                .
.                                .
p2!5                              p1?value
.                                .

```

Die beide Prozesse müssen sich **gegenseitig** benennen und auf einander warten. Eine Pufferung von Nachrichten gibt es nicht.

In CSP gibt es auch **benannte** messages:

```

p1                                p2
.                                .
.                                .
p1!insert(5)                      p1?insert(value)
.                                .

```


Durch ein **alternative command** kann ein Prozeß eine ganze Reihe verschiedener inputs alternativ zulassen und jeweils anders abarbeiten:

```
[p1?pickup() --> .... ◊ ... ◊ p2?putdown() --> ....]
```

Die leere message wird hier wieder als **Signal** verwendet. Es wird gewartet, bis p1 oder p2 ein passendes output-command ausführen und die entsprechende Alternative ausgeführt. Haben sowohl p1 als auch p2 bereits ein passendes output-command ausgeführt, so erfolgt die Auswahl **nicht-deterministisch**.

Die einzelnen Alternativen können auch noch durch sogenannte **guards** - das sind boolesche Ausdrücke - geschützt werden. Das input-command kann dann auch ganz fehlen:

```
[x=0;p1?y --> ... ◊ x>0;p2?y --> ... ◊ x<0 --> ...]
```

Gewartet wird hier nur falls $x \geq 0$.

Ein alternative-command ist nach Ausführung einer Alternative beendet. Wiederholte Ausführung wird durch einen vorgestellten '*' markiert. Die Ausführung eines solchen repetetive-command wird beendet, sobald **alle** guards FALSE liefern.

```
*[i<length;p1?x --> i:=i+1;sum:=sum+x]
```

Das repetetive-command läßt sich mit den CSSA-Facetten vergleichen; die guards entsprechen in etwa den assertions von CSSA. Eine Alternative ohne input-command läßt sich ähnlich wie eine idle-Operation verwenden.

Das Philosophenproblem in CSP

```

phil(i:0..4)::PHIL || fork(i:0..4)::FORK]

PHIL=  left,right:INTEGER;
      (* Berechnung von left und right aus i *)
      *[ TRUE --> (* DENKEN *)
          fork(left)!pickup();
          fork(left)?eat();
          fork(right)!pickup();
          fork(right)?eat();
          (* ESSEN *)
          fork(left)!putdown();
          fork(right)!putdown()
        ]

FORK=  left,right,user,other,u:INTEGER; in_use: BOOLEAN;
      (* Berechnung von left und right aus i *)
      (* ==> free <== *)
      *[ phil(left)?pickup()
        --> in_use:=TRUE;user:=left;other:=right;
            phil(left)!eat();
            (* ==> in_use <== *)
            *[ in_use;phil(user)?putdown()
              --> in_use:=FALSE
              ◊ in_use;phil(other)?pickup()
              --> (* ==> reserved <== *)
                  phil(user)?putdown();
                  phil(other)!eat();
                  u:=user;user:=other;other:=u
            ]
        ◊ phil(right)?pickup()
        --> in_use:=TRUE;user:=right;other:=left;
            <wie oben>
      ]

◊ Bei PUTDOWN muß der Philosoph warten, da es kein freilau-
  fendes SEND gibt.

◊ Die Gabel speichert den Index des wartenden Philosophen.
  Soll ein server Verweise auf Benutzer unterschiedlichen
  Typs speichern, so ist der Index allein nicht mehr ausrei-
  chend und es gibt erhebliche Schwierigkeiten.

◊ Für jeden der beiden Philosophen gibt es im äußersten
  repetitive-command der Gabel eine eigene Alternative. Die
  Verpflichtung, die Quelle einer message zu benennen, führt
  hier zu recht umständlicher Programmierung. Sinnvollerweise
  sollte zwar ein user den server kennen, aber nicht ein
  server alle seine user.

```

Im Vergleich zu PLITS verfügt CSP über bessere Möglichkeiten zu spezifizieren, welche messages jeweils entgegengenommen werden sollen (alternative-command, guards). Path-expressions können daher einfacher implementiert werden. Die niedrige Abstraktionsebene (input - output) ermöglicht es ohne weiteres, mehrere Anfragen parallel abarbeiten zu lassen: Da Fragen und Antworten

explizit versandt werden, können sie auch an beliebiger Stelle versandt werden. Durch die fehlende Pufferung von messages geht allerdings Parallelität verloren.

Die Implementation von **Prioritätsregelungen** ist solange einfach lösbar, wie es nur eine einzige Klasse von Benutzern gibt. Denn dann können alle Informationen über eine Anfrage abgespeichert werden, bis sie beantwortet ist.

Der Hauptvorwurf, der CSP gemacht wird, ist, daß die Quelle einer Nachricht explizit benannt werden muß.

4.3.6 CSSA

An dieser Stelle soll die CSSA-Lösung des Philosophenproblems angegeben werden, da CSSA vom Abstraktionsgrad seiner Sprachbausteine her etwa in der Mitte liegt. Weitere Bemerkungen zu diesem CSSA-Programm, sowie ein detailliertes Ausführungsprotokoll finden sich in Teil D dieser Arbeit.

RAUM FÜR NOTIZEN:

Das Philosophenproblem in CSSA

```

type PHILO is
script ( agent: LFORK,RFORK)
  facet ALIVE is
    operation idle is
      oper: PICKUP,PUTDOWN;
      port: FIRST_FORK,SECOND_FORK;

      (* DENKEN *)
      send PICKUP to LFORK reply to FIRST_FORK;
      wait FIRST_FORK;
      send PICKUP to RFORK reply to SECOND_FORK;
      wait SECOND_FORK;
      (* ESSEN *)
      send PUTDOWN to LFORK;
      send PUTDOWN to RFORK;
    endoperation
  endfacet
  initial ALIVE
endscript

type FORK is
script
  facethead IN_USE
  facet FREE is
    public: PICKUP;
    operation PICKUP is
      reply;
      replace by IN_USE;
    endoperation
  endfacet
  facet IN_USE is
    public: PUTDOWN;
    operation PUTDOWN is
      replace by FREE;
    endoperation
  endfacet
  initial FREE
endscript

```

- Bei dieser Lösung sieht man eine sehr direkte Analogie zum Automatenmodell
- Für PUTDOWN wird das freilaufende SEND verwendet.
- Da bei der Auswahl der Nachrichten aus der mailbox **Fairness** garantiert ist, kann eine explizite Voranmeldung des zweiten Philosophen entfallen: Die PICKUP-Nachricht kann nicht unendlich oft ignoriert werden, das Verhungern eines Philosophen ist daher ausgeschlossen. Eine Facette RESERVED ist nicht notwendig.

Abstrakte Datentypen und path-expressions können in CSSA durch den Facettierungsmechanismus sehr direkt abgebildet werden. CSSA-Facetten entsprechen in etwa den repetitive-commands von CSP. Allerdings bietet das bereits recht abstrakte Sprach-

konstrukt **OPERATION** bessere Möglichkeiten zur Strukturierung von Agenten. Dennoch kann ein Agent problemlos mehrere Anfragen gleichzeitig laufen haben oder die Wartezeit bis zur Antwort nützen, denn **SEND** (fragen) und **WAIT** (Antwort erwarten) können getrennt werden. Dadurch, daß Antworten automatisch an den richtigen Agenten und den richtigen port geschickt werden, steigt die Unabhängigkeit der Agenten: Der server braucht seine user nicht zu kennen.

Zur Implementierung von Prioritätsregelungen gibt es eine Reihe von Hilfen: Durch Facettierung und assertions kann die Reihenfolge der Entnahme von Nachrichten aus der mailbox gesteuert werden. Schließlich kann ein Verweis auf den wartenden Agenten **explizit** (d.h. in normalen Variablen) gespeichert werden, so daß Anfragen zu einem beliebigen Zeitpunkt beantwortet werden können.

4.3.7 DP

Ein DP Programm besteht aus einer **festen** Menge von nebenläufigen Prozessen. Alle n Prozesse werden zu Beginn gleichzeitig gestartet, und hören niemals auf zu existieren. Dadurch entstehen n Kontrollflüsse. Die Anzahl der Kontrollflüsse kann niemals größer werden, nur kleiner (s.u.).

```

process <name>
  <local variables>
  <common procedures>
  <initial statement>
end

```

Ein Prozeß kann nur auf seine eigenen Variablen zugreifen. Er stellt aber eine Reihe von **common-procedures** zur Verfügung, die von anderen Prozessen aufgerufen werden können.

```

<common procedure> ::=
  proc <name> (<in-parms> # <out-parms>)
    <local variables>
    <statement>
  end

```

Das Aufrufen einer common procedure eines anderen Prozesses ist die einzige Möglichkeit zu kommunizieren.

```

call <process> . <procedure> (<parms>)

```

Beim Aufruf werden die IN-Parameter in einer message an den gerufenen Prozeß übergeben. Der rufende Prozeß wartet, bis eine message mit den OUT-Parametern ankommt.

Die gerufene Prozedur wird nicht unbedingt **sofort** ausgeführt, sondern erst wenn der gerufene Prozeß bereit ist, externe Aufrufe zu verarbeiten. Der gerufene Prozeß ist dann dazu bereit, wenn er ein **WHEN**-Statement erreicht hat, das nicht sofort ausgeführt werden kann.

```

when B1 : S2 ... Bn : Sn end

```

Das WHEN-Statement entspricht in etwa einem CASE-Statement: Wenn die Bedingung B_i erfüllt ist, wird das Statement S_i ausgeführt. Sind aber **mehrere** Bedingungen erfüllt, so erfolgt die Auswahl **nicht-deterministisch**. Ist **keine** der Bedingungen wahr, so heißt das WHEN-Statement '**blockiert**'. Der Prozeß geht in einen Wartezustand und ist bereit **beliebige** externe Aufrufe zu verarbeiten. (Das Warten in einem CALL-Statement bedeutet hingegen **nicht**, daß eine common procedure aufgerufen werden kann. Die Wartezeit auf die Antwort kann nicht genutzt werden.) WHEN-Statements können im initial-Statement oder in einer common procedure stehen:

```

process P1                                process P2
  <local variables>                        .
  proc X(.....)                            .
    .                                      .
    when ... end                            .
    .                                      .
  end                                        call P1.X(...);
    .                                      .
  when ... end                              .
    .                                      .
end                                          end

```

Der Prozeß P1 beginnt nach seiner Erzeugung mit der Ausführung des initial-Statements. Führt ihn dann die Ausführung des WHEN-Statements in einen Wartezustand, so kann (so bald ein Aufruf vorliegt) die common procedure X ausgeführt werden. Nach Beendigung von X oder wenn in X wieder ein blockiertes WHEN erreicht wird, kann das initial-Statement fortgeführt werden (sofern jetzt eine Bedingung wahr geworden ist), oder es kann ein weiterer Aufruf einer common procedure ausgeführt werden.

Es kann also beliebig viele '**angebrochene**' common procedures geben. Jeweils bei Erreichen eines blockierten WHEN-Statements oder bei Beendigung einer common procedure wird ein nicht mehr blockiertes WHEN-Statement fortgesetzt, eine neue procedure gestartet oder (wenn beides nicht möglich ist) gewartet, bis ein Aufruf eintrifft. Sind mehrere Alternativen möglich, so ist die Auswahl **nicht-deterministisch**.

Nach Beendigung des initial-Statements hat der Prozeß keinen eigenen Kontrollfluß mehr, sondern akzeptiert nur noch externe Aufrufe.

Das Philosophenproblem in DP

```

process PHILO(5)
  LEFT,RIGHT:INTEGER;
  (* Berechnung von LEFT und RIGHT aus THIS *)
  while true do
    begin (* DENKEN *)
      call FORK(LEFT).PICKUP;
      call FORK(RIGHT).PICKUP;
      (* ESSEN *)
      call FORK(LEFT).PUTDOWN;
      call FORK(RIGHT).PUTDOWN;
    end;
  end;

process FORK(5)
  STATE:(FREE,IN_USE,RESERVED,ASSIGNED);
  proc PICKUP
    when STATE/=ASSIGNED: skip end;
    case STATE of
      FREE:   STATE:=IN_USE;
      IN_USE: STATE:=RESERVED;
      when STATE=ASSIGNED : STATE:=IN_USE end
    end;
  proc PUTDOWN
    case STATE of
      IN_USE:   STATE:=FREE;
      RESERVED: STATE:=ASSIGNED;
    end;
  STATE:=FREE
end

```

- Bei PUTDOWN wartet der Philosoph, da es keine asynchrone Kommunikation in DP gibt.
- Das initial-Statement des Philosophen wird niemals beendet: Er ist ein user.
- Das initial-Statement der Gabel ist sofort beendet: Sie ist ein server.
- SKIP ist das leere Statement.
- Es kann während der Lebensdauer einer Instanz von PICKUP PUTDOWN aufgerufen werden. So wird die Identität des Philosophen, der auf die Gabel wartet, gespeichert, bis sie ihm zugeteilt wird. Die Ausführung von PICKUP **umfaßt** das Warten auf PUTDOWN.
- Bei der Freigabe der Gabel durch den ersten Philosophen kann nicht sofort die Antwortnachricht an den zweiten (wartenden) Philosophen versandt werden, sondern es wird lediglich durch eine **Zustandsänderung** die Fortführung der **blockierten** PICKUP-Prozedur **ermöglicht**. Da aber keine Annahme darüber gemacht werden kann, wieviel Zeit bis dahin tatsächlich noch vergeht, besteht die Gefahr, daß der erste Philosoph bereits wieder PICKUP aufruft und bei der nicht-

deterministischen Auswahl bevorzugt wird. Aus diesem Grund wurde ein spezieller Zustand (ASSIGNED) eingeführt, in dem der bereits wartende Philosoph Priorität hat.

In DP gibt es keine Möglichkeit, einen Aufruf von außen zu verhindern. Denn es wird nicht auf bestimmte messages gewartet, sondern auf eine **Zustandsänderung**. Die Ausführung einer common-procedure kann aber durch ein WHEN-Statement sofort gestoppt werden. Es kann daher beliebig viele angebrochene common-procedures geben. Da außerdem die Weiterführung der angebrochenen Prozeduren in beliebiger Reihenfolge geschehen kann, reicht für die Prozedurinstanzen ein **stack** nicht aus. Daß es zu jedem Zeitpunkt eine ganze Reihe von Punkten im Programmtext geben kann, an denen die Ausführung fortgesetzt werden kann, erschwert das Nachvollziehen des Programmablaufs sehr: Man kann nicht mehr mit dem Finger auf die Stelle zeigen 'wo man ist'.

Die Abarbeitung von requests in beliebiger Reihenfolge ist in DP sehr einfach:

```
proc REQUEST(WHO:INTEGER)
  QUEUE.INCLUDE(WHO);
  when USER=WHO : ..... end
end
```

Jeder user wird in die Datenstruktur QUEUE aufgenommen. Der request wird solange verzögert bis der server den user ausgewählt hat. Die Identität des wartenden user-Prozesses wird **implizit** in der angebrochenen Prozedurinstanz gespeichert.

Das gleichzeitige Abarbeiten zweier Aufträge ist nicht ohne weiteres möglich, da der CALL-Befehl immer sofort auf Antwort wartet. Wie es dennoch geht, soll hier kurz skizziert werden:

```
call F(5).FIB(I-1);
call F(6).FIB(I-2);
call F(5).GIVE_REPLY(VALUE1);
call F(6).GIVE_REPLY(VALUE2);
```

Der Aufruf von FIB bewirkt lediglich die Übergabe der Parameter. Die eigentliche Berechnung machen F(5) und F(6) erst nach Beendigung der common-procedure FIB. Da es keine Möglichkeit gibt, den aufrufenden Prozeß zu speichern, muß dieser wiederum selbst nach dem Ergebnis fragen.

4.3.8 Ada

Ein Ada-Programm kann aus mehreren **tasks** bestehen. Im allgemeinen ist die Kommunikation zwischen tasks über gemeinsame Variablen möglich. Für Systeme mit verteiltem Speicher sind gemeinsame Variablen aber ausdrücklich verboten, und task-Kommunikation ist nur über sogenannte **Rendezvous** möglich.

Im sichtbaren Teil einer task können sogenannte **entries** spezifiziert werden:

```
task PRODUCER_CONSUMER is
    entry READ (V: out INTEGER);
    entry WRITE(E: in INTEGER);
end;
```

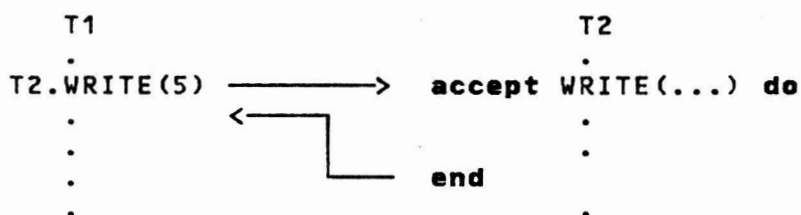
Der BODY einer task wird in Ada getrennt notiert.

Die entry-Deklarationen sehen aus wie Prozedurköpfe. Tatsächlich entspricht ein entry auch weitgehend einer Prozedur, die von einer anderen task aufgerufen werden kann; beispielsweise durch 'PRODUCER_CONSUMER.WRITE(5)'. Ein Aufruf von einer anderen task aus wird aber nicht unbedingt **sofort** ausgeführt; die rufende task muß warten, bis die gerufene task einen Zustand erreicht hat, in dem sie bereit ist, den Aufruf zu akzeptieren und zu bearbeiten. Eine task ist dann bereit einen Aufruf entgegenzunehmen, wenn sie ein ACCEPT-Statement erreicht hat:

```
accept WRITE(E: in INTEGER) do
    <statements>
end
```

Eine der beiden tasks wartet auf die andere, bis sowohl der ENTRY-Aufruf als auch das ACCEPT-Statement erreicht sind. Man nennt diese Form der Kommunikation ein **Rendezvous**. Während des Rendezvous haben beide tasks einen gemeinsamen Kontrollfluß. Nach Erreichen des END des ACCEPT-Statements trennen sie sich wieder und arbeiten unabhängig weiter. Liegen mehrere Aufrufe verschiedener tasks für den gleichen entry vor, so werden sie streng nach dem FIFO-Prinzip abgearbeitet.

In verteilten Systemen wird man Rendezvous durch message-passing implementieren. Beim Aufruf versendet die rufende task eine message mit entry-name und Parametern. Anschliessend geht sie in einen Wartezustand. Bei dergerufenen task gibt es pro entry eine FIFO-queue, in die die ankommenden messages geschrieben werden. Die task entnimmt eine message und führt den ACCEPT-body aus. Bei END wird eine Antwort-message zurückgesandt, die die OUT-Parameter enthält und die die rufende task wieder aufweckt.



Ist der ACCEPT-body leer, so entspricht das dem Versenden einer Nachricht mit Quittung.

Die Möglichkeit, gleichzeitig auf mehrere alternative Aufrufe zu warten, also mehrere ACCEPT-Statements zusammenzufassen, bietet das SELECT-Statement:

```
select accept WRITE (...) do
    <statements>
end
    <statements>
or accept READ (...) do
    <statements>
end
    <statements>
end
```

Es wird erwartet, bis entweder WRITE oder READ aufgerufen wird, danach ist das SELECT-Statement beendet. Da in einer SELECT-Alternative auch hinter dem ACCEPT-body noch Statements stehen können, kann das END also an beliebiger Stelle in der SELECT-Alternative stehen. So wird gesteuert, wann die Parameter zurückgegeben werden und die rufende task geweckt wird.

Die verschiedenen SELECT-Alternativen können auch noch durch Boolesche Bedingungen geschützt werden:

```
select when not FULL => accept WRITE (...) do
    <statements>
end
    <statements>
or when not EMPTY => accept READ (...) do
    <statements>
end
    <statements>
end
```

Bemerkung: Legt man eine Schleife um ein SELECT-Statement, so erhält man so etwas wie eine CSSA-Facette. Die SELECT-Alternativen entsprechen den CSSA-Operationen. Das END des ACCEPT-body entspricht dem REPLY-Statement von CSSA. Das WHEN-Konstrukt ähnelt den assertions von CSSA. Allerdings kann hinter WHEN nicht auf die in der message stehenden Parameter Bezug genommen werden. Dem Facettierungsmechanismus von CSSA entsprechende Konstruktionen können in Ada mit herkömmlichen Kontrollstrukturen programmiert werden. Je nach Facette kann eine CSSA-Operation anders definiert sein; ebenso kann es in Ada zu einem entry mehrere ACCEPT-Statements geben. Während aber in CSSA eine Operation, die in mehreren Facetten unverändert erlaubt ist, nur einmal notiert zu werden braucht, muß in Ada jeweils ein komplettes ACCEPT-Statement notiert werden (in dem allerdings eine herkömmliche Prozedur aufgerufen werden kann). Es sind auch **verschachtelte** ACCEPT-Statements erlaubt, so daß zwei Rendezvous in einer anderen Reihenfolge beendet werden können, als sie begonnen wurden. Verschachtelte Operationen hingegen sind in CSSA ausdrücklich verboten: Die **Unteilbarkeit** von Operationen ist damit gewährleistet.

Das Philosophenproblem in Ada

```
task PHILO(1..5);

task body PHILO is
  LEFT, RIGHT: INTEGER;
  begin -- Berechnung von LEFT und RIGHT aus dem eigenen Index
    loop -- DENKEN
      FORK(LEFT).PICKUP;
      FORK(RIGHT).PICKUP;
      -- ESSEN
      FORK(LEFT).PUTDOWN;
      FORK(RIGHT).PUTDOWN;
    end loop;
  end;

task FORK(1..5) is
  entry PICKUP;
  entry PUTDOWN;
end;

task body FORK is
begin loop do accept PICKUP;
               accept PUTDOWN;
            end loop;
end;
```

- Bei PUTDOWN muß der Philosoph warten, da es keine freilaufenden Nachrichten gibt.
- Da alle Aufrufe für einen entry streng nach dem FIFO Prinzip abgearbeitet werden, ist keine explizite Voranmeldung des wartenden Philosophen nötig. Die Gabel kommt mit zwei Zuständen aus.

Die Ada-Lösung für das Philosophenproblem ist recht kurz und elegant.

Der Kontrollfluß ist einfacher nachvollziehbar als bei DP, da die durch SELECTs und ACCEPTs angegebene Menge der erlaubten Aufrufe sich nicht aufaddiert, sondern stets aus dem lokalen Programmtext entnommen werden kann. Bei DP hingegen kann irgendeine angebrochene Prozedur fortgesetzt werden.

Der Hauptvorwurf, der Ada zu machen ist, liegt in der geringen Flexibilität. Das gleichzeitige Bearbeitenlassen von zwei Aufträgen ist ähnlich umständlich wie bei DP. Besondere Schwierigkeiten gibt es, wenn die vorgeschriebene FIFO Abarbeitung der Aufrufe des gleichen entries durchbrochen werden soll. Es gibt nämlich keine Möglichkeit, den Urheber eines Aufrufs länger zwischenspeichern als für die Zeit der Ausführung des ACCEPT-body's. Die Ausführung verschiedener ACCEPT-bodies kann auch nicht beliebig verschränkt sein wie in DP.

Im Ada Design-Rationale wird folgende Lösung vorgeschlagen: Die user rufen immer wieder TRY_REQUEST auf, bis sie einmal Erfolg hatten, d.h. bis ihr request tatsächlich bearbeitet wurde:

```
while not SUCCESS do SERVER_TRY_REQUEST(.....,SUCCESS);
```

SUCCESS ist ein OUT-Parameter, der angibt, ob der request bearbeitet wurde (TRUE) oder zurückgestellt wurde (FALSE). Der server arbeitet nun jeweils die ganze queue von Aufrufen ab (nach FIFO) und bearbeitet genau einen Aufruf wirklich. Alle anderen erhalten 'no success'. Es wird hier also die einen Aufruf betreffende Information immer wieder übersandt, da es keine Möglichkeit gibt, sie zu speichern!

Was ist ein "FIFO" ?

Ein FIFO (first in-first out) ist ein Datensammler (Datensilo) und ein Datenkomprimierer. In einem Speicher werden nach und nach Daten eingeschrieben. Das Auslesen der Daten erfolgt in gleicher Reihenfolge, wie der Einschreibevorgang. Diese Reihenfolge wird starr eingehalten und es ergeben sich für den Mikroprozessor zahlreiche Anwendungen, z.B. als STACK-Speicher, Interface zwischen Mikroprozessor und Drucker, Eingabetastatur, Anzeigensteuerung usw.

CHIP Juli 1980, S. 24

4.3.9 Zusammenfassung

Die verschiedenen Konzepte bieten unterschiedliche Möglichkeiten, um Prozesse zu **adressieren**: Indizierte Prozeßnamen oder Prozeßvariablen. Zweifellos kann man mit Prozeßvariablen **flexibler** programmieren. Die Frage ist jedoch, ob die hardware und Systemsoftware solch eine flexible Adressierung zuläßt oder ob nur feste Kanäle durchgeschaltet werden können.

Es wird teilweise die **asynchrone** (gepufferte) Kommunikation als Grundprimitivum gewählt, teilweise auch die **synchrone**. Jede der beiden Kommunikationsformen kann durch die andere simuliert werden (durch Quittungen bzw. durch einen Puffer-Prozeß). Offenbar ist es aber einfacher, in einem asynchronen Konzept Quittungen zu implementieren, als in einem synchronen Konzept vor jeden Prozeß noch einen Puffer-Prozeß zu setzen, um freilaufende SENDs zu ermöglichen.

Um requests in einer anderen Reihenfolge abarbeiten zu können, als sie eintreffen, müssen sie so abgespeichert werden, daß ein **wahlfreier** Zugriff möglich ist. Dazu werden teilweise normale Datenstrukturen (arrays, verkettete Listen) in den Prozessen verwendet (PLITS, CSP), oder die Liste der blockierten Prozedurinkarnationen ist als heap organisiert (DP) und die Auswahl des nächsten requests wird durch guarded commands (WHEN-

Statements) gesteuert. CSSA erlaubt die Entnahme von messages aus der mailbox in beliebiger Reihenfolge. Die Steuerung geschieht über Facettierung und assertions. In Ada jedoch wird die zu einem entry assoziierte Queue streng nach FIFO abgearbeitet; da die requests außerdem nicht in normalen Datenstrukturen abgespeichert werden können, ist es nicht einfach, die FIFO-Disziplin zu durchbrechen.

Die hier vorgestellten Konzepte verwenden unterschiedlich abstrakte **Sprachbausteine**. In PLITS sind die Grundprimitive das Versenden und Empfangen einer Nachricht. In anderen Konzepten werden größere Sprachbausteine verwendet, die vom konkreten message-passing abstrahieren. Für die beim message-passing am **häufigsten** auftretenden Kommunikationsstrukturen gibt es dort Sprachkonstrukte, mit denen die häufigen Fälle besonders kurz und übersichtlich programmiert werden können. Dadurch wird aber meist der Grad an Parallelität eingeschränkt. Bei den ungewöhnlicheren Kommunikationsstrukturen gibt es dann oft Schwierigkeiten. Für CSSA wurden daher Sprachbausteine unterschiedlichen Abstraktionsgrades gewählt: Neben dem expliziten Versenden und Empfangen von Nachrichten gibt es auch höhere Konstrukte wie Facetten und Operationen.

Zwei neuere Sprachkonzepte ("Input Tool Process" [BPS81], "Communication Port" [MAY81]) weisen ebenfalls gewisse Ähnlichkeiten zu CSSA auf. Ein Vergleich wäre sicherlich aufschlußreich, konnte aber an dieser Stelle nicht vorgenommen werden.

Discussions about programming languages often resemble
medieval debates about the number of angels that can
dance on the head of a pin.

John Backus

5. Sprachentwurf

Im folgenden wird auf einige Sprachkonzepte eingegangen, die möglicherweise nicht nur für CSSA relevant sind, sondern auch von allgemeinerem Interesse beim Entwurf von Programmiersprachen sein könnten.

5.1 Syntaktische Aspekte

Hinweis: Die gesamte Syntax von CSSA ist im Anhang von Teil B in Form einer erweiterten BNF-Grammatik zusammen mit einer Querverweisliste angegeben.

Ein gutes syntaktisches Layout ist für die **Anwendungsfreundlichkeit** einer Programmiersprache von großer Bedeutung. Es trägt nicht nur dazu bei, daß Programme übersichtlich, klar strukturiert, leicht lesbar und einfach zu verstehen sind, sondern kann indirekt auch den Programmentwicklungsprozeß beschleunigen und damit die Softwareerstellungskosten vermindern, indem durch die genannten Aspekte oft Fehler auf unterschiedlicher Ebene von vornherein vermieden werden und die Wartungsfreundlichkeit der Programme heraufgesetzt wird.

Das Design des syntaktischen Layouts einer Programmiersprache unterliegt damit ähnlichen Bedingungen und Anforderungen wie der Entwurf der **Benutzerschnittstelle des Compilers** (--> Kap. 7.2); auch dort soll dem Anwender bspw. durch eine gut aufbereitete Quellprogrammliste oder durch klare Fehlermeldungen in Termen der Programmiersprache eine Unterstützung gewährleistet werden, die sich positiv auf den Programmentwicklungsprozeß auswirkt.

Die syntaktischen Aspekte sollten jedenfalls unter **Berücksichtigung des Anwenders** und seiner "typisch menschlichen" Eigenschaften entwickelt werden; so sollte etwa die statische Struktur des Programmtextes die dynamische Struktur möglichst gut widerspiegeln (goto-Vermeidung!) und die Programmstruktur in geeigneter Weise "visualisiert" werden. Ein Beispiel mag dies erläutern:

In CSSA gehört zu jedem Schlüsselwort, das eine syntaktische Einheit einleitet (typischerweise ein Modul wie bspw. 'facet' oder eine Kontrollstruktur wie bspw. 'loop'), ein spezifisches **schließendes Schlüsselwort** (in den genannten Fällen 'endfacet' bzw. 'endloop'). Dadurch ist in natürlicher Weise eine **Einrückstrategie** vorgegeben: Der gesamte Programmtext innerhalb der syntaktischen Klammer sollte (sinnvollerweise!) eingerückt werden. Der konzipierte CSSA-Compiler unterstützt eine derartige Einrückstrategie bei Verwendung der struct-option (--> 7.2): Im ausgedruckten Programmtext wird vom einleitenden zum schließenden Schlüsselwort eine senkrechte Linie gezogen.

Im Gegensatz zu PL/1 oder vielen LISP-Dialekten ist es in CSSA nicht möglich, durch schließende **"Superklammern"** mehrere offene syntaktische Klammern zu beenden: Abgesehen davon, daß dies einen schlechten Programmierstil darstellt und Programme schwerer lesbar macht, würden die error-recovery Maßnahmen des Compilers

im Falle eines fehlerhaft vergessenen "end's" wesentlich erschwert werden. Auch an anderer Stelle wurde darauf geachtet, angeblich vereinfachende implizite Annahmen und Abkürzungen zu vermeiden: So müssen etwa alle Variablen und Konstanten explizit deklariert werden, auch eine Typkonversion (INT, REAL, STRING) muß (etwa wieder im Unterschied zu PL/1) explizit angegeben werden. Diese **zwingenden expliziten Angaben** dienen der Fehlervermeidung und erhöhen die Lesbarkeit der Programme.

Eine These von N. Wirth besagt, daß Strukturen, die für den Compiler einer Sprache schwierig zu erkennen sind, es auch für den menschlichen Leser sind [KOV78]. Dies läßt sich in hervorragender Weise an den Möglichkeiten der **Fehlerbehandlung eines Compilers** erkennen: Werden wie in CSSA - und etwa im Unterschied zu MODULA - für unterschiedliche Strukturen disjunkte schließende Schlüsselwörter verwendet, so kann nicht nur ein menschlicher Leser (im Normal- oder Fehlerfall) die Beziehung zur öffnenden syntaktischen Klammer (fast immer) eindeutig herstellen, sondern auch der Compiler kann, etwa bei einem Fehler innerhalb der Struktur, mit großer Sicherheit wiederaufsetzen - selbst dann, wenn lokale Strukturen falsch oder unvollständig geklammert sind! Den Benutzer nur verwirrende und sonst fast unvermeidliche Folgefehlerkaskaden (ALGOL !) lassen sich so vermeiden.

Der eigentliche Grund für die Möglichkeit solch einer guten "Fehlerbehandlung" liegt in der vorhandenen **syntaktischen Redundanz**. Tatsächlich kann ein Fehler nur dann entdeckt werden, wenn überhaupt Redundanz vorhanden ist, und nur geeignet lokalisiert werden, wenn ein genügendes Maß an Redundanz existiert - dies gilt nicht nur für eine automatische Syntaxanalyse, sondern ebenso für den menschlichen Leser! Zur Erhöhung der Redundanz trägt bspw. auch die **explizite Deklaration von Bezeichnern** bei - prinzipiell wäre es auch möglich, den Typ und andere Attribute aus dem Kontext zu ermitteln oder (wie etwa bei FORTRAN) mit default-Annahmen zu arbeiten. In diesem Fall können allerdings bspw. Schreibfehler bei Bezeichnern nicht entdeckt werden!

Aus diesen Gründen wurde bei CSSA auf ein "vernünftiges" Maß an syntaktischer Redundanz geachtet. So sind etwa die Schlüsselwörter 'is', 'of', 'do', aber auch 'returns' und sogar 'then' überflüssig, sie stellen jedoch nicht nur wertvolle **Korrekturaufsetzpunkte** dar, sondern erlauben eine leichtere Programmlesbarkeit - gerade wegen ihres redundanten Füllwortcharakters!

Insgesamt wurde bei CSSA der **Lesbarkeit** (Selbstdokumentation!) der Vorzug vor der leichten und schnellen "Schreibbarkeit" gegeben; so sind auch die schließenden Schlüsselwörter in uniformer Weise durch ein vorangestelltes 'end' aus den entsprechenden die Struktur einleitenden Schlüsselwörtern gebildet worden, selbst wenn dabei relativ lange Schlüsselwörter entstehen (wie etwa 'endprocedure'; zur Identifizierung wird dagegen in ALGOL oft noch länger "'end' 'comment' procedure xyz;" geschrieben!). Abkürzungen von Schlüsselwörtern, wie etwa bei PL/1, sind aus dem gleichen Grund nicht vorgesehen.

Die **Klammerung jeder Struktur** hat - etwa gegenüber ALGOL oder PASCAL - weitere Vorteile: Werden nachträglich an Stellen, an denen ein einziges Statement steht (bspw. ein else-Teil eines if-Statements) weitere **Befehle eingefügt**, so braucht nicht zusätzlich eine begin/end-Klammer eingefügt werden, der else-Bereich reicht in diesem Fall stets von 'else' bis 'endif'. Auch bei **lokalen Deklarationen** erübrigt sich eine begin/end-Klammerung: Diese können stets am Anfang einer derartigen Klammerstruktur, durch einen expliziten Trennstrich vom Statementteil abgesetzt, stehen. Dieser **Trennstrich** hat wieder gleichermaßen die Funktion, dem menschlichen Leser den Beginn des Statementteils aufzuzeigen und dem Compiler einen eindeutigen Aufsetzpunkt zu geben, der insbesondere bei Fehlern im Deklarationsteil einer Struktur von einer relativ großen Bedeutung ist (sonst könnte leicht im Rahmen von Korrekturmaßnahmen der Statementteil als Initialisierungsausdruck eines deklarierten Bezeichners erkannt werden!)

Schlüsselwörter sind - wie etwa bei PASCAL oder ADA, aber im Gegensatz zu PL/1 - **reserviert** und disjunkt zur Menge der Bezeichner. Das scheint schon insofern vernünftig, als nicht ein- und dieselbe Zeichenkette je nach Kontext eine den Anwender verwirrende gänzlich unterschiedliche semantische Funktion haben sollte. Außerdem ermöglicht dies nicht nur eine leichtere Syntaxanalyse, sondern oftmals im Fehlerfall auch eine bessere Fehlerlokalisierung durch eindeutigere Korrekturaufsetzpunkte und damit eine verständlichere Fehlermeldung (bis auf den seltenen Fall, daß als Name bspw. einer Variablen versehentlich ein reserviertes Wort genommen wird). Zudem ist es so schon während der lexikalischen Analyse möglich, die Schlüsselwörter bei der Aufbereitung des Programmtextes und der Ausgabe der Programmliste gesondert zu kennzeichnen und auch damit die Lesbarkeit des Programms zu erhöhen.

Eine Reihe weiterer Maßnahmen, von denen hier nur noch einige stichpunktartig genannt werden sollen, dienen der Fehlervermeidung und der Erleichterung des Umgangs mit Programmen:

- Das **Semikolon** fungiert als Terminator, nicht als Separator (ansonsten eine häufige Fehlerquelle).
- Zur leichteren Lesbarkeit können in Bezeichnern signifikante (!) **"break-characters"** (Unterstreichungsstriche) eingefügt werden.
- Es gibt sowohl geklammerte **"inline"-Kommentare**, als auch **"stand-alone"-Kommentarzeilen**, die durch ein Sonderzeichen in Spalte 1 gekennzeichnet sind. Diese Lösung ist "dokumentationsfreundlich".
- Die Eingabe erfolgt **formatfrei**, das logische Zeilenende wirkt wie ein Leerzeichen.
- Semantisch ähnliche Konstrukte erhielten eine syntaktisch ähnliche Form, semantisch unterschiedliche jedoch auch eine syntaktisch unterschiedliche Form.

- Die gesamte Syntax wurde "geglättet", um durch ein natürliches Aussehen Fehler von vornherein zu vermeiden.

Bei allen Maßnahmen wurde die **LL(1)-Eigenschaft** und die **one-pass Analysierbarkeit** berücksichtigt - somit sind CSSA-Programme nicht nur leicht lesbar, sondern auch einfach zu übersetzen.

5.2 Lokale Kontrollstrukturen

Im Unterschied zu den CSSA-spezifischen **globalen Kontrollstrukturen**, die Kontrollabstraktionen auf höherer Ebene durch Facettierung und Operationsaktivierung mittels Kommunikation zwischen Agenten realisieren, wird durch die **lokalen Kontrollstrukturen** der Kontrollfluß innerhalb eines Moduls (Operation, Funktion, Prozedur) gesteuert.

Der Kontrollfluß verläuft normalerweise **linear**, indem die einzelnen Statements, die in Form einer Statementliste aneinandergereiht sind, nacheinander ausgeführt werden. Von diesem sequentiellen Kontrollfluß kann nur in sehr kontrollierter Form abgewichen werden, und zwar i.w. durch die folgenden drei Möglichkeiten:

- **Aufruf einer Prozedur** (explizit durch 'call') oder einer **Funktion** (implizit in einem Ausdruck) mit anschließender Fortführung des sequentiellen Kontrollflusses, es sei denn, bei Ausführung der Prozedur tritt eine der im letzten Punkt genannten Möglichkeiten ein.
- **Mehrfaches, bedingtes oder alternatives** Ausführen von Statementgruppen innerhalb einer loop-, if- oder select-Struktur.
- **Abbruch** der sequentiellen Statementliste durch 'escape', 'continue' oder 'signal' innerhalb von Schleifen, durch 'return' innerhalb von Funktionen, durch 'stop', womit die Operation beendet wird, oder durch einen Facettierungsbefehl, durch den die aktuelle Operation ebenfalls beendet wird. Schließlich kann durch 'terminate' der gesamte Agent (irreversibel) in den terminierten Zustand übergehen.

Durch die letzte Gruppe von Befehlen kann lediglich eine Struktur **verlassen** werden, sie dienen i.w. als Ersatz für einen nicht vorhandenen **Sprungbefehl** ("goto"): Maßnahmen für einen "freien" Kontrolltransfer sind in CSSA nicht vorgesehen!

Die Aktivierung von Funktionen und Prozeduren enthält im Vergleich zu den anderen bekannten Programmiersprachen keine neuen Konzepte; anzumerken ist lediglich, daß Funktionen keinerlei Seiteneffekt verursachen dürfen. Die Befehle der dritten Gruppe, die den Abbruch des sequentiellen Kontrollflusses veranlassen, sind in Teil B genauer beschrieben.

Die **Auswahlstrukturen** 'if' und 'select' (--> B.13.2), wobei letztere die direkte Auswahl zwischen mehr als zwei Alternativen in Form einer Abfragekette realisiert, haben die etwa aus PASCAL bekannte Semantik ('select' entspricht dabei in etwa dem case-Statement). Allerdings sind beim Vergleich zu PASCAL folgende Vorteile anzumerken:

- Eine begin/end-Klammerung mehrerer Statements in einem der Alternativzweige ist nicht notwendig (--> 5.1).
- Auch lokale Deklarationen in einem Alternativzweig brauchen nicht durch begin/end geklammert zu werden.
- Durch die if/endif-Klammerung werden syntaktische Mehrdeutigkeiten ("dangling else" in ALGOL oder PASCAL) vermieden.
- Das select-Statement enthält - im Unterschied zur case-Anweisung von Standardpascal - einen optionalen 'otherwise'-Ausgang.

Die **loop-Struktur** (--> B 13.2) dient dem mehrfachen Durchlaufen von Statementfolgen. Sie kann in mehreren Varianten auftreten:

- Als **rein repetitive Schleife**: Die Statementfolge wird solange wiederholt durchlaufen, bis die Schleife durch eine der oben genannten Möglichkeiten abgebrochen wird. Insbesondere kann eine derartige Schleife durch 'escape' verlassen werden.
- Als **while-Schleife**: Die Statementfolge wird solange wiederholt ausgeführt, bis eine bei jedem Schleifendurchlauf erneut ausgewertete Abbruchbedingung erfüllt ist.
- Als **ereignisgesteuerte Schleife** ('until'): Die Statementfolge wird solange durchlaufen, bis ein spezifiziertes Ereignis eintritt. Das Eintreten eines Ereignisses wird durch 'signal <Ereignisname>' angegeben. Sodann wird ein dem Ereignis zugeordneter sogenannter **exit-Block** ausgeführt.
- Als **Zählschleife** ('for...in'): Ein implizit deklarierter Schleifenindex ändert bei jedem Schleifendurchlauf seinen Wert in kontrollierter Weise. Der Schleifenindex existiert nur innerhalb der Schleife und besitzt die Konstanteneigenschaft.
- Als **Durchlaufschleife** für **Mengen** und **Relationen** ('for...of'): Eine implizit deklarierte Konstante vom Typ der in der Menge oder der Relation gespeicherten Elemente nimmt nacheinander die Werte aller Elemente an.

Die Schleifentypen können beliebig **kombiniert** werden, lediglich eine Kombination von Zähl- und Durchlaufschleife ist wenig sinnvoll und daher ausgeschlossen. Oft werden jedoch die Basistypen überwiegen, die alle nach dem gleichen Muster

loop <Spezifikation> **do** <ACTIONS> **endloop**

aufgebaut sind. Das **vollständige Syntaxschema** allerdings lautet:

```

loop [<IDENTIFIER>]
  [ for <IDENTIFIER>
    ( in <EXPRESSION> '..' <EXPRESSION>
      [ by <INT-EXPRESSION> ]
    | of <IDENTIFIER>
    )
  ]
  [ while <BOOL-EXPRESSION> ]
  [ until <IDENTIFIER> || or ]
do <ACTIONS>
  ( exit <IDENTIFIER> is <ACTIONS> ) *
endloop

```

Man beachte, daß sowohl eckige Klammern (Optionalteile) als auch runde Klammern Zeichen der die Syntax beschreibenden Metasprache sind! <ACTIONS> bezeichnet eine Folge von Statements mit einem optionalen Deklarationsteil. Schleifen können **benannt** werden, der Name wird dann sofort nach dem Schlüsselwort 'loop' angegeben. Dies ist bei verschachtelten Schleifen von Nutzen, wenn durch 'escape' eine bestimmte Schleife abgebrochen werden soll oder mit 'continue' die nächste Iteration einer äußeren Schleife eingeleitet werden soll (Bsp. s.u.).

Das Schleifenkonzept von CSSA weist eine gewisse **Ähnlichkeit zu ADA** auf. Auch in ADA ist ein uniformes syntaktisches Layout für die repetitive Schleife, die while-Schleife und die for-Schleife vorhanden, gibt es ein "exit-Statement" zum Verlassen von Schleifen und wird der Schleifenindex als eine implizit lokal deklarierte Konstante angesehen. Allerdings besitzt CSSA gegenüber ADA zwei zusätzliche "features":

- Die Schleifentypen können **kombiniert** werden. Dies führt keineswegs notwendigerweise zu unübersichtlichen Konstruktionen; als Beispiel sei eine Schleife angegeben, bei der ein array nur solange durchlaufen wird, bis eine bestimmte Grenze erreicht wird:

```

loop CANDIDATES for NEWPRIME in 3..MAX by 2 do
  loop for I in 1..TBL_LENGTH
    while PRIME(I)**2 < NEWPRIME do
      if DIV(PRIME(I),NEWPRIME)
      | then continue CANDIDATES;
      endif;
    endwhile;
    (* a new prime found *)
  endloop;
endloop;

```

- CSSA enthält im Unterschied zu ADA **ereignisgesteuerte Schleifen** (sogen. "Zahn-Konstrukt"). Mit dem durch 'signal' signalisierten Eintreten eines Ereignisses kann nicht nur eine Schleife "mittendrin" (wie alleine schon durch 'escape') verlassen werden (sogen. n-einhalb-Schleifen),

sondern es wird zusätzlich ein zugehöriger exit-Block, eine Folge von Statements, ausgeführt. Diese Möglichkeit eliminiert nicht nur einen häufigen Anwendungsfall von goto-Befehlen, sondern vermeidet auch flags, die sonst die Tatsache des Eintretens des Ereignisses über die Schleife hinaus retten. Ereignisgesteuerte Schleifen werden ausführlich in [KNU74] diskutiert; die Argumente sollen hier nicht wiederholt zu werden.

5.3 Relationen

CSSA-Relationen sind dynamische Datenstrukturen, die das Prinzip des inhaltsadressierten **Assoziativspeichers** bereits auf Sprachebene zur Verfügung stellen. Das Relationenkonzept stellt den Versuch dar, die grundlegenden Eigenschaften des **Codd'schen Relationenmodells**, das sonst nur in Datenbankmanagementsystemen zur Anwendung kommt, in eine Programmiersprache zu integrieren. Neben der bestechenden Klarheit des Codd'schen Modells hat auch die Erkenntnis, daß in verteilten Systemen mit mehreren **disjunkten Adressräumen** die Verwendbarkeit von pointern (Adressen) stark eingeschränkt ist, die Entwicklung des CSSA-Relationenkonzepts motiviert.

Relationen sind ungeordnete **Mengen von records** gleichen Typs. Dementsprechend können records in Relationen eingefügt (durch 'insert') und wieder entfernt werden (durch 'delete'). Ein Durchlaufen aller records in einer Schleife ist möglich.

```
type ARTIKEL is record int: TEIL;--> real: PREIS; endrecord;
relation PREISLISTE of ARTIKEL;
insert A into PREISLISTE;
delete A in PREISLISTE;
loop for A of PREISLISTE do A.PREIS:=A.PREIS*1.13; endloop;
```

Zusätzliche Möglichkeiten ergeben sich daraus, daß bei der Deklaration eines records spezielle **Schlüsselfelder** vereinbart werden können (die Felder vor '-->'), über die ein record eindeutig identifizierbar ist. Beim Einfügen eines records wird daher evtl. ein anderer mit gleichem Schlüssel **überschrieben**. Beim Entfernen genügt es auch, anstelle eines ganzen records nur die Schlüsselwerte anzugeben:

```
delete PREISLISTE(7385);
```

Vor allem aber ist das **Wiederauffinden** eines records über den Schlüssel möglich. Dazu bietet sich zunächst eine Notation wie beim array-Zugriff an:

```
PREISLISTE(7385).PREIS:=3.75;
```

Da aber immer die Möglichkeit berücksichtigt werden muß, daß die Suche in der Relation erfolglos bleibt, wurde eine spezielle **Kontrollstruktur** für die Aufsuche gewählt:

```
find PREISLISTE(7385) --> X do
|   X.PREIS:=3.75;
|   otherwise do print ("Teil nicht gefunden");
endfind;
```

X ist dabei eine **implizit** deklarierte lokale Variable, die nur innerhalb des FIND-Befehls verwendet werden kann, und die den Wert des gefundenen records hat.

Relationen sind im wesentlichen die einzigen Datenstrukturen in CSSA, die eine **dynamische Speicherplatzzuweisung** ermöglichen. Auf dynamisch allozierbare records mit pointern wurde bewußt verzichtet. Die Speicherplatzvergabe erfolgt daher in sehr kontrollierter Form. Speicherplatz kann nur explizit durch 'delete' freigegeben werden. Ein garbage-collector, der nicht mehr zugreifbare Objekte entdeckt, ist nicht notwendig.

Relationen stellen abgeschlossene Speicherbereiche dar. Innerhalb von Relationen können mit Hilfe von **logischen pointern** (Schlüsselwerten) Datenstrukturen wie verkettete Listen, Bäume usw. aufgebaut werden (vgl. Teil B, Kapitel 11). Solche Strukturen können dann auch verschickt werden, während herkömmliche pointer (Adressen) beim Verschicken ihre Bedeutung verlieren würden: Sie haben nur innerhalb eines bestimmten Adressraumes einen Sinn.

In der Regel wird aber die Verwendung von Relationen Pointerstrukturen überflüssig machen und elegantere Lösungen erlauben. Das folgende Beispiel, in dem die Symboltabelle für den Compiler einer blockorientierten Sprache als Relation dargestellt ist, soll das untermauern:


```

type ENTRY is record string: IDENTIFIER; int: BLOCKLEVEL;
    |
    | --> string: CLASS,DATATYPE;
    |
    endrecord;
relation SYMBOL_TABLE of ENTRY;
var int: BLOCKLEVEL; (* current level *)

function LOOKUP( string: IDENTIFIER) returns ENTRY is
    |
    | loop for I in BLOCKLEVEL .. 1 by -1 do
    | | find SYMBOL_TABLE(IDENTIFIER,I)-->E do
    | | | return E;
    | | endfind;
    | endloop;
    | print ("undeclared identifier");
    | return CHAMAELEON;
    |
    endfunction;

procedure DECLARE( string: ID) --> (ENTRY: NEW_ENTRY) do
    |
    | NEW_ENTRY.IDENTIFIER:=ID;
    | NEW_ENTRY.BLOCKLEVEL:=BLOCKLEVEL;
    | find NEW_ENTRY in SYMBOL_TABLE do
    | | print ("identifier redeclared");
    | | otherwise do insert NEW_ENTRY into SYMBOL_TABLE;
    | endfind;
    |
    endprocedure;

procedure NEW_BLOCK is
    |
    | BLOCKLEVEL:=BLOCKLEVEL+1;
    |
    endprocedure;

procedure BLOCK_EXIT is
    |
    | BLOCKLEVEL:=BLOCKLEVEL-1;
    | loop for E of SYMBOL_TABLE do
    | | if E.BLOCKLEVEL>BLOCKLEVEL
    | | | then delete E in SYMBOL_TABLE;
    | | endif;
    | endloop;
    |
    endprocedure;

```

Für eine effiziente Implementierung von Relationen bieten sich Hash-Tabellen an. Zusätzlich kann mit Hilfe eines Kurzzeitgedächtnisses ("look aside buffer"), in dem Verweise auf die zuletzt angesprochenen records gespeichert werden, erreicht werden, daß dicht aufeinander folgende Zugriffe auf denselben record sehr effizient ausgeführt werden.

6. Compilerbautechnik

In diesem Kapitel werden einige im realisierten CSSA-Compiler zur Anwendung kommende Konzepte der Compilerbautechnik dargestellt. Es wird dabei lediglich auf solche Aspekte eingegangen, die möglicherweise von allgemeinerem Interesse sein könnten. Einzelheiten der Implementation finden sich in Teil E.

6.1 Symboltabellenorganisation

In der Symboltabelle des Compilers sind die **Bezeichner**, die im Laufe der Analyse des Quellprogramms entdeckt werden, zusammen mit ihren **Attributen** (wie bspw. Typ, Typklasse oder Blockverschachtelungstiefe) abgespeichert. Während die Bezeichner selbst als reine Zeichenkettensymbole vom Scanner gefunden werden, stellen die zugehörigen Attribute das Ergebnis einer semantischen Analyse des Kontextes, in dem der Bezeichner auftritt, dar. Die in der Symboltabelle gespeicherte Information wird sowohl zur Überprüfung der **semantischen Korrektheit** des Quellprogramms, als auch zur **Codegenerierung** benötigt.

Für jeden Bezeichner des Quellprogramms wird bei seinem **definierenden Auftreten** ein Symboltabelleneintrag angelegt. Mindestens bei jedem **angewandten Auftreten** eines Bezeichners werden zugehörige Attribute aufgesucht oder nachgetragen. **Schlüsselwörter** sind ebenfalls in der Symboltabelle enthalten, auf diese Weise kann während der lexikalischen Analyse in einem einzigen Suchvorgang festgestellt werden, ob es sich bei einem Wortsymbol um ein Schlüsselwort oder einen Bezeichner handelt, und gleichzeitig kann so die interne Codierung des Schlüsselwortes bzw. ein Verweis auf den Eintrag zurückgeliefert werden. Da in einem typischen CSSA-Programm 60-70% aller lexikalischen Einheiten Schlüsselwörter oder Bezeichner darstellen, sollte die Symboltabellenverwaltung in **effizienter Weise** realisiert werden.

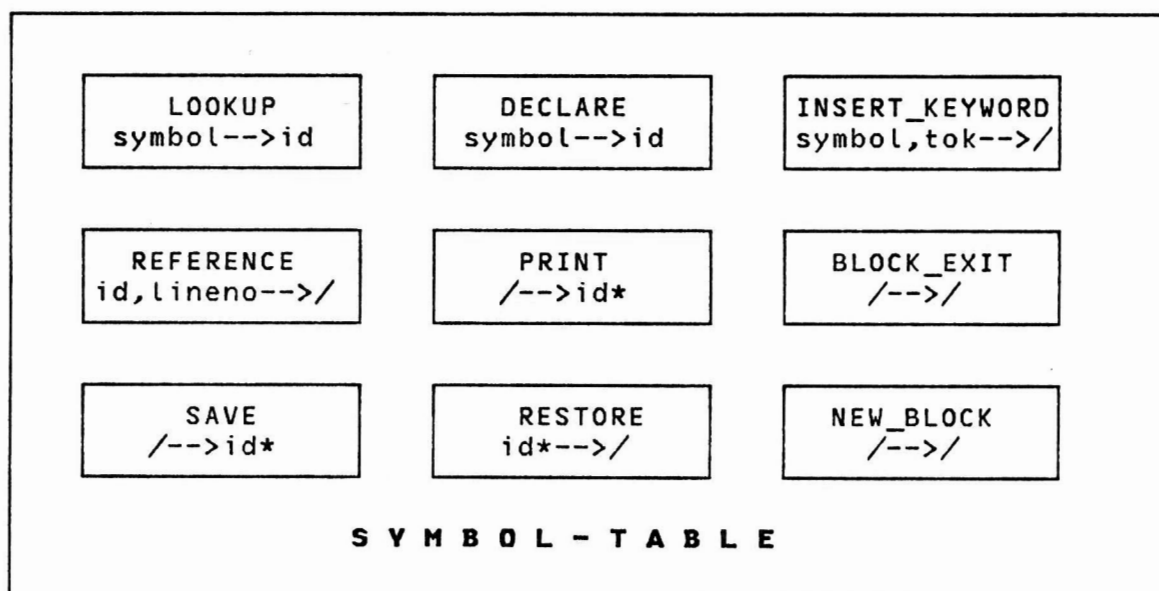
Die **Implementierung** von Symboltabellen kann auf unterschiedliche Art erfolgen. Da es sich bei den Tabelleneinträgen um Informationseinheiten mit einer eindeutigen Schlüsselkomponente handelt (Name, evtl. zusammen mit einer Blockkennzeichnung), kommen grundsätzlich so bekannte Techniken wie Speicherung in (evtl. kellerartig organisierten) **linearen Listen**, **sortierten Listen**, **Binärbäumen** und **hashcodierten** Datenstrukturen in Betracht. Keine dieser Organisationsformen ist in jeder Hinsicht optimal, ihre Vor- und Nachteile werden bspw. in [MCK74a] ausführlich diskutiert. Hier wird im folgenden eine Methode vorgestellt, die die **verschiedenen Vorteile** der unterschiedlichen Verfahren **kombiniert**.

Der CSSA-Compiler arbeitet syntaxgesteuert, der Scanner wird als Hilfsroutine vom Parser aufgerufen; in gleicher Weise stellt die Symboltabelle nur eine Reihe von **Dienstleistungsfunktionen** zur Verfügung, die vom Parser während der semantischen Analyse indirekt über den Scanner oder direkt in Anspruch genommen werden. Daher ist es angebracht, die Symboltabelle zusammen mit den von ihr zur Verfügung gestellten Operationen als einen **abstrak-**

ten Datentyp anzusehen und zu implementieren. Dementsprechend wird zunächst auf die benötigten **Operationen** und ihre Aufgaben eingegangen, bevor die Repräsentation der Daten und die Implementierung der Operationen skizziert und diskutiert wird. Gegenüber der tatsächlichen Implementierung der Symboltabelle (—> E 2.3) erfolgt die Beschreibung hier allerdings in leicht idealisierter Form, indem auf die Darstellung einiger unwesentlicher Details verzichtet wurde und die an einigen Stellen aus Effizienzgründen vorgenommene Abweichung vom streng modulatorientierten Programmieren (wie Ausnutzung von Seiteneffekten und Implementierungsinformation) ignoriert wurde.

Die oben erwähnten Operationen zur **Definition** eines Tabelleneintrages und zum **Aufsuchen** eines Eintrages mit der Möglichkeit, die zugehörigen Attribute zu inspizieren und abzuändern, genügen nicht. Es soll auch die Möglichkeit bestehen, die gesamte Tabelle in einer gewissen Ordnung zu **durchlaufen**, bspw. um eine sortierte cross-reference Liste zu erzeugen oder die Schreibfehlerkorrekturroutinen bei der Suche ähnlicher deklarierter Bezeichner zu unterstützen. Die Situation wird auch dadurch verkompliziert, daß CSSA eine **blockstrukturierte Sprache** ist, also gleichlautende Bezeichner in verschiedenen Blöcken u.U. verschiedene Tabelleneinträge erhalten müssen, und bei **forward-Vereinbarungen** von Prozeduren, Funktionen und Facetten die formalen Parameter erst bei der späteren Definition der eigentlichen Prozedur, Funktion bzw. Facette diesem Gültigkeitsbereich zugeordnet werden müssen. Schließlich stellen (lokale) scripts **Sichtbarkeitsschranken** dar. Die beiden letzten Eigenarten tauchen bspw. bei ALGOL nicht auf.

Die folgende Skizze benennt die von der Symboltabelle zur Verfügung gestellten **Operationen**, mit denen die an sie gestellten Anforderungen erfüllt werden können:



Hierbei bedeuten:

- **symbol:** Ein Wortsymbol, also eine Bezeichner oder ein Schlüsselwort, wie es vom Scanner im Quellprogramm entdeckt wird.
- **id:** Ein Verweis auf einen Tabelleneintrag, der neben dem Namen des Bezeichners auch seine Attribute enthält.
- **tok:** Hier die interne Verschlüsselung eines Schlüsselwortes (z.B. 'ifsy', 'falsesy', 'scriptsy' etc.).
- **lineno:** Eine Zeilenzahl im Quellprogramm.

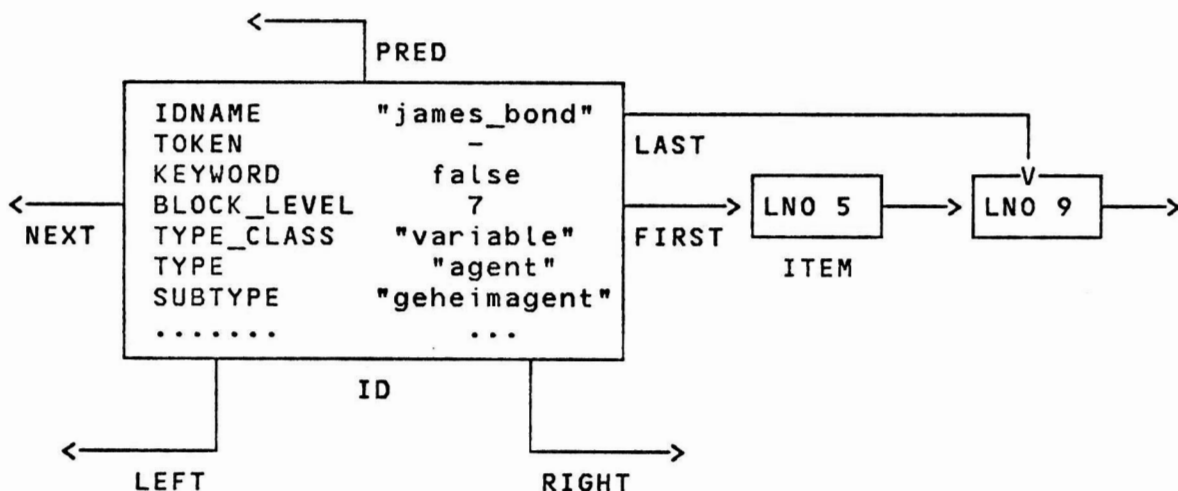
Die **Aufgabe** der oben aufgeführten Operationen wird im folgenden kurz umrissen; dabei werden auch einige charakteristische Größen der Implementierung (—> Teil E) genannt. Die **Semantik** der Operationen wird allerdings nicht im einzelnen erläutert, es wird vielmehr ein intuitives Verständnis, insbesondere für die Wirkung der blockstrukturierenden Operationen, vorausgesetzt.

- **LOOKUP** ist die zentrale Routine, die vom Scanner jedesmal aufgerufen wird, wenn im Quelltext ein Wortsymbol gefunden wird. Zurückgeliefert wird ein Verweis auf einen Tabelleneintrag; dieser Verweis ist allerdings NONE, wenn es sich um einen entsprechend den Sichtbarkeitsgrenzen der Sprache unbekannten Bezeichner handelt (NEWIDSY). Durch Inspektion von Attributen des Tabelleneintrages (wie beispielsweise KEYWORD und BLOCK_LEVEL, s.u.) kann der Scanner anschließend feststellen, ob es sich um ein Schlüsselwort handelt, bzw. ob es sich um einen lokal oder global vereinbarten Bezeichner handelt (LOCALIDSY bzw. GLOBALIDSY). Der Parser erhält ebenfalls Zugriff auf den zuletzt gefundenen Bezeichner (TBLREF und LASTID) und kann so die zur Überprüfung der statischen Semantik oder zur Codegenerierung wesentlichen Attribute ermitteln oder ergänzen.
- **DECLARE** schafft einen neuen Tabelleneintrag für ein Wortsymbol und fügt ihn in die Tabelle ein. Ein Verweis auf diesen Eintrag wird zurückgeliefert.
- **INSERT_KEYWORD** fügt Schlüsselwörter in die Symboltabelle ein. Die Routine wird bei der Initialisierung des Compilers für jedes Schlüsselwort aufgerufen.
- **REFERENCE** vermerkt zu einem Tabelleneintrag die Zeilennummer des (angewandten) Auftretens des Bezeichners. Diese Information wird für die cross-reference Liste benötigt.
- **PRINT** liefert eine sortierte Liste aller Symboltabelleneinträge. Bei der vorliegenden Implementierung wird die Menge der Bezeichner der Tabelle (gewissermaßen als "globaler Seiteneffekt") in strukturierter Form zusammen mit einigen Attributen (Typ, Typklasse) von der Routine selbst ausgedruckt.

- **BLOCK_EXIT** wird vom Parser beim Verlassen eines Sichtbarkeitsbereiches aufgerufen. Diese Routine dient der internen Organisation der Tabelle, um der Blockstrukturierung der Sprache Rechnung zu tragen.
- **NEW_BLOCK** teilt der Symboltabelle entsprechend mit, daß ein neuer Sichtbarkeitsbereich beginnt.
- **SAVE** sorgt dafür, daß alle Tabelleneinträge des letzten Sichtbarkeitsbereiches deaktiviert werden.
- **RESTORE** ermöglicht es, eine Menge von Tabelleneinträge, die früher mit SAVE deaktiviert wurden, im aktuellen Block wieder sichtbar zu machen. SAVE und RESTORE werden benötigt, da bei Parameterlisten vorwärtsdeklarerter Moduln ('functionhead', 'procedurehead', 'facethead') das strenge Blockkonzept durchbrochen wird.

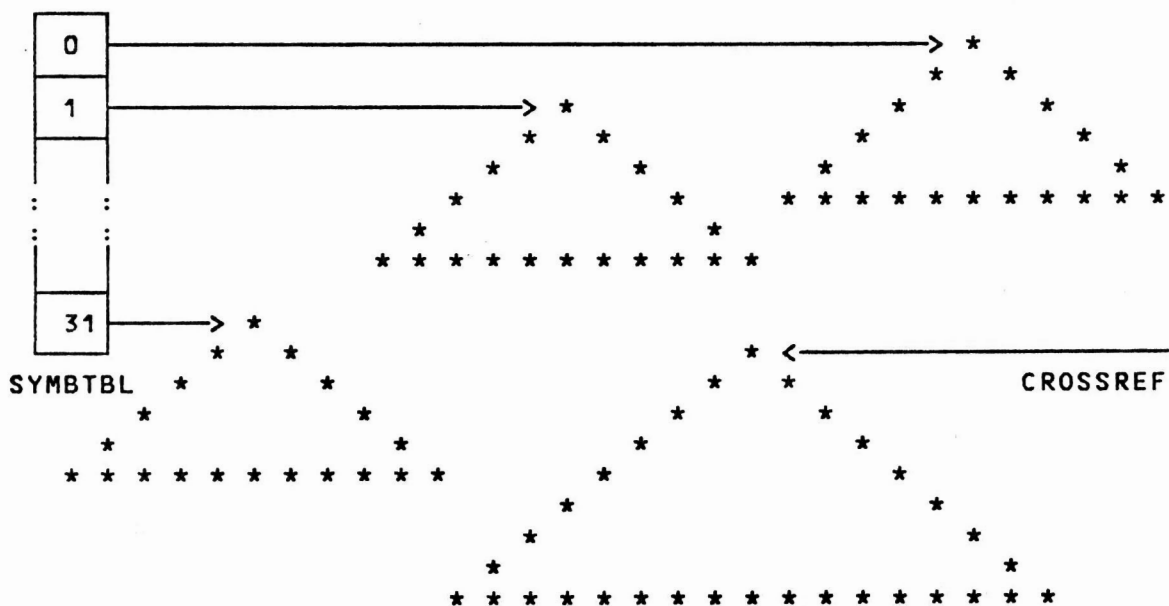
Für die Implementierung der Symboltabelle wurde als Datenstruktur ein **hashadressierter Wald aus geordneten Binärbäumen** mit einer überlagerten **Kellerstruktur** gewählt. Diese Realisierung soll in effizienter Weise die verschiedenen Vorteile der bekannten Methoden kombinieren.

Es sei zunächst der **Aufbau eines einzelnen Tabelleneintrages** beispielhaft skizziert:

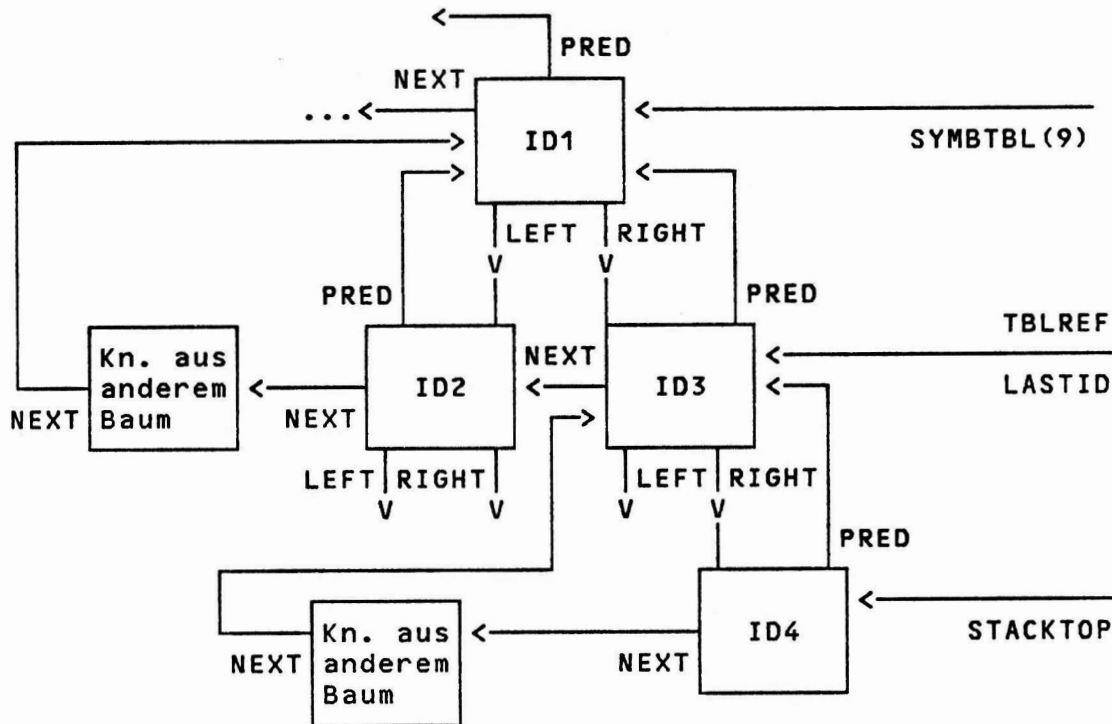


Nur die wesentlichen **Attribute** der CLASS ID, die die Typdefinition für Symboltabelleneinträge darstellt, wurden hier aufgeführt. In einer durch FIRST/LAST zugreifbaren Liste von ITEM-Objekten sind die Fundstellen der Bezeichner im Quelltext für die cross-reference Liste vermerkt, die Objekte werden durch die REFERENCE-Routine erzeugt und verkettet. Der Zeiger NEXT verweist auf das nächste, entsprechend der überlagerten Kellerstruktur ältere, ID-Objekt. Die Zeiger LEFT, RIGHT und PRED verketteten das Objekt in einem geordneten Binärbaum, PRED verweist dabei auf den Vaterknoten.

Die Symboltabelle besteht aus einem Wald aus 33 Binärbäumen. 32 davon (SYMBTBL[0] ... SYMBTBL[31]) werden über einen hash-Mechanismus adressiert, sie stellen die Menge der **aktiven Wortsymbole** (Schlüsselwörter und beim gegenwärtigen Stand der Analyse gültige oder später bspw. nach Überspringen einer Sichtbarkeitsgrenze u.U. wieder gültige Bezeichner) dar. Alle in einem Baum gespeicherten Wortsymbole besitzen den gleichen hash-code. Ein weiterer Binärbaum (CROSSREF) dient lediglich als **Historienspeicher** für die nicht mehr aktiven Bezeichner, um die cross-reference Liste am Ende des Übersetzungsvorganges ausgeben zu können.



Einen schematischen Ausschnitt aus der Symboltabelle zeigt die folgende Skizze:



Die mit ID1,...,ID4 bezeichneten Knoten stellen Einträge dar, die aufgrund des gleichen hash-codes in ein und denselben Baum mit der Wurzel ID1 untergebracht sind. Entsprechend der Ordnung der Binärbäume gilt: $IDNAME(ID2) < IDNAME(ID1) < IDNAME(ID3) < IDNAME(ID4)$. Der Zeiger STACKTOP zeigt auf dasjenige Objekt, das im gegenwärtig innersten aktiven Block zeitlich zuletzt in die Symboltabelle eingefügt wurde. Die Zeiger NEXT dienen der Repräsentation der überlagerten Kellerstruktur, diese Verkettung läuft quer durch alle 32 SYMBTBL-Bäume.

Zur **Arbeitsweise** der oben genannten Operationen auf diesen Datenstrukturen mögen einige kurze Bemerkungen genügen:

- ◊ **LOOKUP** bedient sich des vom Scanner beim Einlesen der einzelnen Zeichen eines Wortsymbols bereits ermittelten hash-codes (s.u.), um den Einstieg in den richtigen Baum zu finden. Sodann wird entsprechend der lexikographischen Ordnung der zugehörige Pfad durch den Baum bis zu einem Blatt oder Schlüsselworteintrag verfolgt. Auf das letzte (!) Vorkommen des gesuchten Bezeichners wird der Zeiger TBLREF (bzw. intern zunächst LASTID) gesetzt. Falls das Wortsymbol nicht gefunden wird, so wird ein interner Hilfszeiger TBLPREV auf das Blatt des durchlaufenen Pfades gesetzt.
- ◊ **DECLARE** fügt ein neues ID-Objekt als Blatt entsprechend der Ordnung als linken bzw. rechten Nachfolger desjenigen Knotens an, auf den der von LOOKUP vorher gesetzte Zeiger TBLPREV verweist. STACKTOP wird auf das neue Objekt gesetzt, der Wert des BLOCK_LEVEL-Attributs wird eingetragen und die NEXT-Verkettung wird ergänzt.

- **INSERT_KEYWORD** arbeitet wie LOOKUP und DECLARE zusammen, der hash-code des Schlüsselwortes wird allerdings vorher ermittelt.
- **PRINT** durchläuft zum Ausdrucken der Symboltabelle den CROSSREF-Baum in symmetrischer Ordnung.
- **BLOCK_EXIT** durchläuft ausgehend von STACKTOP die NEXT-Verkettung und kettet sukzessive Blätter (!) der Bäume aus (hierzu dient der PRED-Rückverweis), solange bis ein anderer Kellerabschnitt erreicht wird. Dies ist der Fall, wenn das BLOCK_LEVEL-Attribut eines Eintrages einen kleineren Wert annimmt. Der Speicherplatz von ID-Objekten, die aus der Symboltabelle entfernt werden, wird wieder freigegeben, falls keine cross-reference Liste erstellt werden soll (Compiler-option), ansonsten werden die Objekte in den CROSSREF-Baum eingefügt.
- **NEW_BLOCK** erhöht lediglich einen Zähler für die Blockverschachtelung, so daß bei folgenden DECLARE-Aufrufen das BLOCK_LEVEL-Attribut richtig gesetzt werden kann.

Die Arbeitsweise von **REFERENCE**, **SAVE** und **RESTORE** dürfte ohne weitere Erläuterungen verständlich sein. Die **Sichtbarkeitsgrenze von scripts** ist als ein einfacher Test bezüglich der (minimal erlaubten) Blockverschachtelungstiefe in LOOKUP implementiert. Die Routinen zur **Schreibfehlerkorrektur** durchlaufen bei der Suche nach ähnlichen Bezeichnungen die NEXT-Verkettung.

Bei der **Effizienzbewertung** der hier beschriebenen Symboltabellenrealisierung müssen folgende Gesichtspunkte berücksichtigt werden: 60-70% aller lexikalischen Einheiten eines CSSA-Programms stellen Wortsymbole dar, für die eine Tabellenaufsuche durch die LOOKUP-Routine vorgenommen wird. Davon sind wiederum ca. 60% Schlüsselwörter, so daß diese alleine ca. 40% aller lexikalischen Einheiten eines CSSA-Programms ausmachen. Von den oben genannten Operationen stellt daher die LOOKUP-Operation diejenige dar, die am weitaus häufigsten angewendet wird.

Die **LOOKUP-Routine** wird ausschließlich vom Scanner aufgerufen, aus Effizienzgründen wurde sie daher als inline-code und in iterativer statt rekursiver Form realisiert. Die **hash-Funktion** ist so gewählt, daß ihr Wertebereich gleichmäßig verteilt ist und sie einfach berechnet werden kann: Es wird die Summe der durch den EBCDIC-Code zugeordneten Zahlen aller Zeichen des Wortsymbols modulo 32 gebildet, diese Summenbildung wird vom Scanner bereits vorgenommen, während er die einzelnen Bytes des Wortsymbols einliest. Das SYMBTBL-array kann als **hash-Tabelle** angesehen werden, im Unterschied zu üblichen Verfahren zur Kollisionsbehandlung wird hier nicht mit rehashing oder sequentiellen Überlauflisten gearbeitet, sondern mit **"Überlaufbäumen"**, die einen Zugriff mit logarithmischer Zeitkomplexität garantieren. Da die Schlüsselwörter während der Initialisierungsphase des Compilers als erste in die Bäume eingefügt werden, stehen sie nahe der Wurzel und werden daher besonders schnell gefunden.

Zur **Effizienz** der übrigen Routinen mögen einige stichpunktartige Bemerkungen genügen: **INSERT_KEYWORD** wird auf jedes Schlüsselwort genau einmal angewendet und hat logarithmische Zeitkomplexität. **DECLARE** wird für jeden vorkommenden Bezeichner einmal aufgerufen und benötigt nur konstante Zeit, da vorher durch **LOOKUP** bereits die Einfügestelle ermittelt wurde. **BLOCK_EXIT** und **NEW_BLOCK** sind ebenfalls sehr effizient: **NEW_BLOCK** erhöht lediglich einen Zähler, bei **BLOCK_EXIT** wird eine Liste sequentiell durchlaufen (konstante Zeit pro Bezeichner); falls eine cross-reference Liste gewünscht wird, wird allerdings nochmals logarithmisch Zeit pro Bezeichner benötigt. **SAVE** und **RESTORE** werden sehr selten und jeweils nur für wenige Bezeichner benötigt.

Zum Schluß noch einige **Anmerkungen** zur gewählten Realisierung und eine kurze **Diskussion** möglicher Designalternativen.

Ein **hash-Schema** wurde gewählt, da aus der Untersuchung von McKeeman [MCK74a] eindeutig hervorgeht, daß eine derartige Realisierung i.a. den schnellsten Zugriff (durch eine minimale Zahl von Speicherzugriffen) auf einen Eintrag gewährleistet. Dabei ist allerdings der zusätzliche Aufwand zur Berechnung der hash-Funktion vernachlässigt worden, der hier jedoch recht klein ist. Um in jedem Fall, auch bei großen Programmen mit vielen Bezeichnern, eine logarithmische Zugriffszeit zu garantieren, wurden **binäre Überlaufbäume** anstelle der sonst üblichen linearen Überlauf Listen verwendet. Ein noch schnellerer Zugriff wäre möglich, wenn an Stelle der 32 Bäume mehrere 100 Bäume verwendet würden - dann müßte nur in seltenen Fällen ein (kleiner) Überlaufbaum durchsucht werden, für alle Programme mit einer "vernünftigen" Anzahl von Bezeichnern wäre ein nahezu **direkter Zugriff** gewährleistet. Bei der vorliegenden Implementierung wurde die Anzahl der Bäume jedoch auf 32 beschränkt, um Speicherplatz beim SYMBTBL-array und bei sonst oft überflüssigerweise erzeugten Baumwurzelobjekten zu sparen; durch einen kleinen zusätzlichen Implementierungsaufwand ließe es sich allerdings vermeiden, Speicherplatz für nicht benötigte Objekte anzufordern.

Im kommerziellen **Fortran H-Compiler** werden für jede mögliche Länge von Bezeichnern (in Fortran auf sechs beschränkt) verschiedene Bäume verwendet. Dies hat jedoch den Nachteil, daß die Bäume unterschiedlich voll werden, da kurze Namen i.a. häufiger als längere Namen verwendet werden. Die Anzahl der Bäume ist begrenzt, die Effizienz ist nicht mit hash-Verfahren vergleichbar.

Interessant ist die in [PEM82] beschriebene Symboltabellenimplementierung des in Zürich entwickelten portablen **PASCAL-Compilers** von Ammann u.a. Dabei wird für jedes **Blockniveau** ein **eigener Binärbaum** verwendet, der über ein **display** - das die maximale Verschachtelungstiefe begrenzt - adressiert wird. **BLOCK_EXIT** und **NEW_BLOCK** sind einfach zu realisieren, es muß i.w. nur der Pegelindikator des displays erniedrigt bzw. erhöht werden. **SAVE** und **RESTORE** können ebenfalls leicht implementiert werden: Der gesamte Baum eines Gültigkeitsbereiches kann durch eine einfache Zeigermanipulation entfernt werden oder wieder über das display adressierbar gemacht werden. **DECLARE** fügt einen Bezeichner in denjenigen Baum ein, der dem aktuellen Gültigkeitsbereich zugeordnet ist. Allerdings kann die sehr häufig

benötigte LOOKUP-Operation nicht so effizient wie im CSSA-Compiler ausgeführt werden: Es werden nacheinander alle Bäume der verschiedenen Gültigkeitsbereiche von innen nach außen durchlaufen, bis der Bezeichner (eventuell !) gefunden wird. Da vordefinierte Bezeichner wie 'integer', 'real', 'nil', 'get' etc. auf Blockniveau 0 implizit deklariert sind, müssen in diesen (häufigen) Fällen stets alle Bäume durchsucht werden. Eine cross-reference Liste ist nicht vorgesehen. Zwar könnten bei BLOCK_EXIT ähnlich wie beim CSSA-Compiler alle Bezeichner in einen eigenen CROSSREF-Baum eingefügt werden, jedoch müßte dann in aufwendiger Weise dafür gesorgt werden, daß die Bäume nicht in einer Standardordnung durchlaufen werden, da der CROSSREF-Baum sonst entartet, was zu Effizienzverlusten führt.

Ähnliche Nachteile weisen Implementierungen mit **Blocktabellen** auf (vgl. etwa [ZIMA82]): Auch hier muß die LOOKUP-Operation sukzessive die lokalen Symboltabellen der jeweils umfassenden Blöcke inspizieren. Zudem erfordert eine effiziente Realisierung die Implementierung der Blocktabelle als ein array; die Obergrenze dieses arrays stellt dann eine Grenze für die Gesamtzahl der Blöcke (nicht der Verschachtelungstiefe!) eines zu übersetzenden Programms dar.

Anstatt im CSSA-Compiler die ID-Objekte bei BLOCK_EXIT aus den SYMBTBL-Bäumen zu entfernen und in den CROSSREF-Baum einzufügen, wäre es auch möglich gewesen, diese durch einen **Löschanzeiger** lediglich **logisch zu deaktivieren**. Das würde jedoch bedeuten, daß bei jeder LOOKUP-Operation eine zunehmend größere Menge deaktivierter Elemente durchlaufen werden müßte. Dies ist gerade bei CSSA ein großer Nachteil, da hier üblicherweise mehrere auf oberstem Blockniveau parallele scripts übersetzt werden. Aus diesem Grund wurde beschlossen, die Einträge von Bezeichnern bei Verlassen eines Sichtbarkeitsbereiches physisch aus der eigentlichen Symboltabelle zu entfernen, so daß i.w. stets nur die Information der **jeweils gültigen Umgebung** in der Symboltabelle enthalten ist. Da hierbei die NEXT-Verkettung durchlaufen wird, kann dies nicht nur in effizienter Weise geschehen, sondern die Bezeichner werden in einer lexikographisch willkürlichen Ordnung in den CROSSREF-Baum eingefügt, so daß dieser i.a. nicht entartet.

Schließlich sollte noch erwähnt werden, daß die **cross-reference Liste** durch NEW_BLOCK und BLOCK_EXIT unter Benutzung von **syntaktischer Strukturinformation** aufgebaut wird. Dadurch kann zwischen gleichlautenden Bezeichnern, die in verschiedenen Gültigkeitsbereichen deklariert sind, unterschieden werden. Dies ist bspw. im kommerziellen **SIMULA-Compiler** nicht der Fall, dort wird lediglich lexikalische Information zum Aufbau der cross-reference Liste benutzt.

6.2 Parser

Der CSSA-Compiler besitzt einen syntaxgesteuerten LL(1)-Parser, der nach der Methode des **rekursiven Abstiegs** arbeitet. Diese Methode wurde in Hinblick auf eine spätere Implementierung auf Mikrocomputersystemen gewählt, da zum einen der Parser einfach aus der Grammatik hergestellt werden kann und zum anderen in den so erhaltenen Parser problemlos die Überprüfungen der statischen Semantik und die Routinen der Codegenerierung eingebaut werden können. Außerdem ist solch ein Parser in hohem Maße selbstdokumentierend und leicht änderbar; letzteres ist besonders bei einer in der Entstehung befindlichen Sprache äußerst wichtig, da hier auch nach einer ersten Implementierung noch kleinere Sprachänderungen möglich sein sollen.

In diesem Abschnitt soll gezeigt werden, daß sich der Parser (als SIMULA-Programm) systematisch aus den Regeln einer erweiterten BNF-Grammatik herstellen läßt. Die Verwendung eines **Parsergenerators** wäre prinzipiell möglich gewesen, ist aber aus Rentabilitätsgründen nicht in Frage gekommen (es wurde nur ein Parser benötigt).

Die LL(1)-Eingabesprache ist durch eine Grammatik in **erweiterter BNF-Notation** (EBNF) spezifiziert (im Anhang zu Teil B abgedruckt). Die Verwendung einer **erweiterten** BNF-Grammatik hat den Vorteil, daß der Parser dadurch effizienter wird, da viele Rekursionen durch Iterationen aufgelöst werden können. Außerdem wird die Grammatik klarer, für den Leser verständlicher und die Integration der semantischen Aktionen ist einfacher möglich.

Die EBNF-Metasprache enthält folgende **Operatoren**, die hier entsprechend ihrer Rangordnung in aufsteigender Folge aufgezählt sind:

- $x_1|x_2|..|x_n$ Alternativen: Entweder x_1 oder x_2 oder .. x_n
- $x\ y$ Konkatenation: x gefolgt von y
- $x||y$ Ein oder mehrmaliges Auftreten von x , jeweils durch y getrennt
- x^+ Ein oder mehrmaliges Auftreten von x
- x^* Kein, ein oder mehrmaliges Auftreten von x

Hierbei stehen $x, y, x_1, \dots, x_n$ für (wohlgeformte) **EBNF-Ausdrücke**, die aus Terminalen, Nichtterminalen, Klammern und den genannten Operatoren aufgebaut sind; EMPTY bezeichnet das leere Wort. Gleiche Operatoren werden von links nach rechts bearbeitet. Außerdem können Optionalklammern [und] für Teile, die weggelassen werden dürfen, und runde Klammern (und) zur Änderung der Bindungsprioritäten verwendet werden.

Zu jeder Regel $A ::= x$ der EBNF-Grammatik von CSSA gibt es eine **Parserprozedur**, die die längstmögliche Zeichenkette aus der Eingabe einliest, die sich aus dem Nichtterminal A ableiten läßt. Für die Strukturierung des Parsers stehen einige **Hilfsprozeduren** zur Verfügung, in denen sich die entwickelte Fehlerbehandlungs-

methode verbirgt. Ihre Rolle für die Fehlerbehandlung wird in 6.3 besprochen, so daß hier nur auf ihre Bedeutung für die Analyse korrekter Programmteile eingegangen zu werden braucht.

Die grundlegende Prozedur ist READ:

```
procedure READ(TOK); integer TOK;
begin
  if TOK<>TOKEN then SYNTAX_ERROR;
  NEXTSYMB
end;
```

Diese Prozedur stellt die Schnittstelle des Parsers zum Scanner dar. Sie wird vom Parser aufgerufen, wenn ein Terminalsymbol gelesen werden soll. Falls das vom Parser erwartete Symbol (TOK) mit dem vom Scanner bereits gelesenen Symbol (TOKEN) übereinstimmt, ruft READ lediglich den Scanner (Prozedur NEXTSYMB) auf, während im Fehlerfall über SYNTAX_ERROR die Fehlermeldung und -Behandlung aktiviert wird. READ wird also für **Terminalsymbole** der Grammatik gebraucht.

Es soll nun als Beispiel gezeigt werden, wie die Regel für <IF-STATEMENT> zu einer Parserprozedur umgesetzt wird:

Die Regel lautet:

```
◦ <IF-STATEMENT> ::= if <EXPRESSION> then <ACTIONS>
                        [ else <ACTIONS> ]
                        endif
```

Die zugehörige Parserprozedur:

```
procedure IF_STATEMENT;
begin
  READ(IFS);
  EXPRESSION;
  READ(THENS);
  ACTIONS;
  if OPTIONAL_PART(ELSES) then begin READ(ELSES);
                                    ACTIONS
                                end;
  READ(ENDIFS);
end;
```

Im folgenden wird eine **Abbildung PARSER** angegeben, die die Regeln einer EBNF einer LL(1)-Grammatik auf Parserprozeduren in SIMULA abbildet. Die dabei verwendeten Hilfsprozeduren OPTIONAL_PART, (LAST_)ALTERNATIVE und REPETITION testen jeweils, ob eines der als Parameter übergebenen Symbole im Eingabestrom vorliegt. Daneben erledigen sie unterschiedliche Aufgaben für die Fehlerbehandlung, was die Verwendung mehrerer verschiedener Prozeduren rechtfertigt (—> 6.3).

Die bei der Angabe der Abbildung verwendete **Metasprache**:

- $x, y, x_1, x_2, \dots, x_n$ stehen für EBNF-Ausdrücke
- A steht für ein Nichtterminalzeichen
- t steht für ein Terminalzeichen
- $\text{FIRST}\{x\} = \{ t \mid x \Rightarrow^* ty \}$
(Menge aller Terminalsymbole, die am Anfang von EBNF-Ausdrücken stehen können (incl. x selbst), die aus x abgeleitet werden können)
- SIMULA-code (aus dem Bild der Abbildung) ist **fett gedruckt**

PARSER: EBNF-REGELN $\longrightarrow$ Parserprozeduren

PARSER $\{A::=x\}$ = **procedure A; begin PARSER** $\{x\}$ **end;**
Jede Regel der EBNF wird in eine Parserprozedur übersetzt, die den Namen des (eindeutigen) Nichtterminals der linken Seite der Regel trägt.

PARSER $\{(x)\}$ = **begin PARSER** $\{x\}$ **end**
Runde Klammern werden in ein begin-end-Paar "übersetzt".

PARSER $\{x_1|x_2|..|x_n\}$ =
 if alternative $(\text{FIRST}\{x_1\})$ **then begin PARSER** $\{x_1\}$ **end else**
 if alternative $(\text{FIRST}\{x_2\})$ **then begin PARSER** $\{x_2\}$ **end else**
 :
 if last_alternative $(\text{FIRST}\{x_n\}, \text{false})$
 then begin PARSER $\{x_n\}$ **end**
 wobei $x_1, x_2, \dots, x_n$ nicht zum leeren Wort abgeleitet werden können (ansonsten gesamte Alternative in $[]$ setzen).
 Der zweite Parameter (false) von last_alternative gibt an, daß dieser Teil nicht optional ist.

PARSER $\{[x_1|x_2|..|x_n]\}$ =
 if alternative $(\text{FIRST}\{x_1\})$ **then begin PARSER** $\{x_1\}$ **end else**
 if alternative $(\text{FIRST}\{x_2\})$ **then begin PARSER** $\{x_2\}$ **end else**
 :
 if last_alternative $(\text{FIRST}\{x_n\}, \text{true})$
 then begin PARSER $\{x_n\}$ **end**
 wobei $\text{FIRST}\{x_i\}$ nicht leer.

PARSER $\{[x]\}$ = **if optional_part** $(\text{FIRST}\{x\})$ **then**
 begin PARSER $\{x\}$ **end**
 (x kann wegen LL(1)-Bedingung nicht zum leeren Wort abgeleitet werden)

PARSER $\{x\ y\}$ = **PARSER** $\{x\}$; **PARSER** $\{y\}$
 Die Konkatenation wird durch Hintereinanderschreibung realisiert.

PARSER $\{x|y\}$ = **PARSER** $\{x\}$; **PARSER** $\{(y\ x)^*\}$

```
PARSER{x+}    = PARSER{x} ; PARSER{x*}
               (x kann wegen LL(1)-Bedingung nicht zum leeren Wort abge-
               leitet werden)

PARSER{x*}    = while repetition( FIRST{x} ) do
               begin PARSER{x} end
               (x kann wegen LL(1)-Bedingung nicht zum leeren Wort abge-
               leitet werden)

PARSER{t}     = read(t)

PARSER{A}     = A
               Ein Nichtterminalsymbol wird auf den Aufruf der zugehörigen
               Parserprozedur abgebildet.

PARSER{EMPTY} =
```

Die Abbildung ist damit noch nicht eindeutig spezifiziert: Bei der Anwendung ist die oben angegebene Rangfolge der EBNF-Operatoren zu beachten (das heißt, bei mehreren Möglichkeiten die am frühesten genannte Abbildungsvorschrift zu verwenden) und Regeln für Alternativen auf die größte mögliche Zeichenfolge anzuwenden. Die Syntax von SIMULA machte es außerdem noch erforderlich, die Übergabe einer **Menge** von Symbolen (FIRST{x}) zu umgehen und Symbole der EBNF evtl. so zu modifizieren, daß korrekte SIMULA-Bezeichner (ohne spitze Klammern) und keine SIMULA-Schlüsselworte entstehen (z.B. 'ifsy' statt 'if', siehe Beispiel oben).

Die Anwendung dieser Abbildung auf die CSSA-Grammatik liefert **das Gerüst für den CSSA-Compiler**. In das entstehende Gerippe werden die semantischen Aktionen eingetragen, was in Kapitel 6.5 demonstriert wird.

6.3 Fehlerbehandlung

Im CSSA-Compiler kommt eine systematische und automatische Syntaxfehlerbehandlungsmethode zur Anwendung, deren Ziel es ist, in einem einzigen Compilerlauf möglichst **alle Fehler** zu finden und möglichst **keine Folgefehler** zu melden. Die prinzipielle Methode wird in dem Aufsatz "Ein sprachunabhängiges Verfahren zur Fehlerbehandlung für LL(1)-Sprachen", der im Anhang abgedruckt ist, an einem tabellengesteuerten LL(1)-Parser erläutert. Hier sollen lediglich die dort nicht erwähnten Konzepte und die Abweichungen, die durch die Anwendung des rekursiven Abstiegs entstehen, behandelt werden.

6.3.1 Fehlermeldungen

```

Programmer response: Probable user error. Correct the
error and rerun the job.
IBM

```

Um den Benutzer schnell und dennoch umfassend über die Fehler zu informieren, wurde das folgende **dreistufige Fehlermeldungs-konzept** realisiert:

- Markierung der Fehlerstelle in der **Programmliste**: Die Zeile, in der sich ein Fehler befindet, wird am linken Rand durch einen auffälligen Pfeil ("==>") markiert. In der Regel wird das fehlerhafte Zeichen unterstrichen; Programmteile, die der Parser überlesen hat, werden gestrichelt unterstrichen. Oft genügt daher ein Blick auf die Liste, um den Fehler zu erkennen.
- Eine **genaue Diagnose** der Fehler erfolgt im Anschluß an die Programmliste in den "parsing-diagnostics". Hier werden neben detaillierteren Fehlerhinweisen, die Aufschluß über die **Fehlerursache** geben, die Zeilennummer und das fehlerhafte Symbol genannt, um den Bezug zu den Markierungen im source-listing herzustellen.
- Die **recovery-Aktionen** des Parsers werden in den parsing-diagnostics in Form von **Korrekturannahmen** (sogenannten "assumptions") gemeldet. Sie geben Aufschluß darüber, in welcher Form der Parser das fehlerhafte Programm (virtuell) "repariert" hat, um es weiteranalysieren zu können. Im allgemeinen wird es nicht nötig sein, sich mit den Korrekturannahmen zu befassen. Lediglich in sehr komplizierten Fällen und scheinbar unerklärlichen [Folge-]Fehlern wird man bemüht sein, sich die Sicht des Parsers zu eigen zu machen.

Neben dieser dreistufigen Diagnosehierarchie wurde das **Layout** des source-listings und der parsing-diagnostics speziell dahingehend konzipiert, um dem Anwender ein Maximum an Information in übersichtlicher Form zu geben und damit den Prozeß der Fehlersuche zu beschleunigen. (Struct-option des Compilers, Hervorhebung von Schlüsselwörtern gegenüber Bezeichnern, Markierung der Blockverschachtelung am rechten Rand, cross-reference Liste mit Attributen etc. —> 7.2)

Das folgende kleine **Beispiel** zeigt typische Fehler und die zugehörigen Meldungen des Compilers:

```

BMS-CSSA-COMPILER                      1981/11/25  11:15:05  PAGE 1

==> 1  type FEHLERDEMO is ;
      2  script
      3
==> 4      var int: I1 ; var int I2, I3,I4 I5;
      5
      6      facet F1 is
      7
==> 8          operation IDEL is
==> 9              const int: I28:=6*9+ * 17;
      10
      11          loop until FOUND do
      12              if I1>7 then print ("JA")
==> 13                  else print ("NEIN");
==> 14          endloop;
      15      endoperation
      16
      17      endfacet
      18      initial F1
      19  endscript

BMS-CSSA-COMPILER - RELEASE 30 SEP 1981   8 ERRORS DETECTED
END OF COMPILING ON 81/11/25 AT 11:15:10.32  RETURNCODE = 7
COMPILE-TIME (CPU) = 0.73 SEC.  EXECUTION-TIME = 5.41 SEC
NUMBER OF SOURCE-LINES READ = 19  NUMBER OF TOKENS = 65

```

Offenbar wurden folgende **Fehler** gemacht: In Zeile 1 wurde versehentlich ein Semikolon nach 'is' geschrieben; in Zeile 4 fehlt der Doppelpunkt vor I2 und ein Komma zwischen I4 und I5; in Zeile 8 wurde 'IDEL' statt 'IDLE' geschrieben; der arithmetische Ausdruck in Zeile 9 ist fehlerhaft; am Ende von Zeile 12 fehlt ein Semikolon; das 'endif' wurde vergessen; der exit 'found' wurde nicht definiert.

Nach dem Programmlisting erscheinen die **Fehlermeldungen**. Es sollen hier nicht alle Meldungen aufgezählt werden, einige beispielhafte mögen genügen:

```

==> SYNTAX-ERROR IN LINE 1 : ILLEGAL SYMBOL ';' IN <PROGRAM>
    CHECK, IF 'SCRIPT' SHOULD HAVE BEEN USED INSTEAD

==> SYNTAX-ERROR IN LINE 4 : ILLEGAL SYMBOL 'I5' IN <LOCAL-DECL-
    LIST>
    CHECK, IF ';' | ':' | '-' | '=' | ',' SHOULD HAVE BEEN USED INSTEAD

==> SYNTAX-ERROR IN LINE 8 : ILLEGAL SYMBOL 'IDEL' IN
    <OPERATION-DECL>
    CHECK, IF <LOCAL IDENTIFIER> | 'IDLE' SHOULD HAVE BEEN USED
    INSTEAD

- ASSUMPTION IN LINE 8 : 'IDEL' ASSUMED TO BE A MISSPELLED

```

'IDLE'

=> SYNTAX-ERROR IN LINE 14 : ILLEGAL SYMBOL 'ENDLOOP' IN <IF-STATEMENT> STARTING IN LINE 12 ...

- ASSUMPTION IN LINE 14 : 'ENDIF' ASSUMED TO BE MISSING ...

- ASSUMPTION IN LINE 14 : 'ENDLOOP' ASSUMED TO BE PART OF <LOOP-STATEMENT> STARTING IN LINE 11

=> ERROR BMS038 IN LINE 14 (BEFORE ';') : EXIT FOUND IS STILL UNDEFINED

Die erste Fehlermeldung zu Zeile 4 und die Meldungen zu den Zeilen 9 und 13 wurden hier weggelassen. Die assumption-Meldungen, die Aufschluß über das recovery geben, wurden ebenfalls bis auf die Meldung zu Zeile 14 weggelassen. Bei allen Fehlern, bis auf den letzten, handelt es sich um Syntaxfehler. In diesem Beispiel hat sich der Parser stets sehr schnell wieder gefangen, es wurden keine Programmteile überlesen.

6.3.2 Fehlerbehandlungsmethoden

Die Syntaxfehlerbehandlung geschieht in systematischer Weise. Allerdings kommen auch einige **Heuristiken** zur Anwendung. Diese Heuristiken sind aber weitgehend sprachunabhängig und können für jede ALGOL-artige Sprache verwendet werden. Einige dieser Heuristiken sollen hier erläutert werden :

- **Tippfehlerkorrektur:** Damit soll versucht werden, Schreib- und Tippfehler bei Bezeichnern und Schlüsselwörtern abzufangen. Das fehlerhafte Symbol wird mit der Liste der erwarteten Symbole verglichen; ein unbekannter Bezeichner wird mit der Liste aller sichtbaren deklarierten Bezeichner verglichen. Erkannt werden u.a. Tippfehler, die darauf beruhen, daß ein Zeichen fehlt, zuviel vorhanden ist oder falsch getippt wurde oder zwei benachbarte Zeichen vertauscht sind. Nach [GRI71] fallen 80% der Tippfehler in eine dieser Kategorien.
- **Klassifizierung von Bezeichnern:** Bezeichner werden in drei Klassen eingeteilt: Neue, globale und lokale Bezeichner. An die Stelle einer semantischen Überprüfung im Parser tritt hier eine vorgezogene Symboltabellensuche des Scanners. Der Scanner meldet also direkt, von welcher Art ein gelesener Bezeichner ist. Dadurch können einige Fehler der statischen Semantik – wie vergessene Deklaration eines Bezeichners – wie Syntaxfehler systematisch und automatisch entdeckt und gemeldet werden. Außerdem können so einige Fehler auch früher erkannt und somit besser behandelt werden. Beispielsweise führt 'iff A=0 then ...' nicht dazu, daß fälschlicherweise eine Zuweisung analysiert wird, bei der erst ein Fehler bei 'A' auftritt (':=' expected), sondern daß an dem (in der Regel) nicht deklarierten Bezeichner 'iff' erkannt wird, daß ein Fehler vorliegt. (Statements können nicht mit undeklarierten Bezeichnern beginnen!) Hier

kann die Tippfehlerkorrektur 'iff' in das Schlüsselwort 'if' abändern und ein if-Statement kann analysiert werden - wozu es im ersten Fall zu spät gewesen wäre (dann würde sich der Parser bereits auf dem falschen Pfad befinden).

- **Kostenrechnung:** Hiermit wird gesteuert, ob ein Symbol überlesen werden soll oder ob mit einem Symbol recovery versucht werden soll. Damit kann pragmatisch auf typisch menschliche Schwächen und Eigenschaften eingegangen werden: So sind bspw. Fehler in der Interpunktion häufig; dagegen stehen längere Schlüsselwörter kaum "zufällig" in der Programmlandschaft...
- **Heuristischer Error-Parser (HEP):** Mit seiner Hilfe können deplazierte syntaktische Einheiten analysiert werden, sofern sie mit einem eindeutigen Symbol beginnen. Der HEP erlaubt es, auch dicht aufeinanderfolgende Syntaxfehler zu entdecken, da er versucht, möglichst große Teile (losgelöst vom eigentlichen Parsingprozess) von demjenigen Programmteil zu analysieren, der zwischen der Fehlerstelle und dem Wiedereinstiegspunkt liegt. Der HEP kann Syntaxfehler entdecken und melden und danach sich selbst rekursiv aufrufen.

Syntaxfehlermeldungen werden vom Parser **automatisch generiert**, sobald in der Eingabe ein token entdeckt wird, der nicht erwartet wurde und der dazu führt, daß der bisher analysierte (und nach evtl. vorangegangenen Syntaxfehlern bereits "virtuell reparierte") Programmteil keinen korrekten Präfix eines CSSA-Programms darstellt.

Nachdem der Parser auf einen Syntaxfehler gelaufen ist, wird mit einem der nächsten token **recovery** versucht. Ob recovery mit einem erwarteten token durchgeführt wird, hängt von der oben erwähnten Kostenrechnung ab. Falls kein recovery durchgeführt wird, so wird bei einem token, der eindeutig eine syntaktische Einheit einleitet, der HEP-Parser aufgerufen, ansonsten wird der token überlesen (und im source-listing gestrichelt unterstrichen).

6.3.3 Anpassung an den rekursiven Abstieg

Wie aus der Beschreibung der verwendeten Fehlerbehandlungsmethode im Anhang hervorgeht, werden häufig Informationen aus dem parsing-stack benötigt. Bei einem Parser nach dem rekursiven Abstieg gibt es keinen explizit zugreifbaren parsing-stack; vielmehr liegen hier alle Informationen im - für den Programmierer nicht zugänglichen - Laufzeitkeller bzw. in der aktuellen Verschachtelung der Parserprozeduren. Deshalb wurde im vorliegenden Compiler eine Methode entwickelt, um doch einen expliziten Keller (den sog. recovery-stack) zu verwalten, in dem stets die Terminalsymbole gespeichert sind, die der Parser erwartet, das heißt, die mit Sicherheit noch kommen müssen, um die bisher gelesene Eingabe zu einem richtigen Programm zu ergänzen, oder deren Vorhandensein noch mit Sicherheit abgetestet wird. Dazu wird jede Parserprozedur (und jeder Zweig innerhalb von Prozeduren) **zweimal** durchlaufen. Dies wird durch sogenannte

Metaklammer-Prozeduren PROCSTART und PROCEND am Anfang und Ende jeder Parserprozedur (bzw. LEFT_META_BRACKET und RIGHT_META_BRACKET für Zweige) erreicht. (Diese Prozeduren hätten unproblematisch in die in 6.2 angegebene Abbildung integriert werden können, wurden aber der Übersichtlichkeit halber weggelassen.) Dadurch befindet sich der Parser immer in einer von **drei Phasen**:

- In der **expect-Phase** werden alle Terminalzeichen, die eigentlich abgetestet werden sollen, als erwartet in den recovery-stack eingetragen. Die Hilfsprozeduren liefern dabei immer FALSE, damit Zweige, die in der späteren parsing-Phase nicht unbedingt betreten werden, in dieser Phase auch nicht durchlaufen werden.
- Am Ende des Parser-Abschnitts, in der die expect-Phase begann, wird auf die **parsing-Phase** umgeschaltet und an den Anfang des selben Parser-Abschnitts gesprungen. Dann wird solange Programmtext eingelesen, analysiert und code generiert, bis ein durch eine Abfrage geschützter Abschnitt betreten wird. Dort wird wieder auf die expect-Phase umgeschaltet, da nun weitere Terminalzeichen als Korrekturaufsetzpunkte in den stack eingetragen werden können.
- Bei einem Syntaxfehler versuchen die Fehlerbehandlungsroutinen den Fehler virtuell zu reparieren. Ist es ihnen gelungen, so wird zum Parser zurückgekehrt und der Parser wird in der **recovery-Phase** durchlaufen, bis die Einstiegsstelle erreicht wird; anschließend kann im normalen Zyklus fortgefahren werden.

Durch dieses Verfahren, das in Teil E Kapitel 2.2 und 2.5 detailliert beschrieben wird, konnte die entwickelte Fehlerbehandlungsmethode in den CSSA-Compiler integriert werden, ohne daß auf die Vorteile, die der rekursive Abstieg für die Strukturierung vor allem der Codegenerierung bietet, verzichtet werden mußte.

6.4 Semantische Überprüfungen

Neben der syntaktischen Korrektheit von CSSA-Programmen überprüft der Compiler auch die Einhaltung einer Reihe **nicht kontextfreier** Restriktionen. Viele dieser kontextabhängigen Regeln lassen sich mit Hilfe von **Attributen** deklarierter Bezeichner formulieren. Beispielsweise muß hinter 'call' der Bezeichner für eine Prozedur genannt werden, bei Zuweisungen muß **Typgleichheit** bestehen und aktuelle und formale Parameterlisten müssen kompatibel sein. Manche Konstrukte dürfen nur in einem bestimmten **Kontext** verwendet werden, beispielsweise darf ein return-Statement nur innerhalb einer Funktion stehen. Schließlich muß der Compiler auch die Einhaltung von **Implementierungsrestriktionen** überprüfen, wie bspw. die Maximalzahl von Dimensionen eines arrays. Eine vollständige Liste aller Fehler der statischen Semantik findet sich in Teil C.

This manual omitted any catalog of errors, since the author expected that the compiler would describe the errors in satisfactory detail.

D.N. Freeman, 1964

Während die Spezifikation und Überprüfung der Syntax bei dem heutigen Stand der Compilerbautechnik vollständig **systematisiert, formalisiert** und **automatisiert** ist, gibt es auf dem Gebiet der statischen Semantik zwar einige vielversprechende Ansätze (bspw. attributierte Grammatiken), sie haben aber noch nicht im gleichen Maße in die praktische Compilerbautechnik Eingang gefunden.

In dem hier beschriebenen Compiler wurde zwar keine der formalen Methoden verwendet; dennoch ist es gelungen, den größten Teil der semantischen Restriktionen durch eine detaillierte **Klassifikation aller Bezeichner** zu systematisieren. Die meisten Überprüfungen können dadurch einfach und schematisch vorgenommen werden, indem bei der Deklaration einem Bezeichners eine Reihe von Attributen zugeordnet werden, die dann bei seiner Verwendung überprüft werden. Ob ein Bezeichner überhaupt deklariert wurde, wird sogar bereits durch die **syntaktische** Analyse überprüft (→ 6.3). Der Scanner liefert dazu verschiedene Token, je nachdem ob er einen deklarierten oder undeklarierten Bezeichner gefunden hat.

Eine ausführliche Beschreibung des zur Klassifizierung von Bezeichnern verwendeten Typkonzepts findet sich in Teil B Kapitel 4; hier soll ein kurzer Überblick genügen.

Durch Angabe von **Typklasse**, **Datentyp** und **Subtyp** ist eine Klassifikation aller Bezeichner möglich:

- Eine **Typklasse** wird **jedem** Bezeichner bei seiner Deklaration zugeordnet; sie gibt an, in welchen Konstrukten ein angewandtes Auftreten des Bezeichner erlaubt ist. Beispielsweise kann nur Variablen ein Wert zugewiesen werden, können nur Prozeduren mit 'call' aufgerufen werden und nur Schleifen mit 'escape' verlassen werden.
- Einige Typklassen (z.B. PROCEDURE, FACET, LOOP) werden nicht weiter unterteilt, während die Bezeichner aus anderen Klassen (z.B. VARIABLE, CONSTANT, FUNCTION) noch genauer nach ihrem **Datentyp** eingeteilt werden. Bei Variablen wird der Datentyp der speicherbaren Werte angegeben, während bei Funktionen der Datentyp des berechneten Ergebnisses angegeben wird.
- Neben den **einfachen Datentypen** wie BOOL, INT, REAL oder STRING gibt es auch **benutzerdefinierte Datentypen** wie RECORD, ARRAY, SET oder ENUM. Bei Bezeichnern dieser Datentypen wird in der Deklaration nicht das Schlüsselwort für

einen vordefinierten Typ angeben, sondern ein Bezeichner der Klasse **USER-DEFINED-TYPE**. Als Typangabe wird daher nicht nur der Datentyp vermerkt, sondern zusätzlich wird als **Subtyp** ein Verweis auf den in der Deklaration verwendeten Typbezeichner eingetragen, und damit auch ein **Verweis auf die Typdeklaration**, in der der Aufbau des benutzerdefinierten Datentyps angegeben wurde.

Für die **Typgleichheit** bei benutzerdefinierten Datentypen wird nun auch die Übereinstimmung des Subtyps gefordert; Strukturgleichheit genügt nicht! Folglich braucht der Compiler auch keine komplizierten Vergleiche durchzuführen, um die Typgleichheit zu überprüfen: Der simple Vergleich von Typ und Subtyp reicht stets aus.

Selbst bei der Parameterübergabe **laufzeitabhängiger arrays** genügt der einfache Typvergleich: Zwar wird die Deklaration des arrays-Typs erst zur Laufzeit evaluiert, der Compiler überprüft aber, ob bei der Deklaration von aktuellem und formalem Parameter auf die selbe Typdeklaration Bezug genommen wurde, und stellt so sicher, daß zur Laufzeit beide arrays strukturgleich sind.

Durch Eintragen und Überprüfen der verschiedenen Attribute kann der größte Teil der statischen Semantik in systematischer Weise überprüft werden; auch die Typkompatibilität innerhalb von Ausdrücken läßt sich mit Hilfe einfacher Attributvergleiche sicherstellen. In einigen anderen Fällen sind aber auch aufwendigere Maßnahmen notwendig: Zum Vergleich von Parameterlisten wird zu Prozeduren und ähnlichen Programm-Modulen die Liste aller formalen Parameter abgespeichert, beim Aufruf werden die aktuellen Parameter mit dieser Liste verglichen. Bei Verwendung von arrays muß entsprechend getestet werden, ob die Indizes in Anzahl und Typ passen. Um feststellen zu können, ob beispielsweise 'return' wie gefordert innerhalb einer Funktion aufgerufen wird, wird die statische Verschachtelung von Funktionen, Prozeduren, Schleifen, Facetten und scripts mitgeführt und bei Statements, die nur in einem gewissen Kontext auftreten können, konsultiert. Bei records wird eine Liste aller Felder abgespeichert, und bei Verwendung der Punktnotation wird geprüft, ob das angesprochene Feld überhaupt existiert.

Die Analyse der statischen Semantik geschieht völlig getrennt von der Codegenerierung. Während die Codegenerierung sofort beim ersten Fehler abgeschaltet werden kann, soll die Analyse der statischen Semantik auch nach vorhergehenden Syntax- oder Semantikfehlern fortgesetzt werden, damit möglichst alle Fehler in einem Compilerlauf entdeckt werden. Dadurch besteht allerdings die Gefahr, daß Syntaxfehler unter Umständen **semantische Folgefehler** nach sich ziehen. Bspw. kann ein Syntaxfehler bei der Deklaration eines Bezeichners dazu führen, daß nicht alle Attribute ordnungsgemäß eingetragen werden. Für den hier beschriebenen Compiler wurden daher einige Methoden zur Vermeidung semantischer Folgefehler entwickelt.

Um zu verhindern, daß ein nicht deklarierter Bezeichner bei jedem Auftreten erneut zu einer Fehlermeldung führt, wird das sogenannte **"Chamäleon-Prinzip"** verwendet: Beim ersten Auffinden eines nicht deklarierten Bezeichners wird er als "Chamäleon", also ohne Angabe von Typklasse oder Datentyp, in die Variablentabelle eingetragen. Die Prozeduren zur Überprüfung von Attributen sind so konzipiert, daß zu einem solchen "Chamäleon" niemals eine Fehlermeldung erzeugt wird.

Die globale Größe TBLREF des Compilers verweist normalerweise auf den zuletzt angesprochenen Bezeichner in der Variablentabelle. Im Falle eines Syntaxfehlers hingegen wird TBLREF ebenfalls auf ein solches "Chamäleon" (das aber nicht in die Variablentabelle eingetragen wird) gesetzt. Die Erfahrung hat nämlich gezeigt, daß nach einem Syntaxfehler und erfolgreichem recovery der TBLREF-Verweis oft nicht mehr korrekt ist. Auf diese Weise wird verhindert, daß nach Syntaxfehlern Attribute falschen Bezeichnern zugeordnet werden oder die Attribute falscher Bezeichner bei semantischen Überprüfungen getestet werden. In dem folgenden Beispiel würde andernfalls nach dem virtuellen Einfügen eines Bezeichners hinter 'procedure' und dem Wiederaufsetzen bei '(' der Variablen I die Typklasse PROCEDURE zugeordnet:

```
... var int: I; procedure ( real: R) is ...
```

So aber wird das Attribut einem "Chamäleon" zugeordnet, das später nicht mehr zugreifbar ist.

Falls beim angewandten Auftreten eines "Chamäleons" Attribute eindeutig aus dem Zusammenhang hervorgehen, so werden sie noch nachträglich dem Bezeichner zugeordnet. Auf diese Weise kann überprüft werden, ob der nicht oder fehlerhaft deklarierte Bezeichner zumindest konsistent verwendet wird. Ergibt bspw. die Überprüfung des Statements 'call P', daß es sich bei P um ein "Chamäleon" handelt, so wird die Typklasse PROCEDURE nachgetragen. Diese Maßnahme dient zum Erkennen möglichst vieler Fehler.

Im Zusammenhang mit **komplexen Datenstrukturen** sind eine Reihe von semantischen Folgefehlern denkbar. Betrachten wir als Beispiel das Statement 'R.F:=GELB;' und nehmen wir an, daß R nicht deklariert wurde. Im ungünstigsten Fall würde der Compiler dann melden:

- ◊ R ist nicht deklariert
- ◊ R ist kein record
- ◊ F ist kein Feld von R
- ◊ F ist nicht vom Typ ENUM
- ◊ F ist nicht vom Subtyp FARBE

Ähnliche Folgefehler sind bei dem Statement 'A(3):=GELB;' denkbar, wenn A nicht deklariert wurde:

- ◊ A ist nicht deklariert

- ◊ A ist kein array
- ◊ Die Anzahl der Indizes stimmt nicht
- ◊ Der Index darf nicht vom Typ INT sein
- ◊ Der Basistyp von A ist nicht ENUM
- ◊ A hat nicht den Subtyp FARBE

Zur Vermeidung solcher den Benutzer nur verwirrenden **Fehlerkaskaden** werden sofort alle weiteren Überprüfungen zu dem Bezeichner an dieser Stelle fallengelassen, sobald der erste Fehler in solch einer Sequenz auftritt.

In einigen Fällen sind nach einer syntaktischen Reparatur auch entsprechende **semantische Reparaturen** nötig: Wird bspw. ein 'end' durch die Syntaxfehlerbehandlung eingefügt, so muß entsprechend auch das Verlassen eines Blockes registriert werden und die Bezeichner des obersten Blocks müssen aus der Symboltabelle entfernt werden. Solche semantische Reparaturen ergeben sich aber bei der hier verwendeten Syntaxfehlerbehandlungsmethode automatisch (—> 6.3).

Es wurde gezeigt, daß syntaktische Fehler in einigen Fällen semantische Folgefehler nach sich ziehen können. Es sei an dieser Stelle darauf hingewiesen, daß der umgekehrte Fall in dem hier beschriebenen Compiler niemals auftreten kann: Da das Parsing rein **syntaxgesteuert** arbeitet, also niemals Attribute von Bezeichnern die Analyse beeinflussen, können Fehler der statischen Semantik niemals syntaktische Folgefehler auslösen.

6.5 Codegenerierung

Der CSSA-Compiler ist als ein **one-pass Compiler** konzipiert: Sofort nach dem Erkennen einer syntaktischen Substruktur wird der zugehörige Code generiert. Dementsprechend sind in das durch den Parser vorgegebene Gerüst Aufrufe von **Codegenerierungsroutinen** eingestreut. Die Codegenerierung verläuft i.w. **syntaxgesteuert**, aber auch Informationen der Symboltabelle, globale Größen, sowie lokale Größen in den Parserprozeduren beeinflussen den letztlich generierten Code.

Um die Codegenerierung klar von der syntaktischen und semantischen Analyse zu **trennen**, sind alle Routinen und globalen Größen (z.B. Zähler) der Codegenerierung in einer SIMULA-CLASS zusammengefaßt. Die Aufrufe der Routinen im Parser sind deutlich durch 'INSPECT CODE_GENERATOR DO' gekennzeichnet. Neben dem rein optischen Eindruck hat das auch den Vorteil, daß im Falle eines Fehlers im CSSA-Programm die Codegenerierung leicht **abgeschaltet** werden kann (durch CODE_GENERATOR:-NONE). Bei der Codegenerierung kann also stets davon ausgegangen werden, daß alle Informationen über das bisher analysierte Programm (z.B. Attribute in der Symboltabelle) vollständig und konsistent vorhanden sind. Das bedeutet eine beachtliche programmiertechnische Erleichterung: Eine Reihe von Abfragen zur Vermeidung interner Fehler

können entfallen.

Der Aufwand für die Programmierung ist naturgemäß besonders gering, wenn jeweils sofort nach dem Erkennen eines CSSA-Konstruktes der entsprechende code in der Zielsprache ausgegeben wird. Das ist allerdings nicht ohne weiteres möglich, da die Struktur des generierten codes nicht genau der des Quellprogramms entspricht: Beispielsweise werden verschachtelte Blöcke (Prozeduren, Funktionen, ...) in eine lineare Folge disjunkter (nicht verschachtelter) Codeabschnitte übersetzt, die jeweils mit einer Sprungmarke beginnen und mit einer Verzweigung (in der Regel Rückkehr zur Aufrufstelle) enden. Eine mögliche (aber recht aufwendige) Lösung bestände darin, Codezeilen im Programm zunächst zwischenspeichern und erst später auszugeben. Stattdessen wurde aber hier eine andere Lösung gewählt: Alle Codezeilen werden sofort ausgegeben, aber mit einer 8-stelligen Numerierung versehen, nach der sie im Anschluß an den Compilerlauf **sortiert** werden.

Für die Numerierung der Codezeilen werden die drei globalen Größen `C_SCRIPT_NR`, `C_BLOCK_NR` und `CODE_SECTION` geführt, so daß bei Ausgabe einer Zeile (durch die Prozedur `NEW_LINE`) eine 3-stufige Kennung angehängt werden kann. Darüberhinaus kann mit Hilfe der parametrisierten Prozedur `PUT_LINE` unabhängig von den globalen Größen eine Zeile mit einer bestimmten Kennung ausgegeben werden.

Mit Hilfe der 3-stufigen Numerierung wird die durch den Quellcode vorgegebene Reihenfolge in folgenden Fällen durchbrochen:

- `C_SCRIPT_NR` gibt stets an, das wievielte script gerade übersetzt wird, und hat bei der Numerierung die höchste Signifikanz. Dadurch werden die im Quellcode möglicherweise verschachtelten scripts in eine lineare Folge von Moduln übersetzt.
- `C_BLOCK_NR` gibt stets die eindeutige Nummer des Blocks an, der gerade übersetzt wird, und macht den zweiten Teil der Numerierung aus. Auf diese Weise wird die Verschachtelung der Blöcke innerhalb eines scripts aufgehoben. Die Blöcke werden in disjunkte Abschnitte des script-Moduls übersetzt. Außerdem wird so das Problem gelöst, daß im Quellcode bei Vorwärtsdeklarationen Kopf und Rumpf der Routine getrennt stehen: Beide Teile können getrennt übersetzt werden, werden später aber bei der Sortierung "zusammengeschweißt", da die Blocknummer übereinstimmt.
- Der code innerhalb von Blöcken kann in einigen Fällen ebenfalls nicht chronologisch in der Reihenfolge generiert werden, in der er später stehen soll. Beispielsweise ist dem Compiler erst **am Ende** der lokalen Deklarationsliste bekannt, wie groß der entsprechende Aktivierungssatz sein muß; der Aktivierungssatz muß (im generierten code) aber schon **vor** den Initialisierungen der lokalen Variablen angelegt werden. Aus diesem Grunde wurde als dritte Stufe der Numerierung noch die `CODE_SECTION` aufgenommen, das ist ein einzelner Buchstabe, der den verschiedenen Abschnitten ei-

nes Blocks zugeordnet ist. In der Regel gibt es folgende Abschnitte:

- A: Betreten des Blocks
- D: Anlegen des Aktivierungssatzes
- I: Initialisierungen
- S: Statements
- Z: Vernichten des AS + Rückkehr zur Aufrufstelle

Innerhalb der einzelnen Abschnitte können mehrere Zeilen mit der gleichen Kennung stehen; da die Reihenfolge dieser Zeilen nicht verändert werden darf, muß ein **stabiles** Sortierverfahren verwendet werden.

Zum Vergleich sei hier darauf hingewiesen, daß das genannte Problem in PASCAL 8000 durch eine Spracheinschränkung gelöst wird: Initialisierungen dürfen nicht direkt bei den Deklarationen der Variablen stehen, sondern müssen allesamt **nach** den Variablendeklarationen (hinter INITIALVALUES) vorgenommen werden.

- Alle vom Compiler im Laufe der Übersetzung generierten Kontrollblöcke (z.B. für Aktivierungssätze) sollen im generierten Code an zentraler Stelle gesammelt werden. Sie werden daher durch PUT_LINE mit der Scriptnummer 0 ausgegeben. Da in CSSA – im Gegensatz zu PASCAL – Prozedur- und Variablendeklarationen gemischt werden dürfen, sind außerdem möglicherweise im Quellcode die Variablenlisten verschachtelt. Daher wird bei der Ausgabe der einzelnen Zeilen eines Kontrollblocks auch die C_BLOCK_NR ausgegeben und so beim Sortieren die Verschachtelung aufgelöst.

Da der generierte Code zusammen mit dem Simulationssystem ausgeführt werden soll, ist die **Zielsprache** des CSSA-Compilers SIMULA. Im Hinblick auf eine geplante Implementierung auf einem Mikroprozessorsystem sollte allerdings die Möglichkeit offen gehalten werden, den Compiler ohne allzu großen Aufwand auf eine andere Zielsprache umzustellen. Aus diesem Grunde geschieht die Codegenerierung über eine **sprachunabhängigen, aber maschinennahen Zwischencode** für eine **virtuelle Stackmaschine**.

Allerdings wird der Zwischencode **nicht** vom Compiler ausgegeben und anschließend **interpretiert**: Lediglich die Folge der Aufrufe der Codegenerierungsroutinen der untersten Schicht spiegelt das Programm in Zwischensprache wieder. Die Routinen entsprechen dem Befehlssatz der virtuellen Maschine. Sie geben SIMULA-Code aus, der auf den Datenstrukturen des Laufzeitsystems operiert und auch dort definierte Prozeduren verwendet, und konstituieren so die virtuelle Stackmaschine.

In der Schicht 2 der Codegenerierungsroutinen sind Folgen von Aufrufen der Schicht 1 zu höheren, abstrakteren Konstrukten zusammengefaßt, es wird aber dort kein Code direkt ausgegeben. Daher müssen bei der Umstellung auf eine andere Zielsprache nur die Routinen der Schicht 1 geändert werden, also die virtuelle Maschine neu implementiert werden.

Es folgt die Parseroutine für ein if-Statement, die gegenüber Kapitel 6.2 um die Überprüfung der statischen Semantik und die Codegenerierung erweitert wurde:


```
procedure IF_STATEMENT;  
begin integer MY_NR;  
  READ(IFS);  
  EXPRESSION;  
  inspect STATIC_SEM do COMPARE_TYPES(RES_TYPE,"BOOL");  
  inspect CODE_GENERATOR do  
    begin MY_NR:=NR_OF_IF:=NR_OF_IF+1;  
      IF_HEAD(MY_NR);  
    end;  
  READ(THENS);  
  ACTIONS;  
  inspect CODE_GENERATOR do IF_MIDDLE(MY_NR);  
  if OPTIONAL_PART(ELSES) then begin READ(ELSES);  
    ACTIONS  
  end;  
  READ(ENDIF);  
  inspect CODE_GENERATOR do IF_TAIL(MY_NR);  
end;
```

Anhand des folgenden Beispiels, das zeigt wie ein if-Statement übersetzt wird, soll das Prinzip der **Codegenerierung über Zwischensprachschichten** demonstriert werden. Die Abbildung zeigt die Aufrufstruktur der Parserprozeduren (in spitzen Klammern) und der vom Parser aufgerufenen Codegenerierungsroutinen (Schicht 2-Routinen sind durch '*' gekennzeichnet), sowie den letztlich generierten code (eingerahmt). Das Einlesen eines Terminalsymbols durch die Prozedur READ wird hier jeweils durch das Symbol in Hochkommas angedeutet. Das übersetzte CSSA-Statement lautet:

```
if I<10 then I:=I*5; endif
```


<IF-STATEMENT>

"IF"

<BOOL-EXPRESSION>

<DISJUNCTION>

<KONJUNCTION>

<BOOL-ELEMENT>

<SIMPLE-EXPRESSION>

<TERM>

<FACTOR>

"I"

push(-->I) ==> INT_PUSH(AKT_AS.VAR118);

<">

<SIMPLE-EXPRESSION>

<TERM>

<FACTOR>

"10"

push_int(10) ==> INT_PUSH(10);

compare("INT", "<")

==> BOOL_PUSH(INT_TOP.NEXT.WERT<INT_TOP.WERT);
INT_POP; INT_POP;

*if_head(5)

log_not

==> LOG_NOT;

branch_on_true("IF_ELSE", 5)

==> BRANCH_FLAG:=BOOL_TOP.WERT; BOOL_POP;
IF BRANCH_FLAG THEN GOTO IF_ELSE5;

"THEN"

<ACTIONS>

<STATEMENT-LIST>

<STATEMENT>

<ASSIGNMENT>

"I"

":="

<EXPRESSION>

<TERM>

<FACTOR>

"I"

push(-->I) ==> INT_PUSH(AKT_AS.VAR118);

"X"

<FACTOR>

"5"

push_int(5) ==> INT_PUSH(5);

mult

==> MULT;

assign(-->I)

==> AKT_AS.VAR118:=INT_TOP.WERT; INT_POP;

";"

*if_middle(5)

branch("IF_SKIP", 5) ==> GOTO IF_SKIP5;

put_label("IF_ELSE", 5) ==> IF_ELSE5:

"ENDIF"

*if_tail(5)

put_label("IF_SKIP", 5) ==> IF_SKIP5:

In den Übersetzungsvorgang gehen auch einige **Informationen über den Kontext** ein: Beispielsweise soll '-->I' den Verweis auf den I zugeordneten Symboltabelleneintrag bezeichnen; dort wird der Datentyp und die Position im Aktivierungssatz nachgesehen. Die Prozedur <IF-STATEMENT> erhöht einen globalen Zähler für if-Statements und kopiert ihn auf eine lokale Variable. Auf diese Weise wird diesem if-Statement 5 als eindeutige Nummer zugeordnet, die bei allen Teilen des generierten Codes verwendet wird. Durch die Verwendung einer **lokalen** Variable bleibt auch bei verschachtelten if-Statements die Numerierung korrekt. Ebenfalls mit Hilfe **lokaler Variablen** wird festgehalten, welcher Vergleichsoperator verwendet wurde und welches die Zielvariable der Zuweisung war. Dieser Vorteil des rekursiven Abstiegs konnte im vorliegenden Compiler recht häufig verwendet werden; bei einem tabellengesteuerten Parser wäre die Verwendung **expliziter** stacks notwendig gewesen. In einigen Fällen werden auch globale Variablen verwendet: <FACTOR> vermerkt INT als gefundenen Datentyp und <BOOL_ELEMENT> übergibt ihn an die compare-Routine.

Auf die **dynamische Speicherplatzzuweisung** für Variablen mit Hilfe eines **Laufzeitkellers** wird im nächsten Abschnitt näher eingegangen. Hier wurde angenommen, daß I eine lokale Variable ist, also zur Laufzeit im aktuellen Aktivierungssatz (AKT_AS) zu finden ist.

Für die Abarbeitung von **Ausdrücken** besitzt die virtuelle Maschine pro Datentyp einen **stack**, in dem Werte abgespeichert werden können. Ausdrücke werden in eine **Postfixnotation** übersetzt: Bei Konstanten und Variablen wird ein Wert gepusht. Die Verknüpfungen ersetzen die beiden obersten Elemente durch das Ergebnis. Die grundlegende Konvention ist, das nach Abarbeitung eines Ausdrucks das Ergebnis im entsprechenden stack steht.

Die Folge der Aufrufe der Routinen der Schicht 1 stellt den **Zwischencode** dar, der hier nochmals gezeigt wird:

```

push(-->I);
push_int(10);
compare("INT","<");

log_not;
branch_on_true("IF_ELSE",5);      ] if_head(5);

push(-->I);
push_int(5);
mult;
assign(-->I);

branch("IF_SKIP",5);
put_label("IF_ELSE",5);          ] if_middle(5);

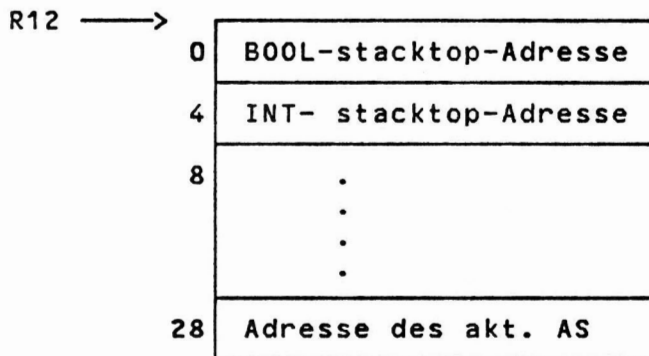
put_label("IF_SKIP",5);          — if_tail(5);

```

Es soll hier einmal exemplarisch gezeigt werden, wie mit geänderten Routinen der Schicht 1 der Zwischencode in **IBM-Assembler** übersetzt werden kann:

L	3,4(12)	INT-stacktop Adr. -->R3	push(-->I)
LA	3,4(3)	stacktop-Adr. erhöhen	
ST	3,4(12)	stacktop-Adr. abspeichern	
L	4,28(12)	Adr. akt. AS nach R4	
L	4,<offset>(4)	Wert der Var. nach R4	
ST	4,0(3)	Wert auf den stack	
L	3,4(12)	s.o.	push_int(10)
LA	3,4(3)	s.o.	
ST	3,4(12)	s.o.	
BALR	4,0	Adr. nächster Befehl	
B	8(4)	umspringe Konstante	
DC	F'<10>'	INT-Konstante nach R4	
L	4,4(4)	lade Konstante nach R4	
ST	4,0(3)	Wert auf den stack	
L	3,4(12)	INT-stacktop Adr. -->R3	compare(INT,<)
LA	4,8	Konstante 8 nach R4	
SR	3,4	R3--> vorl. Kellerelement	
ST	3,4(12)	stacktop-Adr. zurücks.	
L	4,8(3)	ehemaliges stacktop --> R4	
L	3,4(3)	Wert 1. Operand nach R3	
CR	3,4	Vergleich	
L	3,0(12)	BOOL-stacktop-Adr. --> R3	
LA	3,1(3)	stacktop-Adr. erhöhen	
ST	3,0(12)	stacktop-Adr. abspeichern	
MVI	0(3),X'FF'	TRUE in den BOOL-stack	
BALR	4,0	Adr. nächster Befehl -->R4	
BL	8(4)	Sprung wenn kleiner	log_not
MVI	0(3),X'00'	FALSE in den stack	
L	3,0(12)	BOOL-stacktop-Adr --> R3	branch_on_true (IF_ELSE,5)
XI	0(3),X'FF'	logische Negation	
L	3,0(12)	BOOL-stacktop-Adr. -->R3	push(-->I);
BCTR	3,0	vermindere stacktop-Adr.	
ST	3,0(12)	speichere neue stacktop-Adr	push_int(5);
CLI	1(3),X'FF'	Test auf TRUE	
BE	<IF_ELSE5>	Sprung	mult
....		(* wie oben *)	
....		(* wie oben *)	assign(-->I)
L	4,4(12)	INT-stacktop-Adr. -->R4	
LA	3,4	Konstante 4 nach R3	
SR	4,3	vermindere stacktop-Adr	
ST	4,4(12)	speichere neue stacktopadr	
L	3,0(4)	lade ersten Operanden	
M	2,4(4)	mult. mit altem stacktop	branch(IF_SKIP,5)
ST	3,0(4)	Ergebnis in den stack	
L	3,4(12)	INT-stacktop-Adr. -->R3	put_label(IF_ELSE,5)
LA	4,4	Konstante 4 nach R4	
SR	3,4	vermindere stacktop-Adr.	put_label(IF_SKIP,5)
ST	3,4(12)	speichere neue stacktopadr	
L	4,28(12)	Adr. akt. AS nach R4	
MVC	<offset>(4,4),4(3)	aus stack in AS	
B	<IF_SKIP5>	Sprung	
<IF_ELSE5>	NOPR	Sprungmarke	
<IF_SKIP5>	NOPR	Sprungmarke	

Es wurden die üblichen IBM-Assembler **Konventionen** verwendet: Adressen haben 4 Byte, für integer-Zahlen werden ebenfalls 4 Byte benötigt und für Boolesche Werte ein Byte. Es wird hier keine symbolische Adressierung verwendet, lediglich die als Parameter übergebenen Sprungmarken werden hier noch verwendet, damit der code anschaulich bleibt. Es wird angenommen, daß Register 12 auf eine Laufzeit-Adresstabelle zeigt, in der Verweise auf die verschiedenen Datentyp-stacks stehen, sowie ein Verweis auf die Spitze des Laufzeitkellers:



Die verschiedenen möglichen Überläufe sind hier noch nicht berücksichtigt: Arithmetischer Überlauf und Kellerüberlauf (Datentyp-stacks sind als sequentielle Listen implementiert) müssen noch abgefangen werden.

Die Stellen, wo in den generierten code die Parameter der Schicht 1-Routinen eingehen, sind durch spitze Klammern markiert. Wie man sieht, bietet sich die Verwendung von **Assembler-Makros** an.

Die Codegenerierungsroutinen der Schicht 1 lassen sich in folgende **Gruppen** einteilen:

- **Konstanten in stacks schreiben**

Konstanten der verschiedenen Datentypen wie Realzahlen, Integerzahlen, strings oder auch NOAGENT, SELF oder INTERFACE werden in den entsprechenden stack geschrieben.

- **ASSIGN-Routine**

Einer Variablen wird der das oberste Element des entsprechenden stacks zugewiesen und das Stackelement wird entfernt.

- **PUSH-Routine**

Der Wert einer Variablen wird in den entsprechenden stack gebracht.

- **Stackoperationen**

Die Zwischensprache umfaßt arithmetische und logische Operationen, die ihre Operanden aus den stacks entnehmen und das Ergebnis ebenfalls in einen stack schreiben. Beispielsweise ersetzt ADD die beiden obersten Elemente des INT-stacks durch ihre Summe.

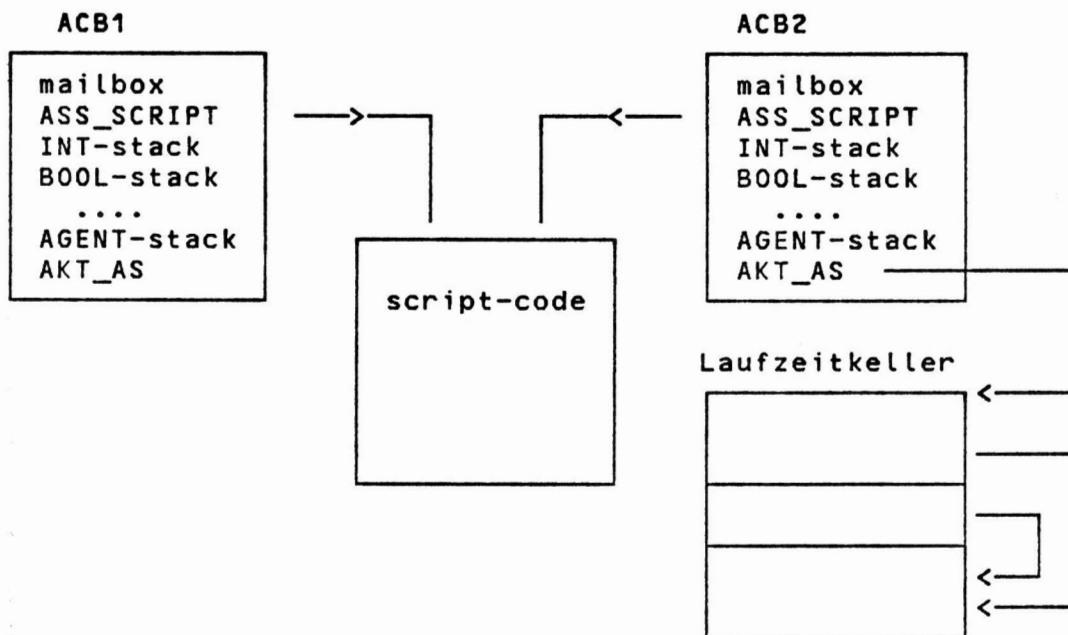
- **Sprünge**
Neben bedingten und unbedingten Sprüngen und Sprungmarken, die aus einem Namen und einer eindeutigen Nummer zusammengesetzt werden, umfaßt die Zwischensprache auch einen Unterprogrammsprung, bei dem die Rücksprungadresse in den aktuellen Aktivierungssatz eingetragen wird, und einen Rücksprung entsprechend der vermerkten Adresse.
- **Laufzeitkellerverwaltung**
Es gibt Routinen für das Anlegen und Entfernen von Aktivierungssätzen im Laufzeitkeller. Die Laufzeitkellerverwaltung wird im nächsten Abschnitt detailliert besprochen.
- **Nachrichten analysieren**
Für die Entnahme einer Nachricht aus der mailbox und die Übernahme der enthaltenen Werte und Informationen in die stacks werden spezielle Befehle in der Zwischensprache benötigt.
- **Nachrichten zusammenstellen**
Umgekehrt müssen beim Versenden einer Nachricht die Parameter und die speziellen Felder wie Zielagent oder Antwortverpflichtung in die Nachricht eingetragen werden.
- **SVC-Aufruf**
Die Dienste des Betriebssystems wie Gründen eines Agenten oder Versenden einer Nachricht werden über SVC-Aufrufe angefordert.

In den Routinen der **Schicht 2** sind jeweils mehrere Aufrufe von Routinen der Schicht 1 zusammengefaßt. Schicht 2 Routinen gibt es vor allem für das Betreten, Verlassen und den Aufruf von Prozeduren, Funktionen, Facetten, Operationen, Blöcken und Kontrollstrukturen. Dort sind die entsprechenden Sprünge und die Organisation des Laufzeitkellers zu sinnvollen Einheiten zusammengefaßt.

6.6 Laufzeitorganisation

In diesem Abschnitt sollen die wesentlichen Datenstrukturen der **virtuellen Stackmaschine** besprochen werden und auch ihre Implementierung in SIMULA angedeutet werden. Beim Entwurf der Zwischensprache wurde auf die strikte Trennung von code und Daten geachtet: Der gesamte lokale Speicher eines Agenten ist im sogenannten **ACB** (Agent Control Block) und dem jedem Agenten zugeordneten **Laufzeitkeller** untergebracht, so daß mehrere Agenten gleichen Typs ihren script-code gemeinsam benutzen können (code sharing, reentrant code).

Ein ACB enthält u.a. die mailbox, pro CSSA-Datentyp einen stack und einen Verweis auf die Spitze des Laufzeitkellers (AKT_AS):



Da die CSSA-Sprache sich am bewährten ALGOL 60-Blockkonzept orientiert, ist die **Stackimplementierbarkeit** der Speicherverwaltung für lokale Variablen gegeben: Für jede Blockinstanz gibt es im Laufzeitkeller einen **Aktivierungssatz**, der alle lokalen Variablen des Blocks (außer komplexen Datenstrukturen), sowie Verweise auf den dynamischen und statischen Vorgänger der Instanz enthält. Schließlich enthält jeder Aktivierungssatz auch Platz für eine Rückkehradresse, bei der bei Rückkehr in den Block die Programmausführung fortgesetzt wird.

Der Laufzeitkeller ist in SIMULA als eine einfach verkettete Liste realisiert. Mit Hilfe des CLASS-Konzeptes ist eine elegante Implementierung möglich: Die Oberklasse AS enthält alle Attribute, die **jeder** Aktivierungssatz - unabhängig von der Art des zugeordneten Blocks - beinhaltet.

```
class AS (LAST_AS, ENV);
  ref (AS) ENV, LAST_AS; (* statischer+dynamischer Vorgänger *)
begin
  integer RAS; (* Rücksprungsadresse *)
end AS;
```

Der Compiler generiert zu jedem Block einen Aktivierungssatz als Unterklasse von AS mit den lokalen Variablen des Blocks:

```
AS class AS3;
begin integer VAR118; (* var int: I *)
      real VAR119; (* var real: R *)
end;
```

Zur Laufzeit können die Variablen über den Verweis auf die Kellerspitze und die environment-pointer zugegriffen werden:

```
AKT_AS.ENV.ENV. ... .ENV qua AS<Blocknr>.VAR<varnr> := ...
```


Anhand des folgenden kleinen scripts soll die Verwaltung des Laufzeitkellers bei einem Prozeduraufruf demonstriert werden:

```

type S is
script ( int: I)
  var real: R :=3.75;

  procedure P is
  | I:=5;
  endprocedure;

  facet F is
  | public: OP;
  | var string: S:="CSSA";

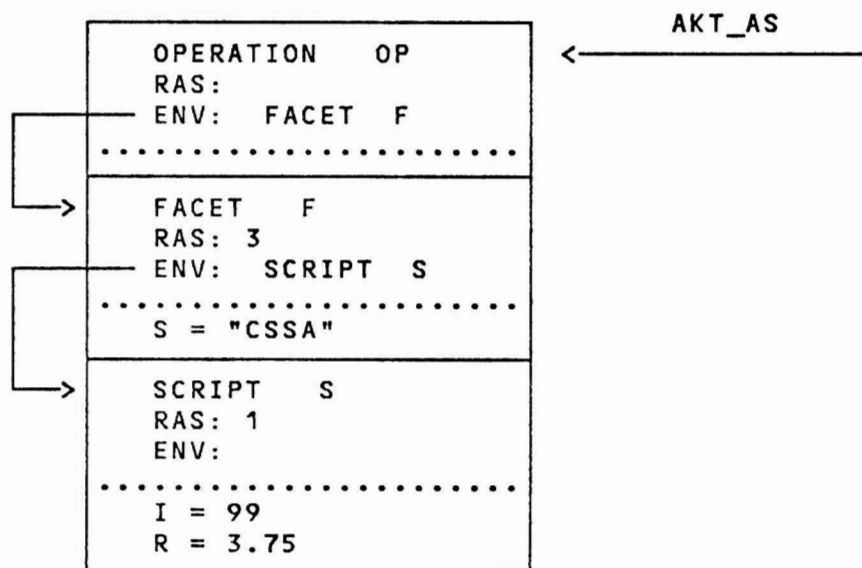
  | operation OP is
  | | call P;
  | endoperation;

  endfacet

  initial F
endscript

```

Die folgende Abbildung zeigt den Laufzeitkeller unmittelbar vor dem Aufruf der Prozedur P (wie er vom interaktiven debugger jederzeit ausgegeben werden kann):



Die gepunktete Linie zeigt die Grenze zwischen Ober- und Unterklasse an: Sie trennt den festen vom variablen Teil eines Aktivierungssatzes.

Der code für den Aufruf der Prozedur wird durch die Schicht 2-Routine PROC-CALL generiert:

```
proc-call(-->P)
  set-environment(2) ==> ENVIRONMENT: AKT_AS.ENV.ENV;
```

```
branch_to_subr("PROC",1) ==> AKT_AS.RAS:=5;
                              GOTO PROC1;
                              RAS5:
```

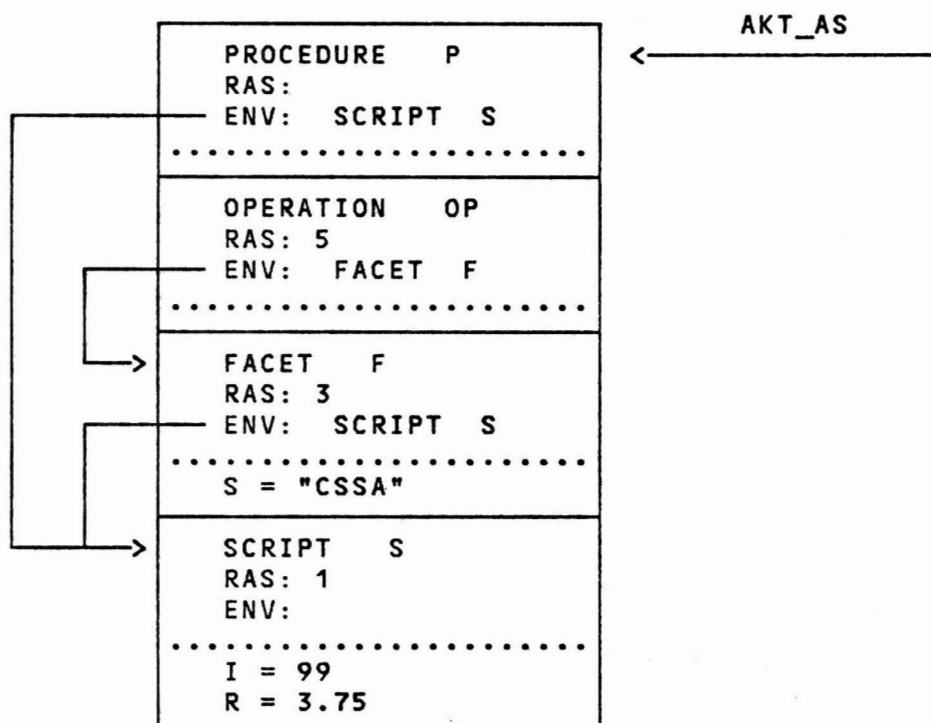
Vor dem Aufruf wird der globale Zeiger ENVIRONMENT auf den statischen Vorgänger der Prozedur P gesetzt. Dabei benutzt der Compiler die Information, daß P um 2 Blocklevel höher liegt als OP. Beim Aufruf wird die Rücksprungadresse vermerkt und verzweigt.

Der Prozedurkopf wird folgendermaßen generiert:

```
proc_head(-->P)
  put_label("PROC",1) ==> PROC1:
```

```
push_as ==> AKT_AS:-NEW AS2(AKT_AS,ENVIRONMENT);
```

Am Anfang der Prozedur P wird ein neuer AS angelegt, und der zuvor gesetzte Verweis auf den statischen Vorgänger wird eingetragen. (Der Compiler weiß, daß 2 die aktuelle Blocknummer ist, und legt dementsprechend ein Objekt vom Typ AS2 an. Bei einer Übersetzung nach Maschinensprache würde stattdessen die Länge des AS in den code eingehen.) Es ergibt sich der folgende Laufzeitkeller:



Innerhalb der Prozedur kann nun die Variable I durch 'AKT_AS.ENV qua AS1.VARnnn' adressiert werden. Bei einer Implementierung in Maschinensprache wäre entsprechend eine indirekte Adressierung über die environment-Zeiger und den offset von I (die relative Position im AS) möglich.

Beim Verlassen der Prozedur P wird der AS wieder entfernt und zur Aufrufstelle zurückgekehrt:

```
proc_tail
  pop_as ==> AKT_AS: -AKT_AS.LAST_AS;

  branch_to_ras ==> GOTO RAS(AKT_AS.RAS);
```

Da SIMULA keine direkten Sprünge zu einer absoluten Adresse erlaubt, wird mit Hilfe eines switches ein berechneter Sprung zu 'RAS5:' durchgeführt.

Das hier besprochene Beispiel zeigt stellvertretend für die gesamte CSSA-Laufzeitorganisation, daß bei der Implementierung von CSSA eine spätere Realisierung auf einem Mikroprozessorsystem bereits ins Auge gefaßt wurde und dementsprechend ein maschinennaher Zwischencode für die Speicherorganisation gewählt wurde. Eine ausführliche Beschreibung des Laufzeitsystems und der Struktur des generierten codes findet sich in Teil E dieser Arbeit.

7. Benutzerschnittstelle

Ein sehr wichtiger Aspekt beim Entwurf und der Realisierung eines Programmentwicklungssystems ist die Schnittstelle zum menschlichen Benutzer. Verständliche Meldungen und eine einfache Bedienung des Systems können den Programmentwicklungsprozeß wesentlich beschleunigen.

Deshalb wurde bei dem vorliegenden CSSA-System auf vielen Ebenen großer Wert auf eine hohe Benutzerfreundlichkeit gelegt:

- ◊ Beim Entwurf der Sprache CSSA wurden Aspekte wie **Fehlerunanfälligkeit und -vermeidung** von vornherein berücksichtigt.
- ◊ Alle Komponenten des Systems wurden in das Entwicklungssystem integriert, das eine **Benutzerführung** gestattet.
- ◊ Die Bedienung und die Ausgaben der einzelnen Komponenten wurden **benutzerorientiert** gestaltet.
- ◊ Der Compiler enthält eine **Fehlerbehandlung**, die weitgehend alle Fehler eines Programms in einem einzigen Übersetzungslauf findet und präzise diagnostiziert.
- ◊ Im interface-Agenten wurde ein **interaktiver debugger** integriert, der Testausgaben in CSSA-Programmen nahezu überflüssig macht.
- ◊ Das **Simulationssystem** wurde als komfortables Experimentiersystem konzipiert.

In diesem Kapitel werden einige der angesprochenen Punkte näher erläutert, und zwar die Vorteile des Entwicklungssystems in Abschnitt 7.1, die Schnittstelle des Compilers zum Anwender (7.2) und das Konzept des interface-Agenten mit dem integrierten interaktiven debugger (7.3). Andere Punkte werden an anderen Stellen besprochen, wie beispielsweise die Simulation (Kapitel 8.), der Sprachentwurf (5.) und die Fehlerbehandlung (6.3).

7.1 Das Entwicklungssystem

Für Entwicklung und Test von CSSA-Anwendungen steht auf dem Siemens 7760-System der Gesellschaft für Mathematik und Datenverarbeitung (GMD) ein **komfortables Entwicklungssystem** zur Verfügung, das alle Komponenten des CSSA-Systems integriert. Das Entwicklungssystem erlaubt eine weitgehende Abstraktion von der realen Rechanlage, indem es den Benutzer vom Betriebssystem abkapselt und die Betriebssystemmeldungen fast vollständig unterdrückt und statt dessen anwendungsorientierte Meldungen ausgibt. Die Steuerung des Entwicklungssystems erfolgt über Menüs, so daß dem Anwender der Eindruck vermittelt wird, als benutze er ein dediziertes CSSA-System. Er kann so ohne Kenntnisse der BS2000-Kommandosprache oder der Dateiorganisation alle Arbeiten durchführen, die zur Entwicklung von CSSA-Anwendungen gehören, wie z.B.:

- ◊ Erstellen eines CSSA-Programms mit dem Editor
- ◊ Ändern oder Erweitern eines CSSA-Programms
- ◊ Übersetzen mit dem CSSA-Compiler
- ◊ Fehlermeldungen in einer Datei ansehen
- ◊ Programm-listing ausdrucken
- ◊ Weiterübersetzen des generierten SIMULA-codes
- ◊ Ausführen eines übersetzten Programms
- ◊ Ausführungsprotokoll ausdrucken

Die Steuerung des Entwicklungssystems erfolgt ausschließlich mit Hilfe von übersichtlichen **Menüs**. Für die Auswahl der verschiedenen Dienste gibt es beispielsweise das folgende Menü:

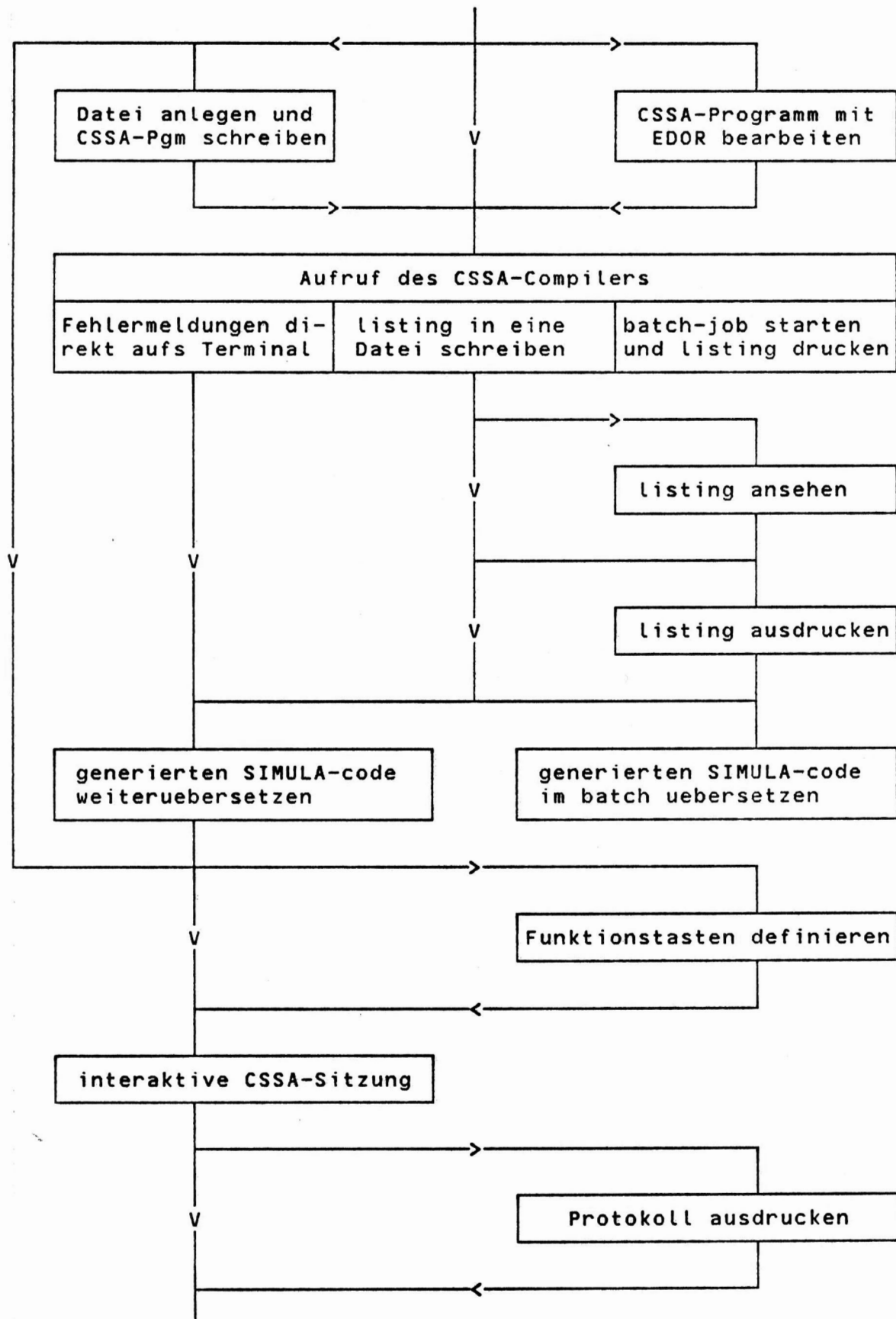
```

*****
*
*      B M S - C S S A - S Y S T E M
*      =====
*
* WAEHLEN SIE ZWISCHEN:
* - NEUES CSSA-PROGRAMM MIT EDOR SCHREIBEN      ( N )
* - ALTES CSSA-PROGRAMM MIT EDOR EDITIEREN      ( E )
* - CSSA-PROGRAMM UEBERSETZEN                   ( U )
* - EIN UEBERSETZTES CSSA-PROGRAMM AUSFUEHREN  ( A )
* - SITZUNG BEENDEN ( IMMER MOEGLICH )          ( X )
*
*****
OPTION=_

```

Der Anwender kann nun durch Eintippen eines einzelnen Buchstaben den gewünschten Dienst auswählen. Die bei der isolierten Benutzung von Compilern sonst üblichen Fehlerquellen – wie Auswahl geeigneter Compiler-Parameter, verschiedene Druckformate, Dateiformate für source-Programm, object-code, listing und Hilfsdateien usw. – entfallen hier.

Das folgende Diagramm gibt einen Überblick über die grobe Struktur des Entwicklungssystems. (Die Verbindungslinien können abwärts und seitwärts durchlaufen werden. Verzweigungen entsprechen in etwa den Menüs.)



Der **Editor** ist in das Entwicklungssystem fest integriert, er wird bei Auswahl der Option 'NEUES CSSA-PROGRAMM MIT EDOR SCHREIBEN' oder 'CSSA-PROGRAMM MIT EDOR EDITIEREN' aufgerufen. Nach Verlassen des Editors kann mit Hilfe des Entwicklungssystems die Weiterverarbeitung des editierten Programms gesteuert werden. Wenn ein neues CSSA-Programm entwickelt werden soll, wird vom Entwicklungssystem automatisch eine neue Datei angelegt, in die ein CSSA-Programmrahmen einkopiert wird. Der Benutzer kann diesen Programmrahmen wie ein Formular zum Erstellen seines CSSA-Programms nutzen. Dabei sind auszufüllende Teile in spitze Klammern gesetzt, die den Nichtterminalen der erweiterten BNF-Grammatik von CSSA entsprechen. Dies ist wohl für Anfänger der einfachste Zugang zum ersten eigenen CSSA-Programm und hilft auch erfahrenen Anwendern gewisse syntaktische Fehler von vornherein zu vermeiden.

Der Programmrahmen, der vom Entwicklungssystem zur Verfügung gestellt wird, hat folgende Gestalt:

```
type <IDENTIFIER> is
script <IDENTIFIER>
  facet <IDENTIFIER> is
    public: <IDENTIFIER> ;
          <LOCAL-DECL-LIST>

    operation <IDENTIFIER> is
      <LOCAL-DECL-LIST>
      _____
      <STATEMENT-LIST>
    endoperation

  endfacet
  initial <IDENTIFIER>
endscript
```

Das **Übersetzen** eines CSSA-Programms wird ebenfalls vom Entwicklungssystem unterstützt. Dabei können (auch über Menüs) verschiedene Modi ausgewählt werden. Die Übersetzung kann ganz oder teilweise im batch ausgeführt werden (dann generiert das Entwicklungssystem automatische geeignete batch-jobs). Bei interaktiver Übersetzung kann die Protokollierung des Compilers gesteuert werden: So ist es beispielsweise möglich, Fehlermeldungen direkt auf das Terminal auszugeben, oder das Übersetzungsprotokoll in eine Datei zum späteren Ansehen oder mit Druckaufbereitung zum Drucken in verschiedenen Ausgabeformaten zu schreiben. Dabei werden automatisch die richtigen options an den CSSA-Compiler übergeben, so daß sich der Benutzer des Entwicklungssystem hierum nicht kümmern muß.

Vor **Ausführung** eines CSSA-Programms "fragt" das Entwicklungssystem, ob die Funktionstasten des Terminals mit häufig benötigten CSSA-Kommandos belegt werden sollen. Falls dies gewünscht wird, wird die Funktionstastenbelegung am Bildschirm angezeigt. Danach wird die interaktive Sitzung gestartet.

Dem Entwicklungssystem kann auch eine Datei mit der zu simulierenden **hardware-Konfiguration** angegeben werden; andernfalls nimmt es eine Standardkonfiguration an. Außerdem kann ihm eine **Datei mit CSSA-Kommandos** übergeben werden, die bei der interaktiven Sitzung automatisch am Anfang oder auf Anfrage ausgeführt werden.

Das Compiler-listing und das Protokoll der interaktiven CSSA-Sitzung können auch nach Wahl **gedruckt** werden. Der Benutzer des Entwicklungssystems kann entscheiden, wo (remote-Station, Laserdrucker etc.) und in welchem Format (durch Wahl von Zeichensatz, Zeichengröße und Seitengröße) ausgegeben werden soll, und sogar, ob die Ausgabe auf DIN A 4-Format zugeschnitten werden soll. Das Entwicklungssystem generiert automatisch Print-Kommandos mit geeigneten Parametern (sonst eine häufige Fehlerquelle!).

7.2 Der Compiler

Zu den wichtigsten Aufgaben eines Compilers gehört neben der Übersetzung die **Kommunikation** mit dem menschlichen Benutzer, denn die meisten Programme, die einem Compiler vorgelegt werden, sind falsch. (Richtige Programme werden in der Regel nur einmal übersetzt.) Deshalb wurde auch beim konzipierten CSSA-Compiler Wert auf eine wenig fehleranfällige Eingabesprache (Sprachdesign siehe Kapitel 5), eine gute Fehlerbehandlung (siehe 6.3 und Anhang) und eine übersichtliche Ausgabe gelegt.

Die **Ausgabe** des CSSA-Compilers besteht aus der aufbereiteten Programmliste mit attributierter cross-reference Liste und ausführlichen Fehlermeldungen mit Korrekturannahmen. Auf Wunsch können auch Teile des Ableitungsbaums ausgegeben werden.

Es folgt ein Beispiel für ein **listing** mit cross-reference Liste, wie es der Compiler erzeugt. Aus drucktechnischen Gründen mußte die Ausgabe in gestauchter Form dargestellt werden.

BMS-CSSA-COMPILER

1982/04/30 14:20:10.00 PAGE 1

DEFAULT-OPTIONS: NOTERM,...,SOURCE,NOSTRUCT,XREF,RESWD=A,...

OPTIONS IN USE: NOTERM,...,SOURCE, STRUCT,XREF,RESWD=A,...

	1	2	3	...	BLOCKNESTING	8
1	\$	*** PHILOSOPHEN-PROBLEM ***	...	\$		00000010
2		type PHILOSOPH is	...			00000020
3		script (agent: LFORK,RFORK)	...		+1	00000030
4		facet ALIVE is	...		+2	00000040
5		public: PHILOSOPH;	...			00000050
6		operation PHILOSOPH is	...		+3	00000060
7		oper: PICKUP,PUTDOWN;	...			00000070
8		port: FIRST,SECOND;	...			00000080
9			...			00000090
10		send PICKUP to RFORK	...			00000100
11		reply to FIRST;	...			00000110
12		wait FIRST;	...			00000120
==>13		SAND PICKUP to LFORK	...			00000130
14		reply to SECOND;	...			00000140
15		wait SECOND;	...			00000150
16		(* EATING *)	...			00000160
17		send PUTDOWN to LFORK;	...			00000170
18		send PUTDOWN to RFORK;	...			00000180
19		(* THINKING *)	...			00000190
20		endoperation (*PHILOSOPH*)	...		-3	00000200
21		endfacet (* ALIVE *)	...		-2	00000210
22		initial ALIVE	...		*1	00000220
23		endscript (* PHILOSOPH *)	...		-1	00000230

BMS-CSSA-COMPILER - DATE OF RELEASE: 30 SEP 81 1 ERROR DETECTED
 END OF COMPILING ON 1982/04/30 AT 14:20:16.00 RETURNCODE = 7
 COMPILE-TIME (CPU) = 1.28 SEC. EXECUTION-TIME = 7.00 SEC.
 NUMBER OF SOURCE-LINES READ = 23 NUMBER OF TOKENS = 72
 NUMBER OF OBJECT-RECORDS GENERATED = 168

*** CROSS-REFERENCE-TABLE ***

ALIVE	FACET	4	22	
FIRST	PORT	8	11	12
LFORK	CONSTANT:AGENT	3	13	17
PHILOSOPH	OPERATION:OPER	5	6	
PHILOSOPH	USER_DEFINED_TYPE:SCRIPT	2		
PICKUP	LITERAL:OPER	7	10	13
PUTDOWN	LITERAL:OPER	7	17	18
RFORK	CONSTANT:AGENT	3	10	18
SECOND	PORT	8	14	15

- MULTIPLE OCCURENCES ON THE SAME LINE ARE MARKED WITH '+'.
 - TOTAL NUMBER OF IDENTIFIERS USED IN THIS PROGRAM: 9

==> SYNTAX-ERROR IN LINE 13 : ILLEGAL SYMBOL 'SAND'
IN <OPERATION_DECL> STARTING IN LINE 6
CHECK, IF 'ENDOPERATION'|';'|...|'REPLY'|'SEND'
SHOULD HAVE BEEN USED INSTEAD

- ASSUMPTION IN LINE 13 :
'SAND' ASSUMED TO BE A MISPELLED 'SEND'
- ASSUMPTION IN LINE 13 : 'SAND' ASSUMED TO BE PART OF
<STATEMENT_LIST> EXPECTED IN <OPERATION_DECL>
STARTING IN LINE 6

In jeder **Seitenüberschrift** steht das Datum und die Uhrzeit, an dem die Übersetzung begonnen wurde; außerdem eine Seitenzahl.

Auf Seite eins folgt eine Liste der **default-options**, sowie eine Liste der für diesen Übersetzungslauf **aktiven options**.

In der Kopfzeile des source-listing erscheint eine Zeile, in der **Spaltennummern** angezeigt werden (manche Fehlermeldungen beziehen sich auf bestimmte Spalten).

Das **Quellprogramm** wird im wesentlichen unverändert zwischen zwei senkrechten Begrenzungslinien ausgegeben. Insbesondere wird auf keinen Fall die Anordnung der Zeichen innerhalb der Zeilen verändert. Die Fehlermeldungen erscheinen nicht zwischen den Zeilen, sondern im Anschluß an die Programmliste, um eine gute Übersichtlichkeit und ein schnelles Auffinden der Fehlermeldungen auch bei längeren Programmen zu gewährleisten. Fehlerhafte Zeilen werden jedoch durch einen auffälligen Pfeil am linken Rand gekennzeichnet; außerdem wird in der Regel das fehlerhafte Symbol unterstrichen. So reicht meistens ein Blick in die Programmliste aus, um einfache Fehler zu erkennen. Nur bei schwierigeren Fehlern müssen die Meldungen nachgelesen werden. Die Ausgabe des Quellprogramms kann durch Steueranweisungen beeinflusst werden (z.B. Seitenvorschub mit einer neuen Kopfzeile).

Schlüsselwörter werden je nach verwendeter option hervorgehoben - hier durch Kleinschreibung und Fettdruck. Der Inhalt der **Spalten 73-80** wird - getrennt vom Text der Spalten 1-72 - ganz rechts außen ausgegeben, da dieser nicht analysiert wird. Links neben den Zeilen erscheint eine **Zeilennummerierung**, die mit der Satzzahl innerhalb der Datei übereinstimmt. Zeilen, zu denen Fehler gemeldet werden und Zeilen, in denen sich aufgrund von Fehlerkorrekturmaßnahmen des Compilers überlesene Symbole (gestrichelt unterstrichen) befinden, werden mit einem **Pfeil** am linken Rand gekennzeichnet.

Unter der Überschrift **BLOCKNESTING** wird rechts neben dem Quellprogramm noch eine Information über die **Blockstruktur** des Programms ausgegeben. Diese Information kann insbesondere im Fehlerfall darüber Auskunft geben, wie der Compiler die Blockstruktur "verstanden" hat. Die versetzte senkrechte Linie vermittelt einen anschaulichen Eindruck von der syntaktischen Struktur des Programms: Sie ist entsprechend der Schachtelungstiefe nach rechts eingerückt. Der Anfang und das Ende eines Blockes sind gekennzeichnet, so daß sich leicht zu jeder syntaktischen Klammer die korrespondierende Klammer finden

läßt. Die Rückkehr auf ein Blockniveau wird ebenfalls markiert. Es läßt sich daher auf einen Blick zu jeder Zeile des Quellcodes feststellen, welches die direkt umgebende syntaktische Klammer ist.

Auf Wunsch (durch eine Compiler-option) wird die Blockstruktur auch durch **senkrechte Linien im source-listing** hervorgehoben. Dadurch wird erzwungen, daß der Programmierer den Quelltext geeignet eingerückt aufschreibt (was gleichzeitig Fehler in der Blockstruktur vermindert, bzw. offensichtlich macht). Aus dem gleichen Grund wurde auf ein automatisches Einrücken bewußt verzichtet. (Automatisches Einrücken hatte einmal bei der Eingabe mittels Lochkarten seine Berechtigung; ist aber bei der interaktiven Sprache CSSA nicht notwendig)

Im Anschluß an die Programmliste werden noch einige **Statistiken** über den Übersetzungslauf angegeben. Es folgt eine **attributierte cross-reference Liste**, die zwischen verschiedenen aber gleichnamigen Bezeichnern unterscheidet (siehe Bezeichner PHILOSOPH im Beispiel).

Der Compiler erzeugt ggf. eine dem **Terminal angepaßte Ausgabe**: Das listing wird ohne Spalten 73-80 und ohne Pfeile ausgegeben, damit das Layout nicht durch Zeilenumbrüche unübersichtlich wird; es wird so der Tatsache Rechnung getragen, daß ein Bildschirm weniger Platz bietet als output-Papier. Fehlermeldungen werden bereits ab Spalte 70 umgebrochen und direkt vor den fehlerhaften Zeilen ausgegeben.

Auf Wunsch (durch eine Compiler-Steueranweisung) kann zu ausgewählten Zeilen eines CSSA-Programms ein **Ableitungsbaum** in preorder bezüglich der in [BMS-B] angegebenen erweiterten BNF-Grammatik ausgedruckt werden. Man erhält so detaillierte Information, wie der Compiler die syntaktische Struktur des Programms aufgefaßt hat.

Hier als Beispiel der erste Teil des Ableitungsbaumes zu dem oben abgedruckten Programm (Terminalsymbole sind fett gedruckt; ein längeres Beispiel findet sich in Teil C Kap. 4.6):

```
LINE  2
      <PROGRAM>
        TYPE
        PHILOSOPH
        IS

LINE  3
      <SCRIPT_DEFINITION>
        SCRIPT
        <PATTERN>
          (
            <DECL_FIELD>
              <TYPE>
                AGENT
              :
            <IDENTIFIER_LIST>
              LFORK
            ,
```

RFORK

,

LINE 4

```
<LOCAL_DECL_LIST>
<FACET_SPECIFICATION>
  <FACET_DECL>
    FACET
    ALIVE
    IS
```

7.3 Die interaktive CSSA-Sitzung

CSSA stellt bereits vom Konzept her eine integrierte, interaktive Benutzerschnittstelle zur Verfügung, den **interface-Agenten**. Dieser ist dynamisch programmierbar, die interaktiv eingegebenen Instruktionen werden interpretativ ausgeführt. Dadurch kann der Benutzer Agenten erzeugen, lokale Größen deklarieren und Zuweisungen an lokale Größen vornehmen; vor allem aber kann er mit den anderen Agenten des Netzes mit den üblichen CSSA-Sprachmechanismen (send/receive) kommunizieren.

Darüberhinaus kann der Benutzer - dank der Simulation - die gesamte Berechnung steuern und überwachen. Das interface bearbeitet neben der CSSA-Teilsprache noch einige **privilegierte Anweisungen**, die in einem realen System nicht (bzw. nicht so einfach) implementiert werden können, da dabei ausgenutzt wurde, daß in der Simulation alle Agenten eine gemeinsame Umgebung haben. So kann der Benutzer das Fortschreiten der Simulation steuern und Agenten stoppen oder starten. (Die Steuerung der Simulation ist ausführlich in Kapitel 8.3 beschrieben.) Außerdem kann über den interface-Agenten der **interaktive debugger** bedient werden, der eine Reihe von Zustandsinformationen über die Agenten und das Simulationssystem liefert, sowie unterschiedliche Grade der Systemprotokollierung - bis hin zum statementweisen tracing - erlaubt.

Law of Debugging:

A DIFFICULT ERROR IS NEVER WHERE YOU THINK IT IS.

P. Brown

Es folgt eine Übersicht über alle debugging-Möglichkeiten. Dabei sind Befehle nach ihrem Detaillierungsgrad gestaffelt aufgeführt:

Befehle zur
Zustandsinformation

Befehle zur
Protokollierung

Sichtweise

V	V	V	Meldung (Kennung im Protokoll)
	OBSERVE	Agentennetz	Versenden/Empfangen von Nachrichten, Facettieren, Gründen von Agenten etc. "+++"
STATUS	EVENT	Agentennetz	Alle Agenten mit akt. Facette, Operation und mailbox "+++ ALL EXISTING AGENTS:"
MAILBOX		Mailbox e. Agenten	Operationsbezeichner Typ und Wert der Parameter "MAILBOX OF ...:"
DUMP	SNAPSHOT	lokaler Speicher e. Agenten	Laufzeitkeller akt. Zeilennummer lokale Variablen "+++ RUNTIME STACK OF ..."
	TRACE	Code e. Agenten	Ausführung eines Befehls "—— TRACING ... ——"
	SYSTEM	Betriebs- system	(De-)Aktivierung v. Agenten Übermittlung v. Nachrichten "///"
SYSSTATUS		Prozessoren und Busse	Auslastung und Belegung "+++ SYSTEM STATUS:"

Am Beispiel des ggT-Algorithmus (siehe Kapitel 4.2) soll hier demonstriert werden, in welchen verschiedenen Sichtweisen und mit welchem Detaillierungsgrad ein Abschnitt einer interaktiven CSSA-Sitzung betrachtet werden kann. Zur Erinnerung sei hier nochmal die reine Ausgabe des ersten ggT-Programms aus Kapitel 4.2 abgedruckt, die durch die Print-Statements der Agenten erzeugt wird. Hier werden also keinerlei Ausgabe-Anweisung des interaktiven debuggers verwendet. (Die Anweisungen zur Erzeugung und Bekanntmachung der Agenten wurden bei allen Beispielen weggelassen - sie können in 4.2 nachgelesen werden.)

=> RUN

```
***      0.300 PROCESS(1) : *** MY CURRENT NUMBER : 108 ***
***      0.300 PROCESS(2) : *** MY CURRENT NUMBER : 76 ***
***      0.700 PROCESS(4) : *** MY CURRENT NUMBER : 60 ***
***      0.700 PROCESS(3) : *** MY CURRENT NUMBER : 12 ***
***      0.700 PROCESS(5) : *** MY CURRENT NUMBER : 36 ***
***      1.500 PROCESS(4) : *** MY CURRENT NUMBER : 12 ***
***      1.500 PROCESS(2) : *** MY CURRENT NUMBER : 4 ***
***      1.600 PROCESS(1) : *** MY CURRENT NUMBER : 32 ***
***      2.500 PROCESS(1) : *** MY CURRENT NUMBER : 4 ***
***      2.500 PROCESS(3) : *** MY CURRENT NUMBER : 4 ***
***      2.600 PROCESS(5) : *** MY CURRENT NUMBER : 12 ***
***      3.200 PROCESS(4) : *** MY CURRENT NUMBER : 4 ***
***      3.500 PROCESS(5) : *** MY CURRENT NUMBER : 4 ***
>>>      4.400 SYSTEM TERMINATED
```

Einen sehr guten Eindruck von einer ablaufenden parallelen Berechnung in CSSA erhält man durch das **OBSERVE-Kommando**. Es meldet die wichtigsten Aktionen aller Agenten, wie Senden und Empfangen von Nachrichten, Facettieren und Starten von Operationen und Erzeugung neuer Agenten. Als Beispiel folgt ein Ausschnitt der Ausgabe des OBSERVE-Kommandos für den o.g. ggT-Algorithmus:

==> OBSERVE; RUN

```

+++ 0.100 P2: PROCESS(1) RECEIVES
      KNOW(AGENT:PROCESS(5),AGENT:PROCESS(2))
      FROM INTERFACE(1)
+++ 0.100 P3: PROCESS(2) RECEIVES
      KNOW(AGENT:PROCESS(1),AGENT:PROCESS(3))
      FROM INTERFACE(1)
+++ 0.100 P2: PROCESS(1) STARTING OPERATION
      KNOW(AGENT:PROCESS(5),AGENT:PROCESS(2))
+++ 0.100 P3: PROCESS(2) STARTING OPERATION
      KNOW(AGENT:PROCESS(1),AGENT:PROCESS(3))
+++ 0.200 P2: PROCESS(1) PERFORMS FACETTING: INIT --> DISPLAY
+++ 0.200 P2: PROCESS(1) STARTING OPERATION IDLE
+++ 0.200 P3: PROCESS(2) PERFORMS FACETTING: INIT --> DISPLAY
+++ 0.200 P3: PROCESS(2) STARTING OPERATION IDLE
*** 0.300 PROCESS(1) : ***** MY CURRENT NUMBER : 108 *****
*** 0.300 PROCESS(2) : ***** MY CURRENT NUMBER : 76 *****
+++ 0.400 P2: PROCESS(1) SENDS EUKLID(INT:108) TO PROCESS(5)
+++ 0.400 P3: PROCESS(2) SENDS EUKLID(INT:76) TO PROCESS(1)
+++ 0.500 P5: PROCESS(4) RECEIVES
      KNOW(AGENT:PROCESS(3),AGENT:PROCESS(5))
      FROM INTERFACE(1)
+++ 0.500 P4: PROCESS(3) RECEIVES
      KNOW(AGENT:PROCESS(2),AGENT:PROCESS(4))
      FROM INTERFACE(1)
+++ 0.500 P6: PROCESS(5) RECEIVES
      KNOW(AGENT:PROCESS(4),AGENT:PROCESS(1))
      FROM INTERFACE(1)
+++ 0.500 P2: PROCESS(1) SENDS EUKLID(INT:108) TO PROCESS(2)
+++ 0.500 P3: PROCESS(2) SENDS EUKLID(INT:76) TO PROCESS(3)
+++ 0.500 P5: PROCESS(4) STARTING OPERATION
      KNOW(AGENT:PROCESS(3),AGENT:PROCESS(5))
+++ 0.600 P5: PROCESS(4) PERFORMS FACETTING: INIT --> DISPLAY
+++ 0.600 P5: PROCESS(4) STARTING OPERATION IDLE
+++ 0.600 P6: PROCESS(5) PERFORMS FACETTING: INIT --> DISPLAY
+++ 0.600 P6: PROCESS(5) STARTING OPERATION IDLE
+++ 0.600 P2: PROCESS(1) PERFORMS FACETTING: DISPLAY-->ACCEPT
+++ 0.600 P3: PROCESS(2) PERFORMS FACETTING: DISPLAY-->ACCEPT
+++ 0.600 P2: PROCESS(1) IS IDLE
+++ 0.600 P3: PROCESS(2) IS IDLE
*** 0.700 PROCESS(4) : ***** MY CURRENT NUMBER : 60 *****
*** 0.700 PROCESS(3) : ***** MY CURRENT NUMBER : 12 *****
*** 0.700 PROCESS(5) : ***** MY CURRENT NUMBER : 36 *****
+++ 0.800 P5: PROCESS(4) SENDS EUKLID(INT:60) TO PROCESS(3)
+++ 0.800 P6: PROCESS(5) SENDS EUKLID(INT:36) TO PROCESS(4)
+++ 0.900 P5: PROCESS(4) SENDS EUKLID(INT:60) TO PROCESS(5)

```

Das **STATUS-Kommando** gibt eine tabellarische Übersicht über alle existierenden Agenten aus. Dabei wird für jeden Agenten gemeldet, auf welchem Prozessor er läuft, in welcher Facette und evtl. in welcher Operation er sich befindet und welche Nachrichten in seiner mailbox sind. (Wenn der Platz reicht, wird auch der Inhalt der Nachrichten gezeigt.) Das **EVENT-Kommando** bewirkt, daß bei jedem wichtigen Ereignis (Versenden und Empfangen von Nachrichten, Gründen von Agenten) die STATUS-Information angezeigt wird. Zuvor wird jeweils der Bildschirm gelöscht, so daß dem Benutzer der Eindruck vermittelt wird, die status-Tabelle würde nicht jeweils erneut ausgegeben, sondern lediglich ein Eintrag würde geändert. Da darüberhinaus auch die gerade versandte Nachricht durch einen vorangestellten Pfeil gekennzeichnet wird, erhält der Benutzer einen recht plastischen Eindruck von der Kommunikation im Agentennetz. Zur Demonstration folgt die status-Tabelle zum Zeitpunkt 1.700 (ein Stern in der Tabelle besagt, daß der Agent gegenwärtig aktiv ist).

==> STATUS

+++ 1.700 ALL EXISTING AGENTS:

AGENT	FACET	OPERATION	MAILBOX
P1: INTERFACE(1) *			
P2: PROCESS(1) *	DISPLAY	IDLE	==> EUKLID(INT:32) EUKLID(INT:108) EUKLID EUKLID
P3: PROCESS(2) *	DISPLAY	IDLE	
P4: PROCESS(3)	ACCEPT		
P5: PROCESS(4) *	DISPLAY	IDLE	
P6: PROCESS(5)	ACCEPT		

Mit Hilfe des Kommandos **MAILBOX** können alle Nachrichten in der mailbox eines Agenten inspiziert werden. Als Beispiel soll hier die mailbox des Agenten PROCESS(3) ausgedruckt werden, die in der status-Tabelle nicht vollständig abgedruckt werden konnte:

==> MAILBOX PROCESS(3)

MAILBOX OF PROCESS(3):

(1) EUKLID(INT:60)
(2) EUKLID(INT:76)

Das **DUMP-Kommando** des interaktiven debuggers liefert den gesamten Laufzeitkeller eines Agenten: Zu jeder Variablen wird der symbolische Name und der aktuelle Wert ausgegeben; Testausgaben in CSSA-Programmen sind somit in der Regel überflüssig. Die **SNAPSHOT-Anweisung** bewirkt, daß jedesmal bei Betreten eines Blockes der Laufzeitkeller des angegebenen Agenten ausgegeben wird. Es folgt der Laufzeitkeller des Agenten `PROCESS(3)` zum Zeitpunkt 0.500. Oberhalb der punktierten Linie stehen die Informationen, die bei jedem Block vorhanden sind: Art und Name des Blocks, Verweis auf den statisch umgebenen Block und Zeilennummer, bis zu der der Block abgearbeitet ist. Darunter stehen - falls vorhanden - die lokalen Variablen des Blocks:

==> DUMP PROCESS(3)

+++ 0.500 RUNTIME STACK OF PROCESS(3)

OPERATION	KNOW
ENV:	FACET INIT
LINE:	21
.....	
FACET	INIT
ENV:	SCRIPT PROCESS
LINE:	20
.....	
SCRIPT	PROCESS
ENV:	
LINE:	37
.....	
N = 12	
NUMBER = 12	
LEFT = PROCESS(2)	
RIGHT = PROCESS(4)	

Die detaillierteste Form der Protokollierung ist das **tracing**. Jeder einzelne Befehl eines Agenten wird unter der Angabe der Zeilennummer genauestens protokolliert. Das Protokoll ist in Abschnitte gegliedert (genauer siehe Teil C Kap. 5.4), die mit dem Namen der aktuellen Facette und evtl. der aktuellen Operation überschrieben sind:

==> TRACE PROCESS(3); RUN

```
0.500  TRACING PROCESS(3) / INIT
        SEARCHING NEXT_MESSAGE
        FOUND : KNOW(AGENT:PROCESS(2),AGENT:PROCESS(4))
        LINE 21: LEFT := PROCESS(2)
        LINE 21: RIGHT := PROCESS(4)
        STARTING OPERATION KNOW(AGENT:PROCESS(2),
                                AGENT:PROCESS(4))
        LINE 22: FACETTING : INIT --> DISPLAY
        SEARCHING NEXT_MESSAGE - NO MESSAGE FOUND
        STARTING OPERATION IDLE
```

```
***    0.700  PROCESS(4) : ***** MY CURRENT NUMBER : 60 *****
```

```
0.700  TRACING PROCESS(3) / DISPLAY / IDLE
***    0.700  PROCESS(3) : ***** MY CURRENT NUMBER : 12 *****
```

```
***    0.700  PROCESS(5) : ***** MY CURRENT NUMBER : 36 *****
```

```
0.800  TRACING PROCESS(3) / DISPLAY / IDLE
        LINE 13: SEND EUKLID(INT:12) TO PROCESS(2)
        LINE 14: SEND EUKLID(INT:12) TO PROCESS(4)
        LINE 15: FACETTING : DISPLAY --> ACCEPT
        SEARCHING NEXT MESSAGE - NO MESSAGE FOUND
        SEARCHING NEXT MESSAGE
        FOUND : EUKLID(INT:60)
        LINE 28: NEIGHBOUR := 60
        SEARCHING NEXT MESSAGE - NO MESSAGE FOUND
```

Mit Hilfe der Protokollierungsanweisung **SYSTEM** bekommt man einen Überblick über die Aktionen der Betriebssystemkerne auf den Prozessoren und den Aktionen der Busse. Die für das Verständnis der Meldungen notwendigen Informationen werden in Kapitel 8 geliefert.

==> SYSTEM; RUN

```
/// 0.100 B1: STARTING TO TRANSMIT
///      KNOW(AGENT:PROCESS(2),AGENT:PROCESS(4))
/// 0.100 P2: PROCESSOR IS ACTIVATED BY EXTERNAL INTERRUPT
/// 0.100 P2: PROCESS(1) ACTIVATED
/// 0.100 B2: STARTING TO TRANSMIT
///      KNOW(AGENT:PROCESS(4),AGENT:PROCESS(1))
/// 0.100 P3: PROCESSOR IS ACTIVATED BY EXTERNAL INTERRUPT
/// 0.100 P3: PROCESS(2) ACTIVATED
/// 0.200 B1: STARTING TO TRANSMIT
///      KNOW(AGENT:PROCESS(3),AGENT:PROCESS(5))
/// 0.200 B2: BUS IS IDLE
/// 0.300 B1: BUS IS IDLE
*** 0.300 PROCESS(1) : ***** MY CURRENT NUMBER : 108 *****
*** 0.300 PROCESS(2) : ***** MY CURRENT NUMBER : 76 *****
/// 0.400 P2: PASSING KNOW(AGENT:PROCESS(2),AGENT:PROCESS(4))
/// 0.400 P2: PASSING KNOW(AGENT:PROCESS(3),AGENT:PROCESS(5))
/// 0.400 P2: PROCESS(1) SUSPENDED
/// 0.400 P2: PROCESS(1) ACTIVATED
/// 0.400 B4: STARTING TO TRANSMIT
///      KNOW(AGENT:PROCESS(3),AGENT:PROCESS(5))
/// 0.400 B3: STARTING TO TRANSMIT
///      KNOW(AGENT:PROCESS(2),AGENT:PROCESS(4))
/// 0.400 B1: STARTING TO TRANSMIT EUKLID(INT:108)
/// 0.400 P3: PASSING KNOW(AGENT:PROCESS(4),AGENT:PROCESS(1))
/// 0.400 P3: PROCESS(2) SUSPENDED
/// 0.400 P3: PROCESS(2) ACTIVATED
/// 0.400 B5: STARTING TO TRANSMIT
///      KNOW(AGENT:PROCESS(4),AGENT:PROCESS(1))
/// 0.400 B2: STARTING TO TRANSMIT EUKLID(INT:76)
/// 0.500 B4: BUS IS IDLE
/// 0.500 P5: PROCESSOR IS ACTIVATED BY EXTERNAL INTERRUPT
/// 0.500 P5: PROCESS(4) ACTIVATED
/// 0.500 B3: BUS IS IDLE
/// 0.500 P4: PROCESSOR IS ACTIVATED BY EXTERNAL INTERRUPT
/// 0.500 P4: PROCESS(3) ACTIVATED
/// 0.500 B1: BUS IS IDLE
/// 0.500 B5: BUS IS IDLE
```

Einen Überblick über die Verteilung laufender Agenten und Nachrichten auf die simulierte hardware gibt die **SYSSTATUS-Anweisung**. Diese Anweisung informiert auch über die Auslastung von Prozessoren und Bussen. In Kapitel 8.4 befinden sich weitere sinnvolle Beispiele zur Verwendung der SYSSTATUS-Anweisung. Zur Demonstration sei hier die sysstatus-Tabelle zum Zeitpunkt 1.000 angegeben:

==> SYSSTATUS

+++ 1.000 SYSTEM STATUS:

PROC.	UTIL.	AGENTS		
P1	100%	INTERFACE(1)*		
P2	50%	PROCESS(1)		
P3	50%	PROCESS(2)		
P4	50%	PROCESS(3)		
P5	50%	PROCESS(4)		
P6	50%	PROCESS(5)		
P7	0%			
BUS	UTIL.	#MSGs	AVG-QLEN	MESSAGES
B1	60%	7	0.43	EUKLID
B2	50%	6	0.17	EUKLID
B3	30%	4	0.50	EUKLID
B4	0%	0	0.00	
B5	0%	0	0.00	
B6	0%	0	0.00	

8. Das Simulationssystem

Bei der Suche nach einer geeigneten **Zielmaschine** für den CSSA-Compiler stellte es sich seinerzeit heraus, daß noch keine dem CSSA-Konzept wirklich angemessene hardware zur Verfügung stand. Auch sollte zu diesem frühen Zeitpunkt vermieden werden, das CSSA-System an eine bestimmte Maschine anzupassen, die doch bald wieder von der raschen Entwicklung auf dem Gebiet der Mikroprozessoren überholt würde. Stattdessen wurde ein **Mehrprozessor-Simulationssystem** entworfen, das es erlaubt, verschiedene hardware-Konfigurationen in realistischer Weise nachzubilden und Erfahrungen mit verteilten Betriebssystemen für solche Konfigurationen zu gewinnen. Vor allem aber ermöglicht das Simulationssystem die realitätsgetreue **quasi-parallele** Ausführung der vom Compiler generierten script-Module auf einem normalen Großrechner.

Es zeigte sich, daß die Simulation hervorragend geeignet ist, um ein umfassendes **Experimentiersystem** für das CSSA-Konzept zur Verfügung zu stellen:

- Es kann eine Vielzahl verschiedener **hardware-Konfigurationen** aus beliebig vielen Prozessoren und Bussen hinsichtlich ihrer Eignung für CSSA und CSSA-Anwendungen untersucht werden.
- Es können Erfahrungen mit dem **interrupt-gesteuerten, verteilten multiprogramming-Betriebssystem** gewonnen werden und seine Eignung für CSSA untersucht werden. Es bieten sich eine Reihe von Ansatzpunkten für eine noch detailliertere Simulation: Ein besserer, speziell für CSSA geeigneter Scheduler, der den Agenten Rechenzeit zuteilt, oder auch eine neue Strategie zur Verteilung der Agenten auf die Prozessoren könnten entwickelt werden.
- Es kann getestet werden, inwieweit die **CSSA-Sprache** zur Formulierung **paralleler Algorithmen** geeignet ist, und in welchem Maße CSSA-Anwendungen die Möglichkeiten zur Parallelarbeit ausnutzen, die von der hardware-Seite angeboten werden.

Die Bedeutung der bei der Simulation gewonnenen Ergebnisse wird dadurch aufgewertet, daß einerseits die in der CSSA-Sprache formulierten Algorithmen auf einer **realistischen** Konfiguration laufen, und andererseits hardware und Betriebssystem nicht unter einer fiktiven (modellierten) Last getestet werden, sondern bei der tatsächlichen Ausführung **echter** CSSA-Programme.

Darüberhinaus bietet die Simulation eine Reihe von Vorteilen, die bei einer Implementation auf einem Mikroprozessorsystem entfallen wären:

- CSSA-Berechnungen verlaufen im allgemeinen **nicht-deterministisch**; das erschwert in hohem Maße das Austesten von Programmen, da auftretende Fehler oft nicht reproduzierbar sind. In der Simulation wird das nicht-deterministische Verhalten aber durch Zufallszahlengeneratoren hervorgerufen, die bei wiederholter Ausführung eines

Programms stets die gleiche Zahlenfolge erzeugen, so daß die **Reproduzierbarkeit** hier gewährleistet ist.

- In einer schwach gekoppelten Mehrrechnerkonfiguration ist eine Kommunikation **ausschließlich über Nachrichten** möglich. Durch die von null verschiedene Nachrichtenlaufzeit bekommt das System einen relativistischen Charakter: Die Gleichzeitigkeit zweier Ereignisse oder eine lineare Ordnung von Ereignissen ist nicht mehr ohne weiteres feststellbar oder gar nicht mehr gegeben, was das Entwickeln und Testen von Programmen für verteilte Berechnungen erschwert. In der Simulation hingegen kommunizieren zwar die Agenten ebenfalls nur durch Nachrichten, für Testhilfen, Statistiken und Protokolle jedoch wird die Tatsache ausgenutzt, daß alle Agenten und Prozessoren eine gemeinsame **globale Umgebung** haben. Durch die quasi-Parallelität ist eine lineare Ordnung der Ereignisse gegeben. Protokolle unterschiedlichsten Detaillierungsgrades können von Prozessoren, Bussen und Agenten direkt ausgegeben werden. Für Statistiken können globale Zähler verwendet werden. Der Zustand von Agenten, Bussen und Prozessoren kann jederzeit inspiziert werden; beispielsweise können die lokalen Variablen oder die mailbox eines beliebigen Agenten gezeigt werden. In beschränktem Umfang sind sogar direkte Eingriffe in das System möglich.
- Erst durch die Simulation war es möglich, das CSSA-System auf einem **Großrechner** einer Vielzahl von Benutzern zur Verfügung zu stellen. Die auf einem Mikroprozessorsystem zu erwartenden **Speicherplatzprobleme** entfielen hier. Die CSSA-Sprache konnte daher im vollen Umfang implementiert werden und eine Reihe von Testhilfen zur Verfügung gestellt werden, so daß insgesamt ein **benutzerfreundliches** und **stabiles** System geschaffen wurde.

Das Simulationssystem wurde in der Sprache **SIMULA** geschrieben. Dadurch, daß der CSSA-Compiler ebenfalls SIMULA-code generiert, ist es nicht nötig, daß die simulierten Prozessoren den generierten code **interpretieren**: Stattdessen wird der generierte code zusammen mit dem Simulationssystem (und einigen Laufzeit-routinen) vom SIMULA-Compiler in Maschinensprache weiterübersetzt. Soll bei der Ausführung eines CSSA-Programms ein Agent rechnen, so wird einfach der zugehörige script-code aktiviert und ausgeführt. Auf diese Weise ergeben sich akzeptable Ausführungszeiten, und der zusätzliche Aufwand für die Simulation bleibt gering.

Es hat sich gezeigt, daß SIMULA nicht nur als Simulationssprache, sondern auch als **vollwertige Programmiersprache** hervorragend geeignet ist. Obwohl mittlerweile 15 Jahre alt, bietet SIMULA mit dem (zugegebenermaßen etwas überladenen) CLASS-Konzept ausgezeichnete Möglichkeiten zur Datenkapselung, Modularisierung und zur Daten- und Kontrollabstraktion, wie sie heute für PASCAL-ähnliche Sprachen vielfach erst gefordert werden. Die Konstrukte für die Simulation fügen sich nahtlos in das Gesamtkonzept ein und sind auch unabhängig von einer Simulation zur Strukturierung größerer Anwendungen in Module, Prozesse und Koroutinen sehr nützlich. Bei der Programmierung des CSSA-

Compilers und des Simulationssystems wurden die SIMULA-Konzepte in extensiver Weise und mit Vorteil genutzt. Daß SIMULA seinerzeit nicht einen ähnlich großen Erfolg hatte wie heute PASCAL, ist vermutlich in erster Linie auf den unglücklich gewählten Namen zurückzuführen, der auf eine reine Simulationsprache hindeutet. Vielleicht aber war SIMULA seiner Zeit einfach zu weit voraus.

Da alle verfügbaren SIMULA-Compiler einen gemeinsamen Stammcompiler haben, gibt es keine Probleme mit unterschiedlichen Dialekten, und **Portabilitätsprobleme** sind für SIMULA-Programme recht klein.

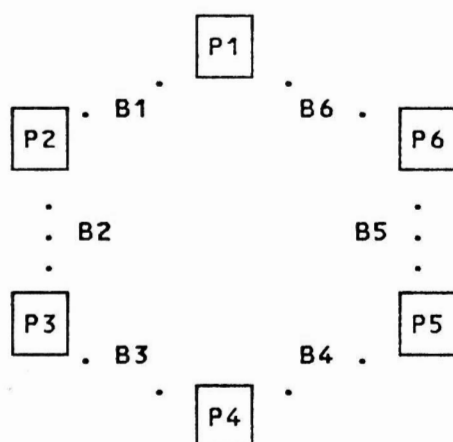
Das hier beschriebene Simulationssystem wurde zumindest indirekt durch den bei den GMD entwickelten Simulator TOCS [MÜH79] beeinflusst, an dessen Entwicklung Michael Spenke beteiligt war.

8.1 Simulation der hardware

Das Simulationssystem ist so **flexibel** angelegt, daß eine Konfiguration aus beliebig vielen Prozessoren und Bussen simuliert werden kann. Der Benutzer kann über eine Eingabedatei dem CSSA-System mitteilen, wie seine Konfiguration aussehen soll (→ C 5.3). Das Spektrum der möglichen Konfigurationen wird dadurch abgegrenzt, daß die Verbindungen zwischen Bussen und Prozessoren in Form einer sogenannten **connection-Matrix** spezifiziert werden: Es wird jeweils zu einem Paar von Prozessoren der Bus angegeben, der sie verbindet. Eine Null (hier ein blank) gibt an, daß keine Verbindung besteht. Die Matrix braucht nicht symmetrisch zu ihrer Hauptdiagonalen zu sein, es kann also für jede Kommunikationsrichtung zwischen zwei Prozessoren einen eigenen Bus (aber höchstens einen) geben.

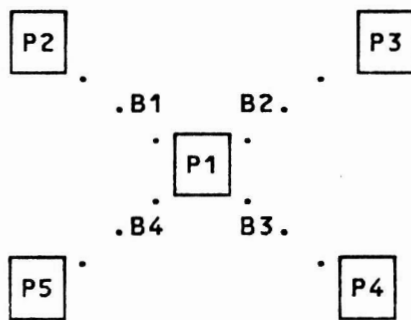
Die folgende Abbildung zeigt einige simulierbare Konfigurationen, jeweils mit der zugeordneten connection-Matrix:

Ring:



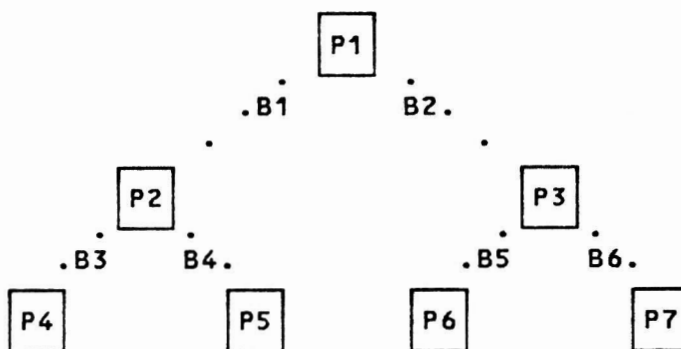
	P1	P2	P3	P4	P5	P6
P1		1				6
P2	1		2			
P3		2		3		
P4			3		4	
P5				4		5
P6	6				5	

Stern:



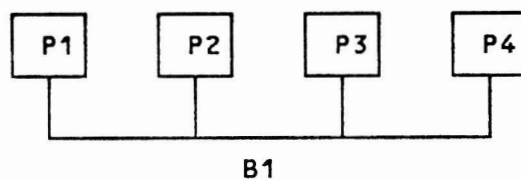
	P1	P2	P3	P4	P5
P1		1	2	3	4
P2	1				
P3	2				
P4	3				
P5	4				

Baum:



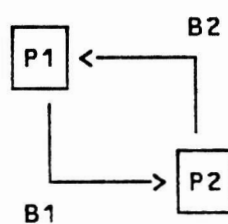
	P1	P2	P3	P4	P5	P6	P7
P1		1	2				
P2	1			3	4		
P3	2					5	6
P4		3					
P5		4					
P6			5				
P7			6				

Zentraler Bus:



	P1	P2	P3	P4
P1		1	1	1
P2	1		1	1
P3	1	1		1
P4	1	1	1	

Doppelverbindung:



	P1	P2
P1		1
P2	2	

Für das verteilte Betriebssystem wird außerdem noch eine zweite Matrix benötigt, die **routing-Matrix**. Sie beschreibt, auf welchen Wegen Nachrichten durch das Netz gereicht werden. Die Matrix bildet zwei Prozessoren wiederum auf einen Prozessor ab. $\text{ROUTING-MATRIX}(i,j)=k$ bedeutet: Soll eine Nachricht von Prozessor i nach Prozessor j geschickt werden, so muß sie zunächst nach Prozessor k geschickt werden, zu dem eine direkte Busverbindung besteht. Auf diese Weise werden die Kommunikationswege **statisch** - also bei der Systemgenerierung - festgelegt. Ein **dynamisches routing**, etwa in Abhängigkeit von der Auslastung der Busse, ist nicht vorgesehen.

Am Beispiel der ringförmigen Konfiguration - die als einzige einen Zyklus enthält - sollen zwei verschiedene mögliche routing-Matrizen gezeigt werden:

	P1	P2	P3	P4	P5	P6
P1		2	2	2	2	2
P2	3		3	3	3	3
P3	4	4		4	4	4
P4	5	5	5		5	5
P5	6	6	6	6		6
P6	1	1	1	1	1	

	P1	P2	P3	P4	P5	P6
P1		2	2	2	6	6
P2	1		3	3	3	1
P3	2	2		4	4	4
P4	5	3	3		5	5
P5	6	6	4	4		6
P6	1	1	1	5	5	

Durch die erste Matrix wird festgelegt, daß alle Nachrichten **gegen den Uhrzeigersinn** geschickt werden, während im zweiten Fall die Nachrichten jeweils den **kürzesten Weg** nehmen (im Zweifelsfall gegen den Uhrzeigersinn).

In der Simulation sind Prozessoren und Busse **aktive** Instanzen, sie werden also als Objekte der SIMULA-Oberklasse PROCESS dargestellt, während Nachrichten **passive** Objekte sind. Beim Versenden einer Nachricht an einen Prozessor wird zunächst nachgesehen, an welchen direkt benachbarten Prozessor die Nachricht gehen soll. Mit dieser Information kann anschließend ermittelt werden, welcher Bus dazu verwendet werden muß. Die Nachricht wird dann in die Warteschlange des Datenbusses eingetragen. In Abhängigkeit von der Länge der Warteschlange muß nun die Nachricht eine gewisse Zeit auf ihre Übertragung warten. Die eigentliche Übertragungszeit für eine Nachricht ist vom Benutzer (unabhängig von der Länge der Nachricht) fest vorgegeben, eine detailliertere Simulation wäre aber hier leicht möglich. Nach dem Verbrauch der Übertragungszeit (durch HOLD) übergibt der Bus die Nachricht an den Zielprozessor und weckt ihn, wenn nötig, per externem interrupt auf (in SIMULA durch ACTIVATE).

Weitere Einzelheiten des unterbrechungsgesteuerten Betriebssystems sind im nächsten Kapitel dargestellt.

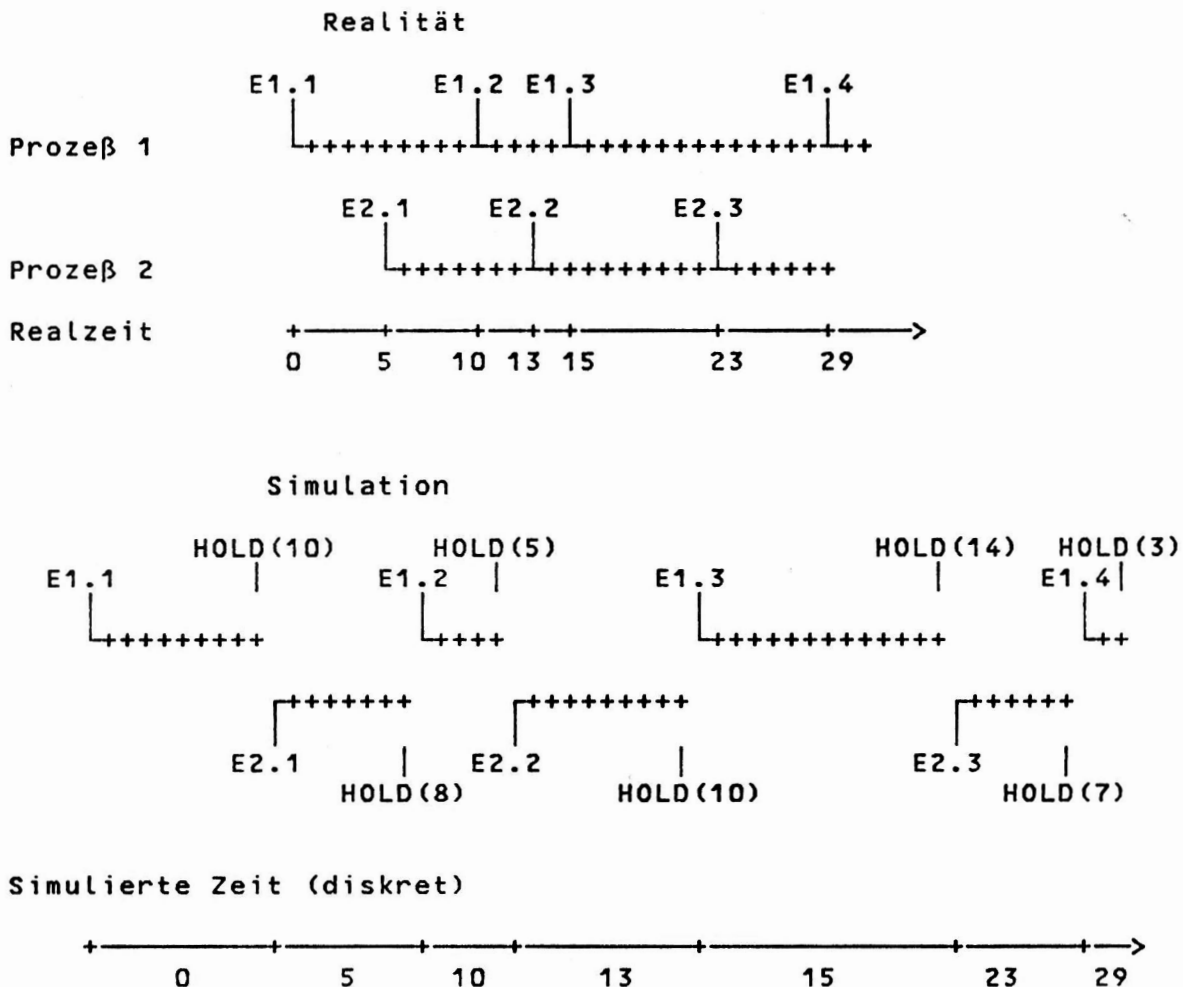
8.2 Simulation des Betriebssystems

In diesem Abschnitt soll das auf die einzelnen Prozessoren verteilte **unterbrechungsgesteuerte multiprogramming-Betriebssystem** beschrieben werden. Zu den Aufgaben des Betriebssystems gehören das Gründen und Terminieren von Agenten, sowie der Nachrichtentransport; außerdem soll es möglich sein, daß mehr Agenten als Prozessoren existieren, das Betriebssystem muß also die quasi-parallele Ausführung mehrerer Agenten auf einem Prozessor unterstützen. Arbeiten auf einem Prozessor mehrere Agenten gleichen Typs, so soll der zugehörige script-code nur einmal vorhanden sein und von den Agenten abwechselnd benutzt werden (code-sharing, reentrant code).

Bei der **Abbildung auf SIMULA** werden Prozessoren und Busse - wie oben bereits erwähnt - als Objekte der Oberklasse PROCESS dargestellt. Alle scripts haben die gemeinsame Oberklasse PROCESS CLASS MODULE, sind also ebenfalls aktive Einheiten der Simulation. Während ein Prozessor Aktionen des Betriebssystems ausführt, sind alle seine scripts passiviert (d.h. nicht im SIMULA-sequencing-set); anschließend wird eines der scripts aktiviert (durch ACTIVATE) und der Prozessor übergibt bis zur nächsten Unterbrechung die Kontrolle an das script (durch HOLD). Das Hin- und Herschalten zwischen Prozessor und script spiegelt die abwechselnde Ausführung von Befehlen des Betriebssystems (im Supervisorstatus) und der Benutzerprogramme (im Normalmodus) wieder.

An dieser Stelle erscheinen einige Anmerkungen zum **Prinzip der Simulation** mit SIMULA angebracht, grundsätzliche Kenntnisse des Simulationskonzepts werden hier allerdings vorausgesetzt. SIMULA unterstützt eine sogenannte **ereignisgesteuerte Simulation**: Um ein System nebenläufiger Prozesse durch ein (sequentielles) Simulationsprogramm zu modellieren, werden innerhalb der Prozesse die **relevanten Ereignisse** identifiziert, die eine Wechselwirkung mit der Umwelt (z.B. Ein-/Ausgabe) oder mit anderen Prozessen haben. Zwischen diesen Ereignissen liegen in der Regel eine Reihe von Aktionen, die lediglich den lokalen Zustand des Prozesses betreffen oder nur in sofern modelliert werden, als sie eine gewisse Zeit benötigen. In der Simulation führt ein Prozeß jeweils ein solches Ereignis und alle dahinterliegenden **unbedeutenden Aktionen** aus und verbraucht anschließend die bis zum nächsten relevanten Ereignis benötigte **Simulationszeit** durch einen HOLD-Aufruf. Dadurch geht die Kontrolle durch einen Zeitsprung an einen anderen Prozeß, der seinerseits bis zum nächsten HOLD aktiv ist. Logisch gesehen ist also ein Prozeß, der nicht an erster Stelle im sequencing-set (SQS) steht, mit Aktionen beschäftigt, die keine Wechselwirkungen mit anderen Prozessen haben. Durch dieses Prinzip werden die Ereignisse, die als wichtig erkannt wurden, in der gleichen Reihenfolge (und zu den gleichen Zeitpunkten) ausgeführt, wie es in der Realität der Fall wäre, während die Reihenfolge der unbedeutenden Aktionen nicht der Realität entspricht. Das folgende Bild soll diesen Zu-

sammenhang verdeutlichen:



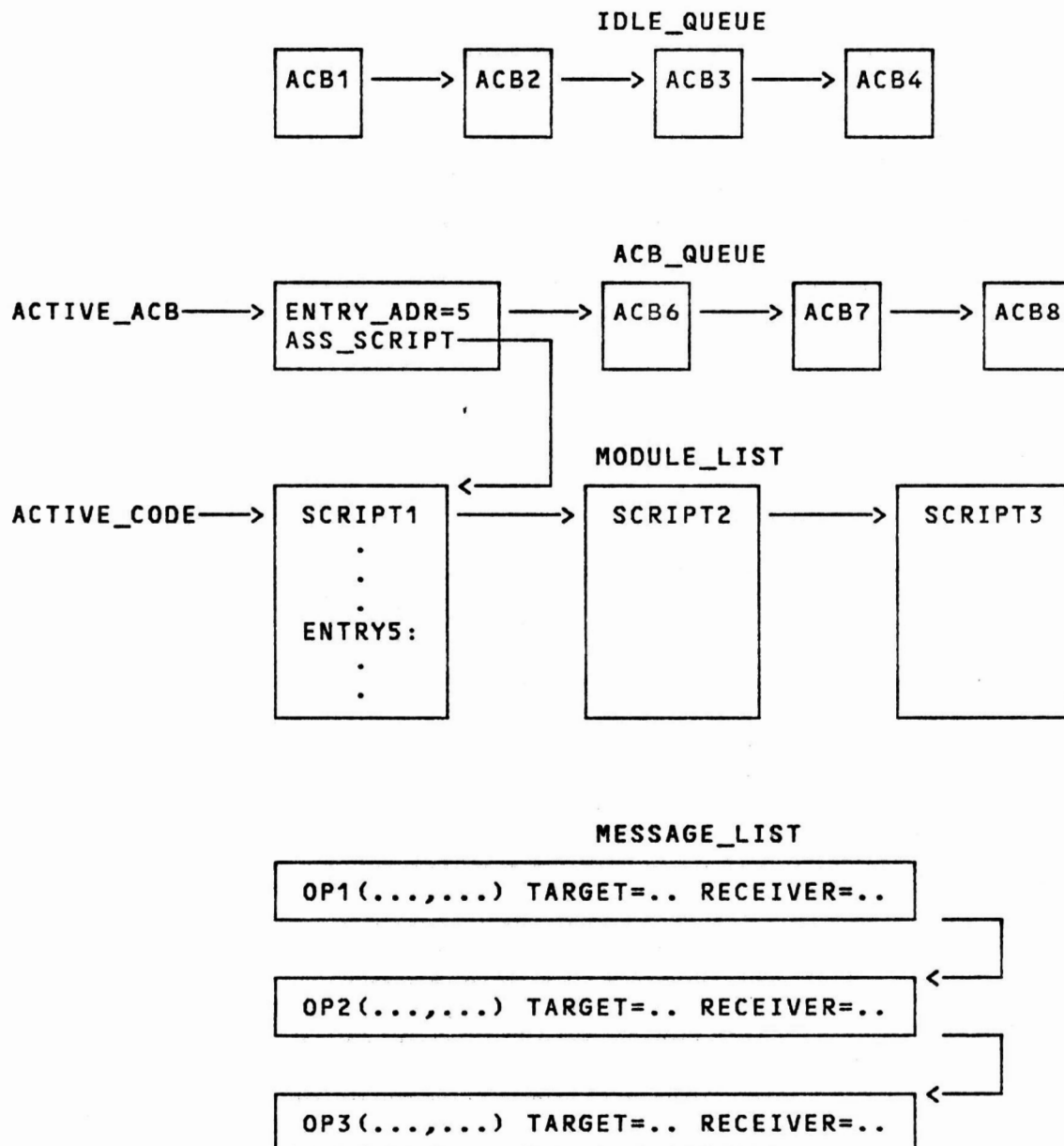
Bei den **Bussen** wurden die Entnahme einer Nachricht aus der Warteschlange und die Übergabe einer Nachricht an einen Prozessor als relevante Ereignisse identifiziert, während das eigentliche Übertragen der Nachricht durch einen HOLD-Aufruf dargestellt wird.

Bei den **Prozessoren** ist die Abarbeitung einer Unterbrechung das relevante Ereignis; die Zeit, in der eines der scripts rechnet, wird durch ein HOLD überbrückt.

Die vom Compiler generierten **script-Module** bestehen im wesentlichen aus code für die einzelnen CSSA-Statements. In den script-code sind aber von Zeit zu Zeit HOLD-Aufrufe eingestreut, durch die jeweils für die zuletzt ausgeführten Befehle Simulationszeit verbraucht wird. Entsprechend den oben genannten Prinzipien der ereignisgesteuerten Simulation erfolgen HOLD-Aufrufe immer unmittelbar bevor ein Agent eine Aktion ausführt, die eine Wechselwirkung mit der Umwelt hat. Solche Aktionen sind im wesentlichen: Versenden einer Nachricht, Entnahme einer Nachricht aus der mailbox, Gründen eines neuen Agenten und Ausgabe von Werten durch 'print'. Der Zeitpunkt der Ausführung von CSSA-Statements, die nur den lokalen Zustand des Agenten betreffen (z.B. assignment) wird nicht unbedingt exakt simuliert.

Die **Agenten** werden im Betriebssystem in Form passiver **Agenten-Kontrollblöcke** (ACB's) dargestellt (s.u.), die alle Informationen über den Zustand eines Agenten enthalten. Dort finden sich die mailbox, ein Verweis auf die Spitze des Laufzeitkellers mit allen lokalen Variablen des Agenten, sowie zu jedem CSSA-Datentyp ein stack für die Evaluierung von Ausdrücken. Vor allem aber enthält jeder ACB einen Verweis auf den zugehörigen script-code und die Einsprungadresse (entsprechend dem lokalen Befehlszähler), mit der der code bei der nächsten Aktivierung durch das Betriebssystem fortgesetzt werden muß.

Einen Überblick über die verschiedenen Warteschlangen des Betriebssystems **innerhalb eines Prozessors** gibt die folgende Abbildung:



Die **ACB_QUEUE** enthält alle Agenten, die sich um die Prozessorzuteilung bewerben. Der erste dieser Warteschlange, der **ACTIVE_ACB**, rechnet gerade. Die **IDLE_QUEUE** enthält alle Agenten, die momentan nicht rechnen können, da keine passende Nachricht vorliegt. Die **MODULE_LIST** enthält alle scripts, zu denen es momentan Agenten gibt. In der **MESSAGE_LIST** stehen alle Nachrichten, die seit der letzten Unterbrechung dem Prozessor von den Datenbussen zugestellt wurden. Die Nachrichten enthalten u.a. auch die Größen **TARGET**, die jeweils den Prozessor benennt, an den die Nachricht geroutet wird, und **RECEIVER**, durch die der Zielagent identifiziert wird. Die **MESSAGE_LIST** ist nicht zu verwechseln mit der mailbox eines Agenten; das lokale Betriebssystem des Prozessors verteilt die Nachrichten aus der **MESSAGE_LIST**, die an eigene Agenten gerichtet sind, an die einzelnen Agenten oder routet die Nachricht über einen Bus an einen anderen Prozessor weiter.

Im folgenden sind die wichtigsten **Komponenten des Simulationssystems** abgedruckt: Die drei **PROCESS**-Klassen **PROCESSOR**, **BUS** und **MODULE**, sowie die **CLASS ACB**. Nebensächliche Details, wie code für die Protokollierung, Statistiken und Fehlermeldungen, wurden weggelassen, damit sie nicht den Blick auf das Wesentliche verschleiern (diese Details finden sich in Teil E Band II). Anschließend werden ausführliche Erklärungen gegeben. Es wurde hier bewußt einmal ein **Programmtext** abgedruckt, um zu zeigen, daß ein Simulationsprogramm mit den mächtigen Konstrukten von **SIMULA** hervorragend geeignet ist, ein Betriebssystemkonzept anschaulich und dennoch exakt darzustellen.

PROCESS CLASS BUS;

```
BEGIN
  REF(HEAD) MESSAGE_LIST; REF(MSG) M;
  PROCEDURE STORE_MESSAGE(M); REF(MSG) M;
    M.INTO(MESSAGE_LIST);
  WHILE TRUE DO
    BEGIN
      M:=MESSAGE_LIST.FIRST;
      IF M==NONE
        THEN PASSIVATE COMMENT WAIT FOR EXTERNAL INTERRUPT;
      ELSE BEGIN
        HOLD(TRANS_TIME);
        M.INTO(M.TARGET.MESSAGE_LIST);
        IF M.TARGET.IDLE
          THEN ACTIVATE M.TARGET AFTER CURRENT;
        END;
      END;
    END;
  END BUS;
```

Die **Datenbusse** arbeiten nach einem sehr einfachen Prinzip: Sie verfügen über eine Liste aller zu versendenden Nachrichten (**MESSAGE_LIST**), in die die Prozessoren mit Hilfe der Prozedur **STORE_MESSAGE** Nachrichten eintragen können. Ein Bus entnimmt jeweils die erste Nachricht der Warteschlange (**FIFO**). Ist die queue leer, so passiviert der Bus, bis er von einem Prozessor wieder aufgeweckt wird, der eine Nachricht verschicken will. Ist eine Nachricht zu verschicken, so verbraucht er die Übertragungszeit (durch **HOLD(TRANS_TIME)**) und trägt anschließend die Nachricht in die Warteschlange des Zielprozessors ein. Ist der

Zielprozessor idle, so wird er durch eine externe Unterbrechung vom Bus aktiviert.

Hier nicht gezeigt ist, daß die Datenbusse auf Wunsch des Benutzers auch eine andere als die FIFO-Strategie verfolgen können: Es wird dann per Zufall irgendeine Nachricht aus der Warteschlange ausgewählt, so daß Überholungen möglich sind, und getestet werden kann, welche Auswirkungen das auf parallele Algorithmen hat. Andere kompliziertere **Abfertigungsstrategien**, die vielleicht die Länge der Nachrichten und unterschiedliche Übertragungszeiten berücksichtigen, könnten leicht implementiert und getestet werden.

Ebenfalls nicht gezeigt ist, welche **Statistiken** über die Auslastung der Busse geführt werden: Neben der Gesamtzahl aller verschickten Nachrichten wird auch der Prozentsatz der Zeit festgehalten, in der der Bus beschäftigt ist (utilization), sowie die durchschnittliche Länge der Warteschlange bei Eintreffen einer Nachricht.

```
LINK CLASS ACB (ASS_SCRIPT);
  REF(MODULE) ASS_SCRIPT;
BEGIN
  REF(PROCESSOR) ASS_PROCESSOR;
  INTEGER ENTRY_ADR; COMMENT ENTRY_ADDRESS IN ASS_SCRIPT;
  REAL LEFT_TIME; COMMENT REMAINING SIMULATION TIME;
  REF(HEAD) MAILBOX;
  REF(AS) AKT_AS; COMMENT TOP OF RUNTIME STACK;
  REF(BOOLEL) BOOL_TOP;
  REF(STRINGEL) STRING_TOP;
  REF(AGENTEL) AGENT_TOP;
  REF(OPEREL) OPER_TOP;
  REF(SCRIPTEL) SCRIPT_TOP;
  REF(REALLEL) REAL_TOP;
  REF(INTELEL) INT_TOP;
  REF(INTELEL) ENUM_TOP;
  REF(MSGEL) MSG_TOP;

  MAILBOX := NEW HEAD;
END ACB;
```

Ein **ACB** enthält unter anderem die mailbox, einen Verweis auf den obersten Aktivierungssatz im Laufzeitkeller, sowie die stacks zur Evaluierung von Ausdrücken. **LEFT_TIME** gibt an, wieviel simulierte Zeit der Agent noch zu verbrauchen hat (genauer s.u.).

```
PROCESS CLASS MODULE;
BEGIN INTEGER NR_OF_ACBS;
  INTEGER OWNMODE;
  REF(PROCESSOR) ASS_PROCESSOR;
END MODULE;
```

MODULE ist die Oberklasse für alle vom Compiler generierten scripts. Hier wird ein Verweis auf den zugeordneten Prozessor gespeichert, sowie die Anzahl der zu diesem script existierenden (gleichartigen) Agenten. **OWNMODE** bezeichnet den zugehörigen script-Typ.

PROCESS CLASS PROCESSOR;

```

BEGIN REF(ACB) ACTIVE_ACB; REF(MODULE) ACTIVE_CODE;
REF(HEAD) ACB_QUEUE, IDLE_QUEUE, MODULE_LIST, MESSAGE_LIST;
INTEGER INTERRUPT_TYPE;
SWITCH INTERRUPT:= LTIMESLICE_RUNOUT,    LSEND_MESSAGE,
                  LRECEIVE_MESSAGE,    LCREATE_AGENT,
                  LTERMINATION,        LAGENT_IDLE;

```

THE ACTIVE_ACB (IF ANY) IS MOVED TO THE END OF THE ACB_QUEUE.
THE NEW ACTIVE_ACB IS SELECTED FROM THE ACB_QUEUE. THIS SIMPLE
SCHEDULER ALWAYS ACTIVATES THE FIRST ACB IN THE ACB_QUEUE.

```

SCHEDULER: IF INTERRUPT_TYPE/=0
                THEN GOTO INTERRUPT(INTERRUPT_TYPE);
                IF NOT MESSAGE_LIST.EMPTY THEN GOTO LRECEIVE_MESSAGE;
                INSPECT ACTIVE_ACB DO INTO(ACB_QUEUE);
                ACTIVE_ACB:=ACB_QUEUE.FIRST;
                IF ACTIVE_ACB==NONE THEN GOTO LIDLE;
                ACTIVE_CODE:=ACTIVE_ACB.ASS_SCRIPT;

```

THE SELECTED ACB IS ACTIVATED.
IN ADDITION A TIMER IS SET SO THAT THE PROCESSOR IS ACTIVATED
AFTER AT LEAST ONE TIME_SLICE, EVEN IF NO INTERRUPT OCCURS.

```

INITIATOR: ACTIVATE ACTIVE_CODE DELAY ACTIVE_ACB.LEFT_TIME;
                INTERRUPT_TYPE:=TIMESLICE_RUNOUT;
                HOLD(TIMESLICE);

```

NOW ONE OF THE AGENTS IS ACTIVE UNTIL HIS NEXT INTERRUPT
("REACTIVATE ASS_PROCESSOR") OR TIMESLICE_RUNOUT

```

DEACTIVATE: ACTIVE_ACB.LEFT_TIME:=ACTIVE_CODE.EVTIME-TIME;
                CANCEL(ACTIVE_CODE);

```

THE PROCESSOR NOW REACTS ACCORDING TO THE INTERRUPT TYPE.
AFTERWARDS THE SCHEDULER WILL SELECT THE NEXT ACTIVE_ACB.

```

GOTO INTERRUPT(INTERRUPT_TYPE);

```

THE ACTIVE_ACB'S MESSAGE IS SENT OFF TO ANOTHER PROCESSOR

```

LSEND_MESSAGE: INTERRUPT_TYPE:=0;
BEGIN REF(PROCESSOR) TARGET;
REF(MSG) M; REF(BUS) B;
M:=ACTIVE_ACB.MSG_TOP.WERT;
TARGET:=M.RECEIVER.ASS_PROCESSOR;
IF TARGET==THIS_PROCESSOR THEN M.INTO(MESSAGE_LIST)
ELSE BEGIN
    TARGET:=
        PROCESSORS(ROUTING_MATRIX(THIS_PROCESSOR.ID,TARGET.ID));
    M.TARGET:=TARGET;
    B:=BUSES(CONNECTION_MATRIX(THIS_PROCESSOR.ID,TARGET.ID));
    B.STORE_MESSAGE(M);
    IF B.IDLE THEN ACTIVATE B AFTER CURRENT;
END;
END;
GOTO SCHEDULER;

```


THE PROCESSOR HAS RECEIVED A MESSAGE AND ROUTES IT TO THE MAILBOX OF THE ADDRESSED AGENT OR PASSES IT TO ANOTHER PROCESSOR.

```

LRCEIVE_MESSAGE: INTERRUPT_TYPE:=0;
BEGIN REF(ACB) RECEIVER; REF(PROCESSOR) TARGET;
  REF(MSG) M; REF(BUS) B;
  M:=-MESSAGE_LIST.FIRST;
  RECEIVER:=-M.RECEIVER;
  TARGET:=-RECEIVER.ASS_PROCESSOR;
  IF TARGET/= THIS PROCESSOR THEN
  BEGIN
    TARGET:-
      PROCESSORS(ROUTING_MATRIX(THIS PROCESSOR.ID,TARGET.ID));
    M.TARGET:-TARGET;
    B:-BUSSES(CONNECTION_MATRIX(THIS PROCESSOR.ID,TARGET.ID));
    B.STORE_MESSAGE(M);
    IF B.IDLE THEN ACTIVATE B AFTER CURRENT;
  END ELSE BEGIN
    MESSAGE_LIST.FIRST.INTO(RECEIVER.MAILBOX);
    RECEIVER.INTO(ACB_QUEUE);
    COMMENT POSSIBLY OUT OF IDLE_QUEUE ;
  END;
END;
GOTO SCHEDULER;

```

THE ACTIVE AGENT CREATES A NEW AGENT. THE PROCESSOR CHECKS, IF HE HAS A MODULE OF THE REQUESTED TYPE IN HIS STORAGE. IN THIS CASE HE NEED NOT LOAD A NEW MODULE (SCRIPT) SINCE SCRIPTS ARE REENTRANT (CAN BE SHARED BY SEVERAL AGENTS)

```

LCREATE_AGENT: INTERRUPT_TYPE:=0;
BEGIN REF(MODULE) M; INTEGER SCR_NR; REF(ACB) A; REF(PROCESSOR) P;
  P:-PROCESSORS(NEXT_CREATION_PROC);
  SCR_NR:=ACTIVE_ACB.SCRIPT_TOP.WERT;
  IF P.MODULE_LIST.FIRST==NONE THEN GOTO NOT_FOUND;
  FOR M:-P.MODULE_LIST.FIRST,M.SUC WHILE M/= NONE DO
    IF M.OWNMODE=SCR_NR THEN GOTO FOUND;
  NOT_FOUND: M:-NEW_MODULE(SCR_NR);
    M.INTO(P.MODULE_LIST);
  FOUND: A:-NEW_ACB(M); A.INTO(P.ACB_QUEUE);
    M.NR_OF_ACBS:=M.NR_OF_ACBS+1;
    M.ASS_PROCESSOR:-P;
    A.ASS_PROCESSOR:-P;
    ACTIVE_ACB.MSG_TOP.WERT.INTO(A.MAILBOX);
    AGENT_PUSH(A);
    NEXT_CREATION_PROC:=NEXT_CREATION_PROC+1;
    IF NEXT_CREATION_PROC>NR_OF_PROCESSORS
      THEN NEXT_CREATION_PROC:=2;
    IF P.IDLE THEN ACTIVATE P AFTER CURRENT;
  END;
GOTO SCHEDULER;

```


THE ACTIVE AGENT IS TERMINATED . THE PROCESSOR CAN DELETE THE AGENTS ACB. IT IS TESTED, IF THE DELETED ACB WAS THE ONLY ACB ASSOCIATED TO ITS SCRIPT. IN THIS CASE THE SCRIPT IS DELETED AS WELL.

LTERMINATION:

```
INTERRUPT_TYPE:=0;
  ACTIVE_ACB.ASS_PROCESSOR:=-NONE;
  ACTIVE_ACB.OUT; COMMENT ...OF ACB_QUEUE;
  ACTIVE_ACB:=-NONE;
  ACTIVE_CODE.NR_OF_ACBS:=ACTIVE_CODE.NR_OF_ACBS-1;
  IF ACTIVE_CODE.NR_OF_ACBS=0
    THEN BEGIN ACTIVE_CODE.OUT; COMMENT ... OF MODULE_LIST;
              ACTIVE_CODE:=-NONE;
            END;
GOTO SCHEDULER;
```

THE INTERRUPTING AGENT IS IDLE BECAUSE HE HAS NO MATCHING MESSAGE IN HIS MAILBOX. THEREFORE HE JOINS THE IDLE_QUEUE UNTIL HE RECEIVES ANOTHER MESSAGE

LAGENT_IDLE:

```
INTERRUPT_TYPE:=0;
  ACTIVE_ACB.INTO(IDLE_QUEUE);
  ACTIVE_ACB:=-NONE;
GOTO SCHEDULER;
```

THE ACTIVE AGENT HAS WORKED FOR A FULL TIME_SLICE WITHOUT INTERRUPTING THE PROCESSOR. ANOTHER AGENT WILL BE ACTIVATED NOW

LTIMESLICE_RUNOUT:

```
INTERRUPT_TYPE:=0;
  COMMENT NO ACTION;
GOTO SCHEDULER;
```

THE PROCESSOR IS IDLE BECAUSE ALL HIS AGENTS ARE IDLE. HE HAS TO WAIT FOR AN EXTERNAL INTERRUPT (E.G. RECEIVE MESSAGE) THAT ALLOWS ONE OF THE AGENTS TO GO ON .

LIDLE:

```
INTERRUPT_TYPE:=0;
  PASSIVATE; COMMENT WAIT FOR EXTERNAL INTERRUPT;
GOTO SCHEDULER;

END PROCESSOR;
```

In der **CLASS PROCESSOR**, mit der der **Betriebssystemkern** simuliert wird, stehen zunächst die Deklarationen der verschiedenen lokalen Warteschlangen und die Deklaration eines weiter unten benötigten **switches**, mit dessen Hilfe in Abhängigkeit von der Art der Unterbrechung zum entsprechenden **interrupt-handler** verzweigt werden kann.

Es folgt der zentrale Teil des Betriebssystems, der immer wieder (nach jeder Unterbrechung) durchlaufen wird: der **Scheduler**. Hier wird zunächst getestet, ob noch eine Unterbrechung vorliegt, und falls das der Fall ist, entsprechend verzweigt. Nach Abarbeitung der Unterbrechung kehrt die Kontrolle in jedem Falle an den Anfang des Schedulers zurück. Falls keine Unterbrechung vorliegt, wird getestet, ob vor der letzten Unterbrechung Nachrichten von einem Datenbus eingetroffen sind; diese werden dann empfangen, d.h. an einen lokalen Agenten übergeben bzw. an einen benachbarten Prozessor weitergereicht.

Sobald alle Unterbrechungen abgearbeitet und alle Nachrichten empfangen sind, kann der nächste Agent aus der Warteschlange ausgewählt werden. Dazu wird zunächst der gerade aktive ACB – sofern es einen solchen gibt – vom Anfang der Warteschlange entfernt und am Ende wieder eingefügt. Als neuer aktiver Agent wird nun der erste ACB in der Schlange ausgewählt. Als zu aktivierender Code wird das dem Agenten zugeordnete Script ausgewählt. Ist überhaupt kein rechenwilliger Agent vorhanden, so ist der Prozessor **idle** und muß auf einen externen **interrupt** warten (**LIDLE**).

Wie man sieht, werden bei diesem einfachen Scheduler die rechenwilligen Agenten reihum bedient ("round-robin"). Auf die Implementation eines komplizierteren Verfahrens wurde zunächst verzichtet, da noch keinerlei Erfahrungen über eine speziell für CSSA geeignete Strategie vorliegen.

Der vom Scheduler ausgewählte Agent wird nun **aktiviert**. Er wird aber nicht an erster Stelle in den **sequencing-set** eingetragen, sondern muß zunächst noch die von der letzten Aktivierung verbliebene Simulationszeit (**s.u.**) verbrauchen (**DELAY LEFT_TIME**). Der Prozessor deaktiviert sich (**HOLD**) und erhält spätestens nach einer Zeitscheibe die Kontrolle zurück. Es ist aber auch möglich, daß der rechnende Agent noch vor Ablauf des **timers per interrupt** einen Betriebssystemdienst anfordert (der Betriebssystemkern wird dann mit **'REACTIVATE ASS_PROCESSOR'** vom Agenten vorzeitig aktiviert, **INTERRUPT_TYPE** wird vorher entsprechend der Unterbrechungsart gesetzt; vgl. dazu auch den weiter unten angegebenen **script-code** bei Betriebssystemaufrufen).

Nach der Unterbrechung wird zunächst der aktive Agent **deaktiviert** (durch **CANCEL**). Da er eventuell (bei einem **timeslice-runout**) weiter hinten im **sequencing-set** steht, wird die noch verbleibende Simulationszeit in den ACB eingetragen, damit sie bei der nächsten Aktivierung (mit **DELAY**) verbraucht werden kann. (**TIME** ist die momentane Simulationszeit, und **EVTIME** gibt an, zu welchem Zeitpunkt der Agent wieder ganz vorne im **sequencing-set** stehen wird.) Es sei hier daran erinnert, daß der rechnende Agent jeweils einige CSSA-Statements ausführt und erst anschließend dafür Simulationszeit verbraucht (durch **HOLD**). Im Moment des **timer-interrupts** hat er also in der Regel noch nicht die ge-

samte Simulationszeit verbraucht, das wird dann bei der nächsten Aktivierung nachgeholt. Allerdings hat der Agent dann bereits einige Statements ausgeführt, für die noch keine Simulationszeit verbraucht wurde, die logisch also noch in der Zukunft liegen. Um die **äußere Zeitkonsistenz** des Systems zu gewährleisten, wird vor Statements, die über den lokalen Zustand des Agenten hinaus Auswirkungen haben (z.B. send-Statement) stets erst die für die bereits ausgeführten Statements benötigte Zeit per HOLD auch in simulierter Form verbraucht.

Das folgende Beispiel zeigt den script-code an einer Stelle, wo die Ausführung unterbrochen werden kann:

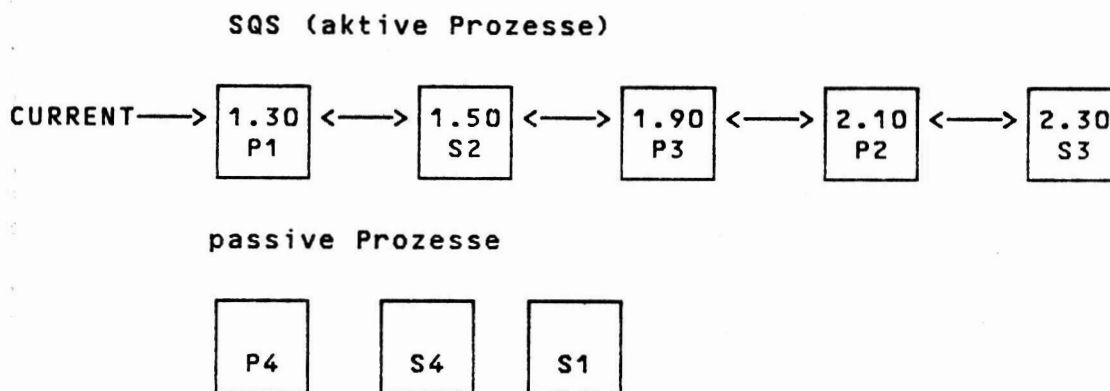
```
ACTIVE_ACB.ENTRY_ADR:=5  
Hold(ELAPSED_TIME);  
goto ENTRY(ACTIVE_ACB.ENTRY_ADR);  
ENTRY5:
```

In einer realen Implementierung würde bei der Deaktivierung der Befehlszähler (und die Registerinhalte) in den ACB gerettet und später wieder geladen. In der Simulation jedoch trägt der Agent selbst schon vor seinem HOLD-Aufruf eine eindeutige Nummer als ENTRY_ADDRESS in seinen ACB ein. Nach dem HOLD wird durch einen berechneten Sprung entsprechend der ENTRY_ADDRESS des ACTIVE_ACB verzweigt. Dadurch wird für den Fall Sorge getragen, daß der Agent, während er (durch HOLD) simulierte Zeit verbraucht, deaktiviert wird, und ein anderer Agent (gleichen Typs) den script-code an einer anderen Stelle fortsetzt. (Man beachte hierbei, daß der script-code reentrant und sharable ist, daß es aber für jeden Agenten, also für jede Instanz eines scripts, einen eigenen ACB gibt und daß ACTIVE_ACB nach dem HOLD einen anderen ACB bezeichnen kann als vorher - nämlich genau dann, wenn zwischenzeitlich der Scheduler einen anderen Agenten gleichen Typs ausgewählt hat!)

Es sei noch angedeutet, wie der script-code an einer Stelle, an der ein **Betriebssystemaufruf** erfolgt, sinngemäß aussieht:

```
ACTIVE_ACB.ENTRY_ADR:=9  
ASS_PROCESSOR.INTERRUPT_TYPE:=SEND_MESSAGE;  
reactivate ASS_PROCESSOR;  
goto ENTRY(ACTIVE_ACB.ENTRY_ADR);  
ENTRY9:
```

Die folgende Abbildung zeigt eine **Momentaufnahme der Simulation**, wie sie sich auf der Ebene der CLASS SIMULATION darstellt:



Der Einfachheit halber wurde ein Beispiel gewählt, bei dem vier Prozessoren (P1,P2,P3,P4) existieren und auf jedem Prozessor nur ein script (S1,S2,S3,S4) läuft (durchaus aber evtl. mehrere Agenten). Die Busse werden hier nicht dargestellt. P1 ist gerade (zum Zeitpunkt 1.3) aktiv und führt Aktionen des Betriebssystems aus. Er hat soeben das script S1 deaktiviert und somit aus dem SQS entfernt (durch CANCEL). Das script S2 wird zum Zeitpunkt 1.5 (also noch vor Ablauf der Zeitscheibe in P2) die Kontrolle erhalten. Es wird dann einige CSSA-Statements ausführen und anschließend (durch HOLD) im SQS weiter nach hinten wandern. Es kann aber auch einen Betriebssystemdienst (bspw. SEND-MESSAGE) per interrupt anfordern und dadurch P2 noch vor Ablauf des timers wieder aktivieren (durch REACTIVATE). Das script S3 steht im SQS **hinter** seinem Prozessor. Es muß noch eine Zeiteinheit Simulationszeit für Statements verbrauchen, die es bereits ausgeführt hat. Erst dann kann es die nächste Aktion ausführen. Logisch gesehen ist S3 also noch mit der Ausführung von Befehlen beschäftigt, die lediglich den lokalen Zustand des Agenten verändern. Der Prozessor P3 wird zum Zeitpunkt 1.9 die Kontrolle erhalten (timeslice-runout) und S3 deaktivieren. Die verbleibende Simulationszeit von 0.4 wird dann im ACB des Agenten vermerkt. Wird anschließend ein anderer Agent gleichen Typs durch P3 aktiviert, so wird S3 hinter dem HOLD-Aufruf fortgesetzt, wo sofort zu der im neuen ACB eingetragenen ENTRY-ADDRESS verzweigt wird. P4 ist idle, da keiner der Agenten zu S4 rechenwillig ist.

Der **Zeitverbrauch** wird nur in sehr einfacher Form simuliert: Für jedes CSSA-Statement werden 0.1 Zeiteinheiten verbraucht; für die Aktionen des Betriebssystems wird keine Zeit verbraucht. Eine detailliertere Simulation wäre aber leicht möglich.

Nach diesen Erläuterungen zum Prinzip der Simulation wird nun wieder mit der Beschreibung des BS-Kerns fortgefahren.

Nach der Deaktivierung des aktiven Agenten wird je nach Art der Unterbrechung zum entsprechenden **interrupt-handler** verzweigt. Nach Abarbeitung der Unterbrechung kehrt die Kontrolle zum Scheduler zurück, und es wird wieder ein Agent aktiviert.

Hinter der Marke **LSEND_MESSAGE** wird eine Nachricht des zuletzt aktiven Agenten versendet. Die zu versendende Nachricht steht als oberste im message-stack des ACB. In der Nachricht sind bereits der Empfänger (**RECEIVER**) und der Zielprozessor (**RECEIVER.ASS_PROCESSOR**) vermerkt. Liegt der Zielagent auf dem selben Prozessor wie der sendende Agent, so kann die Nachricht direkt zugestellt werden. Anderfalls muß aus der routing-Matrix ermittelt werden, an welchen **benachbarten** Prozessor die Nachricht gesandt werden soll, und aus der connection-Matrix, welcher Datenbus dazu verwendet werden muß. Die Nachricht wird in die Warteschlange des Busses eingetragen, und der Bus wird - wenn nötig - per interrupt aufgeweckt.

Zu der Marke **LRECEIVE_MESSAGE** gelangt die Kontrolle, wenn durch den Prozessor selbst oder asynchron von einem Datenbus Nachrichten in die **MESSAGE_LIST** eingetragen wurden. Die erste Nachricht wird entnommen und festgestellt, ob der Zielagent auf diesem oder einem anderen Prozessor läuft. Dementsprechend wird die Nachricht über den entsprechenden Bus an einen anderen Prozessor weitergesandt oder direkt in die mailbox des Zielagenten eingetragen. Der Zielagent wird dann in die Warteschlange der rechenwilligen Agenten eingetragen, da er ja evtl. die eingetroffene Nachricht verarbeiten kann.

Soll ein **neuer Agent** gegründet werden, so geschieht das nicht unbedingt auf dem Prozessor, auf dem auch der erzeugende Agent läuft: Stattdessen werden neue Agenten reihum auf alle Prozessoren verteilt (außer auf Prozessor 1, wo nur der interface-Agent läuft). Aus dem script-stack des erzeugenden Agenten wird entnommen, von welchem Typ der neue Agent sein soll. Existiert auf dem für den neuen Agenten vorgesehenen Prozessor bereits ein Agent gleichen Typs, so muß dort kein neues script-Modul erzeugt werden, da der generierte code **reentrant** ist. In jedem Falle wird ein neuer ACB mit einem Verweis auf das script gegründet und in die **ACB_QUEUE** eingetragen. Die creation-message wird in die mailbox des neuen Agenten gebracht, und ein Verweis auf den neuen Agenten wird dem gründenden übergeben. Schließlich wird - wenn nötig - der Prozessor des neuen Agenten aufgeweckt.

Als zukünftige Erweiterung bietet es sich an, eine bessere Prozessorbelegungsstrategie zu implementieren und das Laden der scripts vom Hintergrundspeicher in die Simulation mit einzubeziehen.

Bei der **Terminierung** eines Agenten wird der zugehörige ACB entfernt. Das zugehörige script wird nur dann gelöscht, wenn kein weiterer Agent gleichen Typs mehr auf diesem Prozessor existiert.

Teilt ein Agent mit, daß er nicht mehr weiterrechnen kann, da keine passenden Nachrichten mehr vorliegen, so wird der ACB in die **IDLE_QUEUE** eingetragen und verbleibt dort, bis einmal eine Nachricht für diesen Agenten eintrifft.

Im Falle eines **timeslice-runout** wird lediglich durch den Scheduler ein anderer Agent (sofern vorhanden) aktiviert.

Hat der Prozessor keinen rechenwilligen Agenten mehr, so passiviert er sich und wartet auf einen externen interrupt eines Datenbusses. An dieser Stelle wird auch – was hier nicht gezeigt ist – die Auslastung des Prozessors berechnet.

Das hier beschriebene Betriebssystem könnte sicherlich an einigen Stellen noch detaillierter simuliert werden; es wurde aber darauf Wert gelegt, zunächst einmal ein einfaches, aber **ausbaufähiges** System zu modellieren, das als Grundlage für weitere Experimente geeignet ist.

8.3 Steuerung des Simulationssystems

Der Ablauf einer Simulation wird vom initial auf Prozessor 1 laufenden **interface-Agenten** aus gesteuert. Der interface-Agent kann nicht nur Kommandos zum Gründen von Agenten und zum Versenden und Empfangen von Nachrichten interpretieren, sondern auch Kommandos zur Steuerung, Überwachung und Protokollierung der Simulation (—> C 5.3). Auf diese Weise wird eine **interaktive Simulation** ermöglicht, d.h. der Benutzer muß den Ablauf nicht von vorneherein festlegen, sondern kann den Fortgang der Simulation beobachten, interessante Passagen besonders genau protokollieren, zu geeigneten Zeitpunkten den Zustand von Agenten, Prozessoren und Bussen inspizieren, die Simulationszeit mal in kleinen, mal in großen Schritten fortschreiten lassen und nach Belieben Agenten stoppen und wieder starten.

Die verschiedenen Formen der **Systemprotokollierung** bis hin zum statementweisem tracing von Agenten und die möglichen Zustandsinformationen sind in Kap. 7.3 und in Teil C dieser Arbeit ausführlich beschrieben. Hier sei nur auf die SYSTEM-option hingewiesen, mit der die Aktionen des Betriebssystems genauestens protokolliert werden können, und auf das SYSSTATUS-Kommando, mit dem Zustand und Auslastung aller Prozessoren und Busse ausgegeben werden kann.

Während der Interpreter Kommandos des Benutzers verarbeitet, bleibt die **simulierte Zeit**, zunächst stehen. Es können daher beispielsweise Agenten gegründet werden, ohne daß sie sofort anfangen zu arbeiten, oder Nachrichten versandt werden, ohne daß sie sofort von den Datenbussen transportiert werden. Erst wenn der Benutzer ein run-Kommando eingibt, gibt der interface-Agent (durch HOLD) die Kontrolle ab, und die Simulation beginnt. Je nachdem, welche Form der Protokollierung gewünscht wurde, werden die Aktionen des Betriebssystems und/oder der Agenten mehr oder weniger detailliert gemeldet. In jeder Meldung wird die simulierte Zeit angegeben. In der Regel wird der Benutzer es nicht wünschen, daß sofort die gesamte verteilte Berechnung simuliert wird. Vielmehr möchte er zu einem ihm geeignet erscheinenden Zeitpunkt wieder die Möglichkeit haben, ein Kommando einzugeben, und so beispielsweise die Form der Protokollierung zu ändern, sich Zustandsinformationen anzeigen zu lassen, oder durch Gründen neuer Agenten oder Versenden von Nachrichten in die Simulation einzugreifen. Es sind daher verschiedene Möglichkeiten vorgesehen, wie der interface-Agent wieder die Kontrolle erhalten kann.

Die einfachste Möglichkeit besteht darin, im run-Kommando eine **Maximalzeit** anzugeben, nach der die Simulation spätestens unterbrochen wird. Die Simulation kann so bis zu einem bestimmten **Zeitpunkt** ausgeführt werden. Oftmals ist es aber für den Benutzer nicht leicht, eine geeignete Zeitspanne anzugeben: Er will ja die Simulation in der Regel nicht bis zu einem bestimmten **Zeitpunkt**, sondern bis zu einem bestimmten **Ereignis** vorlaufen lassen. Aus diesem Grunde wurde eine Methode entwickelt, die es erlaubt, daß der Benutzer den Fortgang der Simulation beobachtet, zu einem beliebigen Zeitpunkt unterbricht und die Kontrolle dadurch an den interface-Agenten zurückkehrt: Es wird ständig überprüft, ob für 0.1 Sekunden simulierte Zeit mehr als 5 Sekunden Realzeit verbraucht wurden; ist das der Fall, so wird die Simulation unterbrochen. Das Überschreiten der **Realzeitgrenze** kann vom Benutzer provoziert werden, indem er nicht sofort weiterblättert, wenn ein Bildschirm vollgeschrieben ist, oder indem er die Programmausführung per interrupt-Taste unterbricht und erst nach 5 Sekunden fortsetzt.

Die Realzeitgrenze kann vom Benutzer verändert werden; beispielsweise ist es sinnvoll, durch einen sehr großen Wert eine Unterbrechung der Simulation völlig auszuschließen, wenn eine vorgefertigte Folge von Interpreterkommendos aus einer Datei gelesen wird.

In jedem Falle wird die Simulationszeit angehalten, wenn das Agentennetz **terminiert** (selbst wenn im run-Kommando eine größere Zeitspanne angegeben wurde). Auf diese Weise wird verhindert, daß die Statistiken über die Auslastung von Bussen und Prozessoren dadurch ihre Aussagekraft verlieren, daß Zeiträume beobachtet werden, in denen gar keine verteilte Berechnung durchgeführt wird.

Die Terminierung des Netzes wird erkannt, wenn alle Busse und alle Prozessoren außer Prozessor 1 (auf dem das interface läuft) im idle-Zustand sind, und Prozessor 1 keine weiterzureichende oder zu verschickende Nachricht mehr in seiner queue hat.

Das hier beschriebene Simulationssystem ist weitaus besser als ein reales Mehrprozessorsystem geeignet, um CSSA-Anwendungen zu testen und zu implementieren: Dadurch, daß Parallelität und verteilter Speicher eben **nur simuliert** werden, können eine Reihe von Test- und debugging-Hilfen zur Verfügung gestellt werden; der Verlauf paralleler Berechnungen kann quasi "in Zeitlupe" beobachtet werden. Vor allem aber stellt das Simulationssystem einen Beitrag zur Lösung einer regelrechten "Verklemmung" dar: Ohne eine geeignete Maschine zur Verarbeitung asynchroner paralleler Algorithmen werden derartige Algorithmen kaum entwickelt und können nicht getestet werden, andererseits besteht kaum die Notwendigkeit zur Entwicklung einer derartigen Maschine, wenn keine Algorithmen und Anwendungen existieren.

8.4 Beispiele

In diesem Abschnitt soll angedeutet werden, welche Art von **Experimenten** mit Hilfe des Simulationssystems durchgeführt werden können. Das in Kapitel 4.2 ausführlich erläuterte Programm zur Berechnung des ggT von N Zahlen wurde auf verschiedenen Konfigurationen ausgeführt; die Ergebnisse werden hier diskutiert. Als Nachrichtenübertragungszeit wurde eine Zeiteinheit angesetzt (zum Vergleich: ein CSSA-Statement benötigt 0.1 ZE); die in Kapitel 4.2 aufgeführten Ergebnisse wurden dagegen mit einer Übertragungszeit von 0.1 Zeiteinheiten erreicht!

Es wurde jeweils der ggT von 108, 76, 12 und 60 berechnet. Zunächst wurde eine **ringförmige Konfiguration** getestet, die genau auf den Anwendungsfall zugeschnitten war: Für jeden der 4 ggT-Prozesse gab es einen eigenen Prozessor, und die Anordnung der Busse entsprach genau den möglichen Nachrichtenwegen. Die physische Verteilung der Agenten auf die Prozessoren stimmte also mit dem logischen Netz überein, so daß jede Nachricht über nur einen Bus laufen mußte. (Die Busse können Nachrichten in beiden Richtungen übertragen.) Es ergab sich das unten abgedruckte Ausführungsprotokoll.

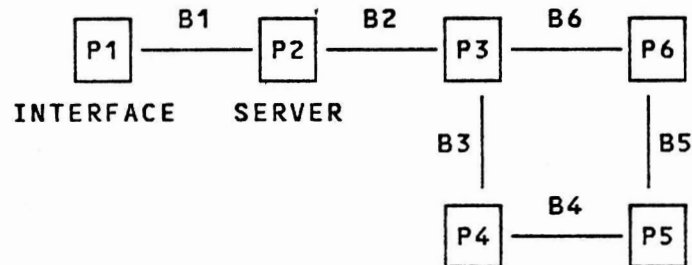
Hieraus lassen sich zunächst folgende Aussagen ableiten:

- Die Prozessoren, auf denen die ggT-Prozesse laufen, sind im Schnitt nur zu knapp über 10% ausgelastet. Der erreichte **Grad an Parallelität** ist dementsprechend nicht besonders hoch.
- Die Busse, die den Ring bilden, sind nicht überlastet. Nur etwa jede vierte Nachricht konnte nicht sofort verschickt werden (die durchschnittliche Länge der Warteschlange vor einem Bus beim Eintreffen einer Nachricht war 0.24 Nachrichten). Vom Absenden bis zum Eintreffen einer Nachricht vergingen dementsprechend im Mittel 1.24 Zeiteinheiten.

=====

C S S A - S I M U L A T I O N - S Y S T E M

=====



=====

```

>>> 0.000 INTERFACE(1) : ENTER CSSA COMMAND -
==> VAR AGENT: S:- NEW SERVER (PROCESS); OPER: GGT
==> SEND GGT(4 (* N *), 108,76, 12, 60) TO S; RUN
*** 2.900 PROCESS(1) : ***** MY CURRENT NUMBER : 108 *****
*** 5.300 PROCESS(2) : ***** MY CURRENT NUMBER : 76 *****
*** 6.900 PROCESS(4) : ***** MY CURRENT NUMBER : 60 *****
*** 7.400 PROCESS(1) : ***** MY CURRENT NUMBER : 32 *****
*** 7.700 PROCESS(3) : ***** MY CURRENT NUMBER : 12 *****
*** 8.900 PROCESS(2) : ***** MY CURRENT NUMBER : 12 *****
*** 9.300 PROCESS(4) : ***** MY CURRENT NUMBER : 12 *****
*** 10.300 PROCESS(1) : ***** MY CURRENT NUMBER : 8 *****
*** 11.800 PROCESS(2) : ***** MY CURRENT NUMBER : 4 *****
*** 11.800 PROCESS(4) : ***** MY CURRENT NUMBER : 4 *****
*** 13.200 PROCESS(1) : ***** MY CURRENT NUMBER : 4 *****
*** 13.200 PROCESS(3) : ***** MY CURRENT NUMBER : 4 *****
>>> 14.400 SYSTEM TERMINATED
==> SYSSTATUS

```

PROC.	UTIL.	AGENTS		
P1	100%	INTERFACE(1)*		
P2	7%	SERVER(1)		
P3	16%	PROCESS(1)		
P4	12%	PROCESS(2)		
P5	8%	PROCESS(3)		
P6	12%	PROCESS(4)		
BUS	UTIL.	#MSGs	AVG-QLen	MESSAGES
B1	7%	1	0.00	
B2	28%	4	1.50	
B3	63%	9	0.33	
B4	42%	6	0.17	
B5	35%	5	0.20	
B6	56%	8	0.25	

==> TERMINATE

CSSA-SESSION-STATISTICS

=====

```

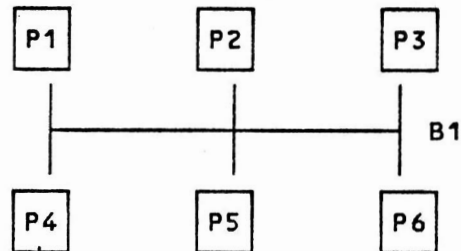
REAL-TIME USED : 52.00 SEC.
CPU-TIME USED : 1.96 SEC.
SIMULATION-TIME USED : 14.400 SEC.
NUMBER OF AGENTS CREATED: 5
NUMBER OF MESSAGES SENT : 29

```

Als nächstes wurde eine Konfiguration getestet, in der es nur einen einzigen, **zentralen Bus** gibt, der alle Prozessoren verbindet. Als Übertragungszeit wurde wiederum - wie auch in allen weiteren Beispielen - eine ZE gewählt:

=====

C S S A - S I M U L A T I O N - S Y S T E M



=====

```

>>> 0.000 INTERFACE(1) : ENTER CSSA COMMAND -
==> VAR AGENT: S:- NEW SERVER (PROCESS); OPER: GGT
==> SEND GGT(4 (* N *), 108,76, 12, 60) TO S; RUN
*** 2.900 PROCESS(1) : ***** MY CURRENT NUMBER : 108 *****
*** 3.900 PROCESS(2) : ***** MY CURRENT NUMBER : 76 *****
*** 4.900 PROCESS(3) : ***** MY CURRENT NUMBER : 12 *****
*** 5.900 PROCESS(4) : ***** MY CURRENT NUMBER : 60 *****
*** 9.000 PROCESS(1) : ***** MY CURRENT NUMBER : 32 *****
*** 11.000 PROCESS(2) : ***** MY CURRENT NUMBER : 4 *****
*** 12.000 PROCESS(4) : ***** MY CURRENT NUMBER : 12 *****
*** 17.000 PROCESS(1) : ***** MY CURRENT NUMBER : 4 *****
*** 18.000 PROCESS(3) : ***** MY CURRENT NUMBER : 4 *****
*** 21.000 PROCESS(4) : ***** MY CURRENT NUMBER : 4 *****
>>> 25.800 SYSTEM TERMINATED
>>> 25.800 INTERFACE(1) : ENTER CSSA COMMAND -
==> SYSSTATUS

```

PROC.	UTIL.	AGENTS		
P1	100%	INTERFACE(1)*		
P2	4%	SERVER(1)		
P3	7%	PROCESS(1)		
P4	4%	PROCESS(2)		
P5	4%	PROCESS(3)		
P6	7%	PROCESS(4)		
BUS	UTIL.	#MSGS	AVG-QLEN	MESSAGES
B1	97%	25	4.16	

==> TERMINATE

CSSA-SESSION-STATISTICS

=====

```

REAL-TIME USED : 58.00 SEC.
CPU-TIME USED : 1.87 SEC.
SIMULATION-TIME USED : 25.800 SEC.
NUMBER OF AGENTS CREATED: 5
NUMBER OF MESSAGES SENT : 25

```

Beim Vergleich mit der ringförmigen Konfiguration lassen sich folgende Beobachtungen machen:

- Die insgesamt verbrauchte Simulationszeit ist wesentlich höher.
- Die Prozessoren sind in geringerem Maße ausgelastet.
- Der einzige Bus ist deutlich überlastet: Im Schnitt hatte eine Nachricht beim Eintreffen in der Warteschlange mehr als 4 weitere Nachrichten vor sich. Bis zur Ankunft einer Nachricht beim Zielagenten vergingen also mehr als 5 Zeiteinheiten.
- Bedingt durch die geänderte Konfiguration änderte sich der Ablauf des verteilten Algorithmus: Es wurde nicht die gleiche Folge von Zwischenergebnissen durchlaufen und insgesamt wurden nur 25 (gegenüber 29) Nachrichten versandt. Bei gleichem Ablauf der Berechnung wäre der Unterschied zur Ring-Konfiguration vermutlich noch deutlicher ausgefallen.

Es hat sich hier deutlich gezeigt, daß es sich bei dem verteilten ggT-Algorithmus um einen **nicht-deterministischen** Algorithmus handelt: Die Wahl der hardware-Konfiguration hat den Ablauf der verteilten Berechnung beeinflußt! Im Simulationssystem ist aber gewährleistet, daß auf **ein und derselben Konfiguration** der Verlauf einer Berechnung exakt wiederholt werden kann, da die Zufallszahlengeneratoren **reproduzierbare** Zahlenfolgen liefern.

Zum Vergleich wurden auch noch zwei **weitere Konfigurationen** untersucht, bei denen größere Gesamtzeiten erzielt wurden, da Nachrichten vielfach über mehrere Stationen weitergereicht werden mußten: Es wurde der in Abschnitt 8.1 gezeigte binäre Baum aus 7 Prozessoren modelliert, sowie eine weitere ringförmige Konfiguration, bei der allerdings Nachrichten nur gegen den Uhrzeigersinn geschickt werden konnten. Eine Übersicht über alle getesteten Konfigurationen gibt die weiter unten abgedruckte **Tabelle**.

Ein Ausschnitt aus dem Protokoll zur Einfachring-Konfiguration zeigt, wie mit Hilfe der SYSTEM-option die **Aktionen des Betriebssystems** protokolliert werden können:

>>> 14.000 INTERFACE(1) : ENTER CSSA COMMAND -
 ==> SYSSTATUS

PROC.	UTIL.	AGENTS		
P1	100%	INTERFACE(1)*		
P2	7%	SERVER(1)		
P3	8%	PROCESS(1)		
P4	8%	PROCESS(2)		
P5	4%	PROCESS(3)		
P6	8%	PROCESS(4)		
BUS	UTIL.	#MSGS	AVG-QLEN	MESSAGES
B1	7%	1	0.00	
B2	29%	4	1.50	
B3	58%	9	1.00	EUKLID
B4	56%	8	0.75	EUKLID
B5	46%	7	0.71	EUKLID
B6	22%	6	1.00	EUKLID EUKLID EUKLID

```

==> OBSERVE; SYSTEM; RUN 1
/// 14.000 P1: TIMESLICE_RUNOUT
/// 14.000 P1: INTERFACE(1) SUSPENDED
/// 14.000 P1: INTERFACE(1) ACTIVATED

/// 14.100 B4: BUS IS IDLE

/// 14.100 P5: PROCESSOR IS ACTIVATED BY EXTERNAL INTERRUPT
+++ 14.100 P5: PROCESS(3) RECEIVES EUKLID(INT:12)
        FROM PROCESS(2)
/// 14.100 P5: PROCESS(3) IS ACTIVATED
+++ 14.100 P5: PROCESS(3) IS IDLE
/// 14.100 P5: PROCESSOR IS IDLE

/// 14.600 B5: BUS IS IDLE

/// 14.600 P6: PROCESSOR IS ACTIVATED BY EXTERNAL INTERRUPT
/// 14.600 P6: PASSING EUKLID(INT:12)
/// 14.600 P6: PROCESSOR IS IDLE

/// 14.900 B6: STARTING TO TRANSMIT EUKLID(INT:12)

/// 14.900 P3: PROCESSOR IS ACTIVATED BY EXTERNAL INTERRUPT
+++ 14.900 P3: PROCESS(1) RECEIVES EUKLID(INT:60)
        FROM PROCESS(4)
/// 14.900 P3: PROCESS(1) ACTIVATED

/// 14.900 B3: BUS IS IDLE

/// 14.900 P4: PROCESSOR IS ACTIVATED BY EXTERNAL INTERRUPT
/// 14.900 P4: PASSING EUKLID(INT:60)
/// 14.900 P4: PROCESSOR IS IDLE

/// 14.900 B4: STARTING TO TRANSMIT EUKLID(INT:60)

+++ 14.900 P3: PROCESS(1) IS IDLE
/// 14.900 P3: PROCESSOR IS IDLE

```

Als **Extremfall** wurde eine Konfiguration getestet, bei der alle ggT-Prozesse auf einem einzigen Prozessor (Monoprozessor) ablaufen. Die Übertragung einer Nachricht kostet hier keine Zeit, andererseits ist keine **echte** Parallelarbeit der Prozesse möglich. Das lokale multiprogramming auf dem Prozessor ermöglicht lediglich die quasi-parallele Ausführung. Es zeigte sich, daß die Berechnung des ggT wesentlich schneller durchgeführt werden konnte als bei allen bisher getesteten Konfigurationen, die ja eine echte Parallelarbeit ermöglichten.

Offenbar lohnt sich die echt parallele Ausführung erst, wenn aufwendigere Berechnungen parallelisiert werden, oder wenn die Übertragungszeiten für Nachrichten noch kleiner werden. Als weiterer Extremfall wurde daher eine Konfiguration gewählt, bei der sowohl jeder ggT-Prozeß seinen eigenen Prozessor hat, als auch die Nachrichten **ohne Zeitverlust** übertragen werden. Diese Konfiguration ist zwar nicht sehr realistisch, gibt aber dennoch einigen Aufschluß über den im Algorithmus enthaltenen (und damit maximal erreichbaren) **Grad an Parallelität**: Gegenüber dem Monoprozessor wurde eine Verbesserung um mehr als Faktor 2 erreicht. Die Auslastung der Prozessoren zeigt, daß im Schnitt zwei Agenten gleichzeitig tätig waren.

Die folgende Tabelle gibt eine Übersicht über die mit den verschiedenen Konfigurationen erzielten Ergebnisse. Neben der benötigten Simulationzeit ist die Gesamtzahl der versandten Nachrichten, sowie zu jedem Prozessor, auf dem ein ggT-Prozeß lief, die Auslastung vermerkt. Zu jedem von den ggT-Prozessen benutzten Bus ist die durchschnittliche Länge der Warteschlange angegeben. (Bei der Baum-Konfiguration konnte allerdings nicht eindeutig zwischen Bussen unterschieden werden, die nur in der Initialisierungsphase benutzt wurden, und solchen, die bei der eigentlichen Berechnung des ggT verwendet wurden.)

	ZE	MSGS	PROCESSOR-UTIL.				AVERAGE-QLNGTH			
Einfachring	26.2	29	9%	6%	4%	6%	0.75	0.47	0.50	1.18
zentraler Bus	25.8	25	7%	4%	4%	7%	4.16			
Baum	17.1	25	10%	6%	6%	10%	.54	.83	.5	.4 1.5
Doppelring	14.4	29	16%	12%	8%	12%	0.33	0.17	0.20	0.25
Monoprozessor	6.6	25	100%				—			
Ü-zeit 0	2.9	25	59%	38%	38%	59%	—			

Wie die durchgeführten Experimente zeigen, lassen sich mit dem Simulationssystem auf einfache Art und Weise Algorithmen auf unterschiedlichen Konfigurationen testen und damit auch Aussagen über den erreichbaren **Parallelisierungsgrad** gewinnen. Falls erforderlich, können sogar die Aktionen des verteilten Betriebssystems detailliert protokolliert werden und so bspw. **Engpässe** analysiert werden. Dies zeigt zusammen mit den früher geschilderten Konzepten des interaktiven debuggers und des Programm-entwicklungssystems sowie der Benutzerschnittstelle des Compilers, daß das realisierte CSSA-System als **vollwertiges Test- und Experimentiersystem** den Anwender bei Entwurf und Entwicklung von CSSA-Programmen in hohem Maße unterstützt.

9. Literaturverzeichnis und Dokumentation

Die folgende kommentierte Bibliographie enthält die Quellangaben der zitierten Literatur und die wesentlichen Veröffentlichungen, die als Grundlage für den praktischen Teil (Thematik Compilerbau) und den theoretischen Teil (Themen wie Programmiersprachen-Design, message-passing, verteilte Systeme, parallele Algorithmen etc.) dieser Arbeit dienten. Auch bekannte Veröffentlichungen, die keinen direkten Einfluß auf die Arbeit hatten, jedoch ähnliche Konzepte oder alternative Lösungen zu interessanten hier behandelten Problemen enthalten, wurden aufgenommen. Des weiteren werden Nachschlagewerke und Systemhandbücher genannt, die für die Anwendung des CSSA-Simulationssystems von Bedeutung sein könnten. Schließlich sind alle Berichte aufgeführt, auch die internen Memos, die im Rahmen des CSSA-Projektes erstellt wurden und in irgendeiner Form die vorliegende Arbeit beeinflußt haben.

Nicht aufgeführt sind dagegen Veröffentlichungen zu Themen wie Betriebssystem- und Hardwareaspekte von Multiprozessorsystemen (VLSI, Rechnernetze, Kommunikationsaspekte, verteilte Betriebssysteme, optimale Prozessortopologie etc.), Synchronisationskonzepte paralleler Prozesse bei gemeinsamem Speicher, Effizienz und Leistungsuntersuchungen bei nebenläufigen Berechnungen, Parallelisierung numerischer Algorithmen, Verifikation paralleler Berechnungsschemata etc. Diese Aspekte paralleler Systeme werden in der vorliegenden Arbeit nicht behandelt.

Bei den **Anmerkungen** zu den Veröffentlichungen handelt es sich i.a. nicht um vollständige Inhaltsangaben; oft werden nur diejenigen Aspekte genannt, die für die vorliegende Arbeit von einer gewissen Bedeutung sind.

- [AHU77] Aho, A.V. ; Ullman, J.D. : **Principles of Compiler Design**. Addison-Wesley, 1977.
Compilerbautheorie mit praktischen Hinweisen. Enthält u.a. ein Kapitel über Fehlerbehandlung in Compilern.
- [AUG79] Augenstein, B. : **Ein Interlisp-Interpreter für CSSA**. Memo SEKI-BN-79-01, Institut f. Informatik III, Universität Bonn, 1979.
Beschreibt den in LISP implementierten Interpreter für die sequentielle Vorversion von CSSA.
- [BAW80] Balkovich, E.E. ; Whitby-Strevens, C. : **On the Performance of Decentralized Software**. ACM 0-89791-019-2, 1980.
Als Beispiel wird eine Implementierung von EPL (-->[HUNT79]) gewählt. Die EPL-features (dynamisches Gründen von "actors", message passing, starre Sender/Empfänger-Synchronisation) werden vorgestellt; als Beispiel wird das Sieb des Eratosthenes angegeben (CSSA-Realisierung in Teil D der vorliegenden Arbeit). Die notwendigen Betriebssystemkernfunktionen (Nachrichtenübermittlung, Gründen und Be-

enden von actors, Scheduling) werden erwähnt.

- [BDG] Barth, W. ; Dressler, O. ; Goffarth, A. : Diplomarbeit, Institut für Informatik III, Universität Bonn (in Vorbereitung).
Beschreibt die Implementierung von CSSA auf einer Mikroprozessor-Anlage.
- [BFR77] Böhm, H.P. ; Fischer, H.L. ; Raulefs, P. : **CSSA - Language Concepts and Programming Methodology.** Proc. Symp. on AI and Programming Languages (Rochester 77), Sigplan Notices Vol 8 No 12 pp 100-109, 1977.
Übersicht über einige Konzepte, insbesondere deren Semantik, der früheren (sequentiellen) CSSA-Version. Auf Parallelität wird nicht eingegangen.
- [BMS-A] Beilken, C. ; Mattern, F. ; Spenke, M. : **Entwurf und Implementierung von CSSA -** Beschreibung der Sprache, des Compilers und eines Mehrrechnersimulationssystems. Teil A : **Konzepte.** Diplomarbeit, Institut für Informatik III, Universität Bonn, 1982.
Der vorliegende Band.
- [BMS-B] Beilken, C. ; Mattern, F. ; Spenke, M. : **Entwurf und Implementierung von CSSA -** Beschreibung der Sprache, des Compilers und eines Mehrrechnersimulationssystems. Teil B : **CSSA-Sprachbeschreibung.** Diplomarbeit, Institut für Informatik III, Universität Bonn, 1982.
Vollständige Beschreibung der vom Compiler akzeptierten CSSA-Sprache. Die Syntax ist in Form einer erweiterten BNF-Grammatik angegeben, die Semantik der einzelnen Sprachkonstrukte wird in informeller Weise spezifiziert. Enthält einen Sprachüberblick, kurze Beispiele zu allen Sprachkonstrukten und ein Schlagwortverzeichnis.
- [BMS-C] Beilken, C. ; Mattern, F. ; Spenke, M. : **Entwurf und Implementierung von CSSA -** Beschreibung der Sprache, des Compilers und eines Mehrrechnersimulationssystems. Teil C : **CSSA-Systembenutzung.** Diplomarbeit, Institut für Informatik III, Universität Bonn, 1982.
Beschreibt in ausführlicher Form den Umgang mit dem unter Siemens BS2000 laufenden CSSA-System. Dazu gehören das in Form einer Kommandoprozedur realisierte Entwicklungssystem, in das alle anderen CSSA-Komponenten integriert sind, der Compiler und das interaktive CSSA-System, bestehend aus interface-Agent, Mehrrechnersimulationssystem und interaktivem debugger.
- [BMS-D] Beilken, C. ; Mattern, F. ; Spenke, M. : **Entwurf und Implementierung von CSSA -** Beschreibung der Sprache, des Compilers und eines Mehrrechnersimulationssystems. Teil D : **CSSA-Programmbeispiele.** Diplomarbeit, Institut für Informatik III, Universität Bonn, 1982.

Enthält CSSA-Beispielprogramme und Testprogramme. Die Beispielprogramme zeigen, wie kleine Probleme mit CSSA programmiert werden können, sie enthalten jeweils eine Problembeschreibung und ein Ausführungsprotokoll. Dabei werden auch die Möglichkeiten zur Steuerung und Überwachung des Systems durch interface-Kommandos demonstriert. Die Testprogramme zu einzelnen Sprachkonstrukten zeigen Semantikfehler und Laufzeitfehler und demonstrieren die Funktion des Sprachkonstrukts.

- [BMS-E] Beilken, C. ; Mattern, F. ; Spenke, M. : Entwurf und Implementierung von CSSA - Beschreibung der Sprache, des Compilers und eines Mehrrechnersimulationssystems. Teil E (in 2 Bänden) : **Programmdokumentation**. Diplomarbeit, Institut für Informatik III, Universität Bonn, 1982.

Enthält u.a. den kommentierten Quellcode des Compilers und des Laufzeitsystems (bestehend aus interface-Agent, Mehrrechnersimulationssystem und debugger). Prozeduren, Steuerbefehle und andere Hilfssoftware zur Ausführung des CSSA-Systems ist ebenfalls aufgeführt. Zu einem CSSA-Programm ist der generierte Code abgedruckt. Auf die Portabilität des CSSA-Systems wird eingegangen.

- [BMS-MEM02] Beilken, C. ; Mattern, F. ; Spenke, M. : CSSA-Memo Nr. 2, Interner Bericht, 4. Dez. 1979.

Historisches Dokument. Beschreibt den Testcompiler und eine vorläufige Version des Simulationssystems (dies waren Vorläufer des gegenwärtigen CSSA-Systems). Enthält Ideen zu einer CSSA Parallel-Version und geht auf notwendige Unterschiede zur sequentiellen CSSA-Version ein. Enthält auch Anmerkungen zu Zugriffsrechten.

- [BMS-MEM03] Beilken, C. ; Mattern, F. ; Spenke, M. : CSSA-Memo Nr. 3, Interner Bericht, 4. Feb. 1980.

Hinweise zur Benutzung des Testsystems. Syntax der implementierten CSSA-Sprache. Programmbeispiele mit Systemprotokoll: Philosophenproblem, Sortieren durch pipelining und Binärbaum und Berechnung von Primzahlen nach der Eratosthenes-pipelining-Methode.

- [BMS-MEM04] Beilken, C. ; Mattern, F. ; Spenke, M. : CSSA-Memo Nr. 4, Interner Bericht, 25. Feb. 1980.

Die Ziele des schließlich realisierten und in der vorliegenden Arbeit beschriebenen CSSA-Systems werden entwickelt. Die Vorteile eines Simulationssystems werden ausführlich dargelegt.

- [BMS-MEM05] Beilken, C. ; Mattern, F. ; Spenke, M. : CSSA-Memo Nr. 5, Interner Bericht, 29. Feb. 1980.

Beschreibt das sogenannte "file"-Konzept, ein Sprachkonzept, das später zu dem Relationenkonzept weiterentwickelt wurde.

- [BMS-MEM06] Beilken, C. ; Mattern, F. ; Spenke, M. : **CSSA-Memo Nr. 6, Interner Bericht, 10. Apr. 1980.**
Ausführlicher Sprachvorschlag als Diskussionsgrundlage: Rahmenbedingungen, lokale Kontrollstrukturen, elementare und komplexe Datenstrukturen, Zugriffsrechte, assertions, Kommunikation, Facettierung, interface-Agent, exception-handling. Aus der ausführlichen Diskussion dieser und alternativer Sprachvorschläge [FRV80] kristallisierte sich schließlich im Sommer 1980 die gegenwärtige CSSA-Version heraus.
- [BMS-MEM07] Beilken, C. ; Mattern, F. ; Spenke, M. : **CSSA-Memo Nr. 7, Interner Bericht, 26. Okt. 1981.**
Ein Vorläufer von Teil C und D dieser Arbeit. Enthält eine Benutzungsanleitung für das CSSA-System und eine Reihe von Beispiel- und Testprogrammen, teilweise mit Ausführungsprotokollen.
- [BPS81] Bos, J.v.d. ; Plasmeijer, R. ; Stroet, J. : **Process Communication Based on Input Specifications.** ACM TOPLAS Vol 3 No 3, July 1981.
Gibt zunächst einen kurzen Überblick über die Interprozesskommunikation einiger neuerer Programmiersprachenmodelle wie CSP, DP, Smalltalk, Plits, Ada und Communication Ports. Anschließend wird auf die "input tool method" (ITM) eingegangen: Tools sind hierarchische Objekte, die Patterns eingehender Nachrichten matchen können und eine Rückgabefunktion berechnen. ITP ("input tool process") ist eine Erweiterung von ITM speziell für schwach gekoppelte Systeme, die über Nachrichten kommunizieren. Verschiedene Kommunikationsformen (Nennung des Empfangsprozesses und/oder eines tools in einem Prozess) werden vorgestellt, eine Reihe von Beispielen wird angegeben. Interessant ist die gewisse Ähnlichkeit zu CSSA-Konzepten. So entsprechen die tools in vielerlei Hinsicht den CSSA-Operationen: Send/receive sind unteilbar, tool-Aufrufe werden serialisiert etc. (-->[STR80]).
- [BRI78] Brinch Hansen, P. : **Distributed Processes: A Concurrent Programming Concept.** CACM Vol 21 No 11, pp 934-941.
"DP" ist wie das gleichzeitig entstandene "CSP"-Konzept von Hoare (-->[HOA78]) als Sprachkonzept für verteilte Prozesse sehr bekannt geworden und bildet die Grundlage für viele neuen Sprachentwürfe auf diesem Gebiet. Die autonomen Prozesse kommunizieren nur über parametrisierte Prozeduraufrufe miteinander, gemeinsame Speicherbereiche oder Variablen gibt es nicht. Die Anzahl der nebenläufigen Prozesse ist fest. Benutzt werden die von Dijkstra vorgeschlagenen "guarded commands" zur nichtdeterministischen Auswahl zwischen mehreren Statements und wait/when-Konstrukte, wobei das letztere das Warten auf das Eintreten eines von mehreren Ereignissen gestattet. Die von außen aufrufbaren

Prozeduren sind unteilbar. Das paper enthält mehrere Beispiele, u.a. einen Sortieralgorithmus und eine Lösung des Philosophenproblems sowie einen Scheduler, der das "shortest-job-next"-Problem löst: Hierzu muß der Scheduler die Identität der sich bewerbenden Prozesse kennen. Bemerkung: DP wurde vor allem für Realzeitanwendungen konzipiert.

- [BWL79] Ball, J.E. ; Williams, G.J. ; Low, J.R. : **Preliminary ZENO Language Description.** Sigplan Notices Vol 14 No 9, 1979.

Zeno ist eine Programmiersprache für verteilte parallele Prozesse; die Sprache ist in der Algol/Pascal Tradition gehalten, basiert aber i.w. auf Euklid. Die einzelnen Module kommunizieren über Nachrichten (send/receive) miteinander, die Lebensdauer dynamisch gegründeter Moduln ist unabhängig vom Sender, sie terminieren eigenverantwortlich. Interessant ist die sogen. "aset"-Datenstruktur, das ist i.w. ein Assoziativspeicher, der an die CSSA Datenstruktur "relation" erinnert. Wesentlich an Zeno sind Prädikate und assertions. Interessante sequentielle Sprachmechanismen: Benannte Blöcke und exit-Statements mit einer continue-Anweisung ähnlich wie in CSSA.

- [CHA82] Chang, E.J.H. : **Echo Algorithms : Depth Parallel Operations on General Graphs.** IEEE-SE Vol. 8 No. 4, July 1982.

Behandelt verteilte Graphenalgorithmen, die systemglobale Aktivitäten durchführen. Jeder Knoten des Graphen stellt einen autonomen Prozess dar und arbeitet asynchron, kein Knoten hat eine globale Systemsicht. Die Kommunikation erfolgt über Nachrichtenaustausch. Der Basis-Echoalgorithmus ("pure traversal") wird ausführlich behandelt (im Anhang von Teil C dieser Arbeit ist eine CSSA-Realisierung angegeben) und darauf aufbauend werden drei weitere Algorithmen beschrieben.

- [CSSA80] **Design, Semantics, and Implementation of a Programming Language for Asynchronous Multiprocessor Systems (Summary).**

Beschreibt einige Konzepte von CSSA und geht kurz auf Vergleiche mit anderen Programmiersprachen ein. Einige Ideen zur formalen Semantik werden skizziert. Die geplante Implementierung auf dem SMS3 Multiprozessorrechner wird kurz vorgestellt.

- [COOK81] Cook, R.P. : **Abstractions for Distributed Programming.** Computer Languages Vol 6 pp 131-138, 1981. Beschrieben wird die Programmiersprache StarMod für verteilte parallele Prozesse, die über Nachrichten kommunizieren. In einem deklarativen "network module" können globale Größen vereinbart werden und die Prozessortopologie spezifiziert werden. Ports stellen Warteschlangen für Nachrichten dar. Interessant ist das "region"-Konstrukt, mit dem sogenannte

"access-regions" definiert werden können: Dies sind Programmblöcke, in denen auf das Eintreffen einer Nachricht für eine spezifizierte Menge von ports gewartet wird. Die access-regions entsprechen somit den CSSA-Facetten. Als Beispiel ist eine Lösung des readers/writers Problems angegeben: Durch eine globale Schleife und expliziten Zustandsfallunterscheidungen wird hier mittels access-regions das Facettenkonzept "nachempfunden".

- [DEL79] DeRemer, F. ; Levy, Ph. (Ed) : Summary of the Characteristics of Several "Modern" Programming Languages. Sigplan Notices Vol 14 No 5, 1979.
Neun Sprachen werden behandelt: Alphard, Aspen, CLU, Euclid, Gypsy, Modula, Pascal, Simula, Smalltalk. Enthält neben einem tabellarischen Überblick und Vergleich zu jeder Sprache Angaben zum Zweck und Zielaspekt der Sprache, zu den wesentlichen Sprachkonzepten und zu den Möglichkeiten der Daten- und Kontrollabstraktion. Eine Lösung für ein Beispielproblem (Symboltabellenverwaltung durch hashing) wird in allen Programmiersprachen angegeben.
- [DMN71] Dahl, O.-J. ; Myhrhaug, B. ; Nygaard, K. : Simula 67 Common Base Language. Norwegian Computing Center Oslo, Publication No. S-22, 1971.
Definiert die Simula-Sprache. In Simula sind der Compiler und das Laufzeitsystem geschrieben; die Sprachkonzepte eigneten sich in hervorragender Weise zur Programmierung des Mehrrechnersimulationssystems. (-->[NCC75])
- [DOD77] Department of Defense: Revised "Ironman". Sigplan Notices, Dec. 1977.
Hier werden die generalstabsmäßig geplanten Anforderungen spezifiziert, die eine Programmiersprache für das DOD erfüllen muß. Viele dieser Anforderungen sind "allgemeingültig" und entsprechen dem "state of the art" im Programmiersprachen-Design, so daß es interessant ist zu untersuchen, inwieweit CSSA diesen Anforderungen genügt. Das Ergebnis der Bemühungen stellt übrigens die Programmiersprache Ada dar.
- [DOD79] Rationale for the Design of the Ada Programming Language. Sigplan Notices, Vol. 14 No 6 (June 1979) part B.
Enthält Begründungen für Designentscheidungen beim Sprachentwurf; erwähnt Vor- und Nachteile alternativer Lösungen, teilweise auch im Vergleich mit anderen Programmiersprachen.
- [DOD80] Reference Manual for the Ada Programming Language. US DOD-Report, July 1980.
Ada entstand etwa gleichzeitig wie CSSA. Die Zielsetzungen sind etwas unterschiedlich, dennoch lassen sich viele Ähnlichkeiten und Gemeinsamkeiten, gerade auch bei Details der sequentiellen Sprachmechanismen, feststellen.

- [FEL79] Feldman, J.A. : **High Level Programming for Distributed Computing.** CACM Vol 22 No 6, June 1979. Beschrieben wird die inzwischen sehr bekannt gewordene Programmiersprache "PLITS", die ebenso wie CSSA und eine Reihe weiterer neuerer Programmiersprachen auf dem "module-message"-Paradigma für verteilte Berechnungen beruht. Viele grundsätzliche Probleme werden ausführlich diskutiert (wie z.B. das eindeutige Kennzeichnen von Nachrichten), weitere Überlegungen zu assertions und exception-Behandlung werden dargestellt. PLITS ist als Erweiterung von Pascal implementiert, Module und Nachrichten stellen Typen dar, jedes Modul ist ein Typdefinitionsobjekt, aus dem dynamisch Instanzen gegründet werden können, die ausschließlich über Nachrichten kommunizieren und nur eigenständig terminieren. Bemerkungen auch zur Implementierung (Betriebssystemkern). Interessant wäre ein Vergleich mit den CSSA-Konzepten.
- [FI77] Fischer, H.L. : **Konzept und definierender Interpretierer für CSSA.** Diplomarbeit, Memo SEKI-77-02, Fakultät für Informatik, Universität Karlsruhe 1977. Behandelt wird die sequentielle Vorversion von CSSA.
- [FIR79] Fischer, H.L. ; Raulefs, P. : **Design Rationale for the Interactive Programming Language CSSA for Asynchronous Multiprocessor Systems.** Memo SEKI-BN-79-09, Institut f. Informatik III, Universität Bonn, Nov. 1979. Beschreibt einige Spracheigenschaften und Eigenschaften des CSSA-Modells. Die wesentlichen Konzepte werden kurz mit denen anderer Programmiersprachen für verteilte Berechnungen verglichen. CSSA-Syntax in einer vorläufigen ad-hoc Version.
- [FKPRZ80] Fischer, H.L. ; Kleinen, G. ; Pein, W. ; Raulefs, P. ; Ziegler, P. : **Die interaktive Benutzerschnittstelle von CSSA.** Notizen zum interaktiven Programmieren, pp 36-65, März 1980. Beschreibt die Benutzerschnittstelle der sequentiellen Vorversion von CSSA. Dazu gehören der Kommando-Agent (mit tracing-Möglichkeit, breakpoint Verwaltung, Zustandsinformation, Protokollierung etc.), die interaktive Fehlerbehandlung und der interaktive symbolische Interpretierer.
- [FRV80] Fischer, H.L. ; Raulefs, P. ; Voss, H. : **CSSA Working Paper I/80,** Interner Bericht, Mai 1980. CSSA-Sprachvorschläge. Diente zusammen mit [BMS-MEM06] als Diskussionsgrundlage, aus der schließlich die "endgültige", im Simulationssystem implementierte, CSSA-Sprachversion hervorging.
- [FRV81] Fischer, H.L. ; Raulefs, P. ; Voss, H. : **CSSA-Projekt,** Arbeitsbericht April 1979-März 1981. Memo SEKI-BN-81-02, Institut f. Informatik III, Universität Bonn, 1981. Beschreibt die Entwicklung der Programmiersprache

CSSA in den Jahren 1979 und 1980. Enthält eine Übersicht über die CSSA-Sprachkonzepte und eine Diskussion dieser Konzepte im Vergleich zu anderen Programmiersprachen. Die geplante Implementierung wird vorgestellt und es wird kurz auf das mathematische Berechnungsmodell eingegangen. Die beschriebene Syntax und Semantik entspricht weitgehend dem realisierten Übersetzer und Interpreter des CSSA-Simulationssystems.

- [GEN81] Gentleman, W.M. : **Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept**. Software Practice and Experience Vol 11 pp 435-466, 1981.

Behandelt folgende Probleme: Soll send/receive blockierend oder asynchron funktionieren? Wie werden Prozesse identifiziert? Werden sowohl Sender als auch Empfänger benannt (wie bspw. in CSP), ist "broadcasting send" sinnvoll oder ist die many-to-one Semantik (DP, Ada) besser? Wie können dynamisch gegründete Prozesse adressiert werden? Wie soll eine Nachricht aufgebaut sein? Soll sie eine feste oder variable Länge haben? Soll die syntaktische Struktur ähnlich wie bei Prozeduren bzw. Prozeduraufrufen sein? Wie soll auf Kommunikationsfehler reagiert werden? Vorgeschlagen wird ein reply-Statement, bei dem eine Rückgabe einer Antwortnachricht möglich ist, ohne daß der reply-ausführende Prozess den Auftraggeber explizit kennt; ein derartiges Konstrukt ist in CSSA realisiert.

- [GIL81] Giloi, W. : **Rechnerarchitektur**. Springer-Verlag HTB208, 1981.

Enthält eine Reihe wesentlicher Bemerkungen zu den Konzepten der Parallelarbeit, der wichtigsten organisatorischen Maßnahme zur Leistungssteigerung in einer Rechnerarchitektur. Multiprozessorarchitekturen werden ausführlich behandelt, insbesondere wird auch auf verteilte Systeme eingegangen.

- [GLÜ79] Glücker, R. : **Ein Interlisp-Übersetzer für CSSA**. Memo SEKI-BN-79-02, Institut f. Informatik III, Universität Bonn, 1979.

Beschreibt einen in LISP implementierten Übersetzer für die sequentielle Vorversion von CSSA. Der Übersetzer benutzt einen automatisch erzeugten LALR(1)-Parser und erzeugt interaktiv Codestücke für eine virtuelle Maschine, die den CSSA-Interpreter bildet (—> [AUG79]). Enthält auch die Grammatik der sequentiellen Version.

- [GRI71] Gries, D. : **Compiler Construction for Digital Computers**. J.Wiley, 1971.

Behandelt u.a. die Funktion von Scanner und Parser, die Methode des rekursiven Abstiegs, Symboltabellenorganisation und Fehlerbehandlungsverfahren.

- [HAR81] Harland, D.M. : **Concurrency in a Language Employing Messages.** Information Processing Letters Vol 12 No 2, 13 April 1981.
Kurzes Plädoyer für die meisten Konzepte, die in CSSA bereits realisiert sind, wie bspw. asynchrones und synchrones send/receive, mailbox ("message pool"), interface ("predeclared instance value 'world'"), dynamisches Gründen von Instanzen, Verschickbarkeit von Instanzen selbst etc. "...is considered to be quite new".
- [HEW77] Hewitt, C. : **Viewing Control Structures as Patterns of Passing Messages.** Artificial Intelligence, Vol 8 pp 323-364, 1977.
Hier wird das Actor-Modell vorgestellt, aus dem CSSA hervorging: Ein Actor soll nur durch sein Verhalten beschrieben werden (festgelegt durch ein "script"), nicht durch seine physische Darstellung. Einen globalen Zustand gibt es nicht. Ein Actor besitzt direkte Bekanntschaften ("acquaintances") auf endlich viele andere Actors. Kontrollstrukturen werden als Systeme von Nachrichten angesehen, die zwischen den Actors ausgetauscht werden. Mit dem Actor-Modell soll eine modulare Verteilung von Wissen in Expertensystemen einfach modelliert werden können.
- [HII77] Hikita, T. ; Ishihata, K. : **An Extended Pascal and its Implementation Using a Trunk.** Rep. Compt. Centre Univ. Tokyo, 1977.
Beschrieben wird die Implementierung des Pascal 8000 Compilers. Für uns war die Beschreibung einer Spracherweiterung, die sogen. "loop-Schleife", interessant: Es handelt sich um eine ereignisgesteuerte Schleife, die durch exit-Statements verlassen werden kann ("Zahn-Konstrukt"). Als Schleifenepilog kann nach dem Abbruch der Schleife ein spezifizierter Block von Statements ausgeführt werden.
- [HOA72] Hoare, C.A. : **Towards a Theory of Parallel Programming.** In: D. Gries (Ed.) : Programming Methodology, Springer-Verlag 1978. Ursprünglich in: Hoare, Perrott (Eds.): Operating Systems Techniques. Academic Press, 1972.
Enthält u.a. einen Abschnitt über kooperierende Prozesse, in dem die Lösung des Philosophenproblems diskutiert wird, bei der ein Philosoph "über Kreuz" ißt.
- [HOA74] Hoare, C.A. : **Monitors: An Operating System Structuring Concept.** CACM 17, pp 549-557, 1974.
Hier wird das Konzept des Monitors eingeführt, das später von Hansen für Concurrent Pascal verwendet wurde. Monitore wurden zunächst für die Verwaltung von Betriebsmitteln konzipiert, da sie jedoch Programmodule mit exklusiv aufrufbaren Prozessen darstellen, die ausschließlich den Zugriff auf lokale Daten erlauben, drängt sich ein Vergleich mit Agenten bei CSSA auf.

- [H0A78] Hoare, C.A.R. : **Communicating Sequential Processes**. CACM Vol 21 No 8, pp 666-677.
Dieses grundlegende paper, in dem "CSP" als abstrakte Programmiersprache vorgestellt wird, ist sehr bekannt geworden. Input, output und Parallelität werden als Programmierprimitive angesehen: Sequentielle Prozesse kommunizieren miteinander durch Nachrichtenaustausch ("I/O-commands"); die Nachrichten werden nicht gepuffert (Sender und Empfänger synchronisieren sich), für ankommende Nachrichten ist Patternmatching vorgesehen. Die sogen. "guarded commands" von Dijkstra werden benutzt. Die Anzahl der Prozesse ist fest, ein dynamisches Gründen ist nicht vorgesehen. Das paper enthält mehrere Beispiele, u.a. die Eratosthenes-Siebmethode und das Philosophenproblem sowie eine ausführliche Diskussion der vorgeschlagenen Konzepte.
- [H0R74] Horning, J.J : **What the Compiler Should Tell the User**. In: Bauer, F.L. ; Eickel, J. (Eds.) : Compiler Construction. Springer-Verlag, 1974.
Gibt u.a. Hinweise darauf, wie das Compiler-listing zweckmäßigerweise auszusehen hat, was die cross-reference Liste enthalten soll, was in Compiler-Statistiken gemeldet werden soll und wie Fehlermeldungen aussehen sollten.
- [HUNT79] Hunt, J.G. : **Messages in Typed Languages**. Sigplan Notices Vol 14 No 1, 1979.
Behandelt Probleme der Kommunikation: Compilezeitüberprüfbarkeit der Typkompatibilität übertragener Nachrichten, port-Konzepte und logische Kommunikationskanäle, Benennung von Prozessen (statisch, dynamische Prozesse) und die Problematik mehrerer Sender/mehrerer Empfänger. Vergleich mit anderen Konzepten (u.a. EPL und actors von Hewitt) bezüglich Typisierung, Benennung von Sender und Empfänger, Pufferung der Nachrichten und Statik/Dynamik der Kommunikationskanäle.
- [JEW75] Jensen, K. ; Wirth, N. : **Pascal User Manual and Report**, Springer-Verlag, 1975.
Pascal ist wegen seiner Einfachheit zu einem gewissen Standard geworden. Die Sprachkonzepte von Pascal sind oft diskutiert und kritisiert worden, viele neueren Sprachen bauen auf Pascal auf oder lehnen sich an Pascal an. Beim Design einer "Algol-artigen" Programmiersprache kommt man um Pascal nicht herum.
- [KIS79] Kieburtz, R.B. ; Silberschatz, A. : **Comments on "Communicating Sequential Processes"**. TOPLAS Vol 1 No 2, Oct 1979.
Bemerkungen zu [H0A78]. Kritisiert vor allem die starre Sender/Empfänger-Synchronisation von CSP. Vorgeschlagen wird die Verwendung von ports und die Pufferung von Nachrichten. Kritisiert wird u.a. auch noch, daß der handshake-Kommunikationsmechanismus bei der Implementierung zusätzliche Kommunikation

(request/confirmation-Signale) zwischen den Prozessen erfordert.

[KNU74]

Knuth, D.E. : **Structured Programming with goto Statements**. Computing Surveys Vol 6 No 4, 1974.

CSSA ist eine goto-freie Programmiersprache. Daher war es notwendig zu erfahren, wo und warum goto-Statements verwendet wurden. Besonders interessant sind die hier diskutierten ereignisgesteuerten Schleifen, die in abgewandelter Form für CSSA übernommen wurden. Dazu gehören auch Statementblöcke, die ausgeführt werden, nachdem ein Ereignis eingetreten ist und die Schleife verlassen wurde. Diskutiert werden weiterhin verschiedene andere Möglichkeiten, Schleifen zu verlassen.

[KON79]

Konrad, W. : **Communication and Testing in a Loosely Coupled Multi Microcomputer System**. Interner Bericht 14/79, Fachbereich Informatik der Universität Kaiserslautern, 1979.

Behandelt kurz die Kommunikationsphilosophie des ESRA-Projektes (asynchroner FIFO-gepufferter Nachrichtenaustausch durch send/receive ohne "mailbox"-Puffer) und das Problem der Testmöglichkeiten in verteilten Systemen (globaler Systemzustand, zentraler Supervisorknoten, maintenance-Bus etc.). Insofern interessant, als im CSSA-Simulationssystem die vorgeschlagenen und weitere Testmöglichkeiten vorhanden sind!

[KOV78]

Kovats, T.A. : **Program Readability, Closing Keywords and Prefix-Style Intermediate Keywords**. Sigplan Notices Vol 13 No 11, 1978.

Beschäftigt sich mit dem Syntax-Layout von Programmiersprachen. Nennt die Wirth'sche Hypothese "Was für den Compiler schwierig zu erkennen ist, ist es auch für den menschlichen Leser." Behauptet, daß schließende Schlüsselwörter nicht nur zur Visualisierung der Blockstruktur sinnvoll seien, sondern auch eine bei Algol und Pascal übliche Fehlerquelle ausschließen: Wird bei Programmen dieser Sprachen ein Statement eingefügt, so ist oft eine begin/end-Klammerung notwendig, die leicht vergessen wird. Behauptet, daß begin/end häßlich und überflüssig sei. Kritisiert Modula, wo ununterscheidbare schließende Schlüsselwörter verwendet werden ("end"). Diskutiert ferner "intermediate keywords" wie bspw. das "then". COMMENT: In CSSA existiert für jedes "öffnende" Schlüsselwort ein spezielles schließendes Schlüsselwort, u.a. wegen der hier genannten Gründe. Um mit Knuth zu sprechen: TNEMMOC.

[KRS78]

Kramer, H. ; Schübel, T. : **Die Programmiersprache CSSA**. Seminar Daten- und Kontrollabstraktionen, Institut f. Informatik III, Universität Bonn, SS 1978.

Behandelt wird die sequentielle Vorversion von CSSA. CSSA-Agenten werden mit Simula-Klassen verglichen.

- [LDHH78] Lewi, J. ; De Vlaminck, K. ; Huens, J. ; Huybrechts, J. : **The ELL(1) Parser Generator and the Error Recovery Mechanism.** Acta Informatica, pp 209-228, 1978.
Enthält die Definition einer erweiterten LL(1)-Syntax mit Angaben, wie die syntaktischen Ausdrücke in systematischer Weise in ParserROUTINEN übersetzt werden können. Das vorgestellte Fehlerbehandlungsverfahren bedient sich sogenannter "Synchronisationstripel", mit denen im Fehlerfall beim Erkennen einer schließenden syntaktischen "Klammer" (wie bspw. 'end') wiederaufgesetzt werden kann, oder aber bei Erkennen einer öffnenden syntaktischen Klammer die ganze Einheit losgelöst vom Rest des Programms analysiert werden kann.
- [LIA81] Lint, B. ; Agerwala, T. : **Communication Issues in the Design and Analysis of Parallel Algorithms.** IEEE-SE, pp 174-178 March 1981.
- [LIS82] Liskov, B. : **On Linguistic Support for Distributed Programs.** IEEE-SE Vol 8 No 3, May 1982.
Bemerkungen zu Programmiersprachen für verteilte Berechnungen, hier am Beispiel einer Erweiterung von CLU. Die Module werden "guardians" genannt, ein guardian enthält einen oder mehrere "Prozesse" und Datenobjekte mit gemeinsamem Adressraum. Ein guardian residiert ganz auf einem physischen Knoten. Die Anzahl der guardians ist dynamisch, mehrere Prozesse eines guardians können parallel ausgeführt werden. Guardians kommunizieren nur über Nachrichten miteinander, in Nachrichten werden Werte (entsprechend einem "call by value" bei Prozeduren) übertragen, auch Werte abstrakter selbstdefinierter Datentypen. Die Typkonsistenz bei der Kommunikation soll compilezeitentscheidbar sein. Besondere Beachtung findet das Problem der Sicherheit und Systemkonsistenz bei Hardwarefehlern; diskutiert werden atomare save-Aktionen.
- [MAR81] Massar, R. ; Racke, W.F. : **LADY - Eine Implementierungssprache für verteilte Betriebssysteme.** In: J. Nehmer (Hg.): Bericht 7 des German Chapter of the ACM, Feb. 1981.
Zwei Aspekte sind bei Systemkonzepten für verteilte Systeme wesentlich: Die Modularisierung und die Kommunikation. Die Modularisierung wird in LADY auf oberster Ebene durch sogen. Instanzen realisiert, das sind autonome, auf einem Knoten des Rechnersystems ablauffähige Module, die aus einer Menge von Prozessen und Monitoren bestehen, die wiederum in Prozeduren und Klassen strukturiert sind. Die Kommunikation innerhalb der Instanzen geschieht über Monitore, die Kommunikation zwischen Instanzen ausschließlich durch Nachrichten über ports. Ports selbst sind verschickbar. Das Übertragungskonzept basiert auf der Rendezvous-Technik, so daß keine Zwischenspeicherung der Nachrichten im Kommunika-

tionssystem notwendig ist. Weitere Aspekte: Strenges Typkonzept, getrennte Übersetzbarkeit, Beschreibung in drei Ebenen (Systemebene, Instanzenebene, Modulebene).

- [MAY80] Mao, T.W. ; Yeh, R.T. : Communication Port: A Language Concept for Concurrent Programming. IEEE-SE Vol 6 No 3, 1980.

CP ist ein Sprachkonzept für parallel arbeitende sequentielle Prozesse in verteilten Systemen, die über Nachrichten miteinander kommunizieren. Wesentlich sind drei Statements: Das port-Statement, connect-Statement und disconnect-Statement. Das port-Statement gibt einer Statementfolge einen Namen und erlaubt die Spezifikation von in-Parametern. Beim connect-Statement wird ein Prozessname, ein port-Name und eine Parameterliste angegeben. Läuft ein Prozess auf ein connect-Statement, so wird er so lange unterbrochen, bis der Zielprozess ein entsprechendes port-Statement ausführt. Der "Empfänger" kann den "Sender" mit dem disconnect-Statement wieder weiterlaufen lassen und dabei out-Parameter übergeben. Die Prozessidentifikation des Senders wird automatisch mitübermittelt. Die Auswahl zwischen mehreren ports kann nichtdeterministisch erfolgen, außerdem ist es möglich, einen port mit einer Booleschen Bedingung zu verknüpfen, die erfüllt sein muß. Dadurch, daß das disconnect-Statement am Anfang, in der Mitte oder am Ende eines ports stehen kann, ist die Simulation verschiedener anderer Kommunikationsmechanismen möglich. Das paper enthält einige Beispiele, u.a. den "Highest Priority First Scheduler", der öfters als ein Maß dafür genommen wird, inwieweit sich die Kommunikationsmechanismen eines Sprachkonzeptes für verteilte Systeme zum eleganten Programmieren eignen.

- [MCK74a] McKeeman, W.M. : Symbol Table Access. In: Bauer, F.L. ; Eickel, J. (Eds.) : Compiler Construction. Springer-Verlag, 1974.

Behandelt verschiedene Methoden der Symboltabellenorganisation und vergleicht diese bezüglich Effizienz. Eine Kombination der hier vorgestellten Methoden wurde im vorliegenden CSSA-Compiler verwendet.

- [MCK74b] McKeeman, W.M. : Programming Language Design. In: Bauer, F.L. ; Eickel, J. (Eds.) : Compiler Construction. Springer-Verlag, 1974.

McKeeman plaudert über mögliche "features" von Programmiersprachen und stellt u.a. die Algol-Sprachfamilie als Modell vor.

- [MEL81a] Mertens, E. ; Lattrell, H.-J. : Siemens BS2000 Simula Programmer's Guide, Rechenzentrum der Universität Mannheim, 3. Auflage, Feb. 1981.

Enthält implementierungsspezifische Details des Simula-Compilers und Laufzeitsystems. Insofern re-

levant, als das CSSA-Laufzeitsystem in Simula geschrieben ist und der CSSA-Compiler Simula-Programme generiert, die vom Simula-Compiler in Maschinencode übersetzt werden.

- [MEL81b] Mertens, E. ; Lattrell, H.-J. : **External Procedure Library Siemens BS2000 Simula User's Guide**, Rechenzentrum der Universität Mannheim, 1981.
Beschreibt die verfügbaren externen Simula-Routinen. Einige davon werden im CSSA-Compiler und Laufzeitsystem verwendet; sie können auch in eigenen externen CSSA-Routinen verwendet werden.

- [MOR82] Morrison, R. : **The String as a Simple Data Type**. Sigplan Notices, 1982.
Behandelt das in S-algol realisierte string-Konzept. Strings in Programmiersprachen sind für die Textbearbeitung nützlich. Sinnvolle Operationen sind: Erzeugen eines strings, Konkatenation von strings, Teilstringbildung, Suchen von Teilzeichenketten, Vergleich von strings, Zuweisung von strings. Nicht sinnvoll sei das Ändern, Hinzufügen oder Löschen einzelner Zeichen eines strings. Behandelt die Konsequenzen der Implementierung von string-Zuweisungen als Wertkopie und Referenzkopie. Falls das Ändern von Teilzeichenketten nicht ermöglicht wird, ist eine effiziente Implementierung durch Referenzzuweisung mit der gleichen Sicherheit der Wertkopie möglich. Anmerkung: Die hier vorgeschlagenen Konzepte sind, bis auf das Suchen von Teilzeichenketten, in CSSA realisiert.

- [MÜH79] Mühlenbein, H. : **TOCS - ein Programmsystem zur Simulation von Rechenanlagen**. GMD St. Augustin, Interner Bericht RZ.BI-79.208, 1979
TOCS (transaction oriented computer system simulation) ist ein in Simula geschriebenes Programmsystem, mit dem zentrale Betriebssystemalgorithmen und Hardwarearchitekturen simuliert werden können. Unter anderem wurden verschiedene Scheduler für das BS2000 von Siemens mit TOCS evaluiert. Einige Konzepte des TOCS-Systems haben das CSSA-Simulationssystem zumindest indirekt beeinflusst, da Michael Spenke an der Entwicklung von TOCS beteiligt war.

- [NAUR63] Naur, P. (Ed.) : **Revised Report on the Algorithmic Language ALGOL 60**. Numerische Mathematik 4, 420-453, 1963.
Algol ist als Urahne heutiger (und künftiger) Programmiersprachen noch immer von einer gewissen Bedeutung. Die klare und prägnante Spezifikation war (ist) beispielhaft; einige Konzepte (z.B. das Blockkonzept) haben sich bis heute gehalten. Gleichzeitig zeigt die Lektüre des Reports, welche Fortschritte der Entwurf von Programmiersprachen in den letzten 20 Jahren gemacht hat.

- [NCC75] **Simula IBM /360 Users Guide.** Norwegian Computing Center Oslo, Revised Edition 1975.
Syntaktische und semantische Definition von Simula, derjenigen Sprache, in der Compiler, Laufzeitsystem und Simulationssystem geschrieben sind. (-->[DMN71])
- [NYD78] Nygaard, K. ; Dahl, O.J. : **The Development of the Simula Languages.** Sigplan Notices Vol. 13 No. 8, 1978.
- [PED82] Pemberton, S. ; Daniels, M.C. : **Pascal Implementation, the P4 Compiler.** Ellis Horwood Ltd., 1982. (2 Bände).
Enthält das listing des weit verbreiteten portablen Pascal P4 Compilers und des P-Code Interpreters von Ammann, Nori, Jacobi, Jensen und Wirth (ETH Zürich). Der Compiler wird von Pemberton und Daniels beschrieben und kommentiert. Insofern interessant, als CSSA in seinen sequentiellen Teilen eine "Pascal-ähnliche" Sprache ist und der hier beschriebene Compiler ebenfalls nach der Methode des rekursiven Abstiegs funktioniert und Zwischencode für eine stack-Maschine erzeugt. In einigen Details sehr ähnlich dem CSSA-Compiler, andere Details sind unterschiedlich gelöst - ein Vergleich ist interessant !
- [PEIN79] Pein, W. : **Interaktive Fehlerbehandlung in CSSA.** Memo SEKI-BN-79-03, Institut f. Informatik III, Universität Bonn, 1979.
Beschreibt die auf [AUG79] und [GLÜ79] aufbauende Fehlerbehandlung der sequentiellen Vorversion von CSSA.
- [PIE77] Pieper, F. : **Einführung in die Programmierung paralleler Prozesse.** Oldenbourg Verlag, 1977.
Didaktisch orientiert, auf Betriebssystem-Theorie zugeschnitten. Enthält eine ausführliche Diskussion des Philosophenproblems.
- [RAU80] Raulefs, P. : **Actors und Agenten.** Notizen aus der Vorlesung "Semantik, Konstruktion und Analyse von Programmen für kooperierende parallele Prozesse", Universität Bonn, SS 1980.
Die wesentlichen Konzepte des Actor-Modells von Hewitt und des Agenten-Modells (aus dem die Programmiersprache CSSA entwickelt wurde) werden kurz vorgestellt.
- [SCH80] Schwartz, J.T. : **Ultracomputers.** ACM-TOPLAS, October 1980.
- [SCHW82] Schüler, G. ; Wybranietz, D. : **Entwurf und Teilrealisierung eines Betriebssystems auf einer Multicomputer-Architektur,** Diplomarbeit, Institut f. Informatik III, Universität Bonn, 1982.

- [SIEMENS] **Diverse Handbücher zum Siemens BS2000 Betriebssystem.**
In das CSSA-System integriert ist u.a. der an der GMD entwickelte und von Siemens vertriebene EDOR Programmeditor; das Entwicklungssystem ist als Prozedur von BS2000 Kommandos realisiert. In den Siemens Handbüchern finden sich daher u.U. für die Benutzer des CSSA-Systems notwendige oder nützliche Hinweise.
- [SIL81a] **Silberschatz, A. : On the Synchronization Mechanism of the Ada Language.** Sigplan Notices, Feb. 1981.
Zunächst wird kurz das Ada handshaking-Konzept zur Kommunikation und Synchronisation bei tasks vorgestellt (accept-Statement und when-Statement, wobei letzteres die nichtdeterministische Auswahl zwischen mehreren "rufenden" tasks gestattet). Es wird dann behauptet, daß diese Konstrukte nicht mächtig genug seien, um viele Synchronisationsprobleme in eleganter Weise zu lösen; als Beispiel wird das Prioritäts-Scheduling genannt. Vorgeschlagen wird ein "await-Statement" mit der Möglichkeit, Prioritäten zu spezifizieren; angeblich kann damit das "shortest-job-next"-Problem einfach und elegant gelöst werden.
- [SIL81b] **Silberschatz, A. : A Note On The Distributed Program Component Cell.** Sigplan Notices, Aug. 1981.
Schon wieder ein Konzept für verteilte Berechnungen durch sequentielle Moduln, hier "cell" genannt, die durch send/receive miteinander kommunizieren. Nicht viel Neues: Ungepufferte Nachrichtenübertragung, handshake-Kommunikation mit impliziter Synchronisation durch "accept" wie in Ada, selektive Auswahl von Nachrichten durch "when". Aus einer cell können mehrere Instanzen generiert werden, die dann ein hierarchisches System bilden. "Neu" sind lediglich Sprachkonzepte, die das Scheduling unterstützen ("await" und Prioritätsangaben).
- [SPE81] **Spenke, M. : Message Passing Konzepte.** Seminar "Neuere Arbeiten über Synchronisation paralleler Prozesse", Institut für Informatik III, Universität Bonn, SS 1981.
Enthält zunächst einige allgemeine Bemerkungen über Kommunikationsformen durch Nachrichtenaustausch; anschließend werden die Programmiersprachen PLITS, CSP, CSSA, DP und Ada anhand eines konkreten Beispiels (Philosophenproblem) verglichen.
- [SPE82] **Spenke, M. : Ein sprachunabhängiges Verfahren zur Fehlerbehandlung für LL(1)-Sprachen.** WASC0-Tagung, Budapest, 1982.
Darstellung der im CSSA-Compiler zur Anwendung kommenden Fehlerbehandlungsmethode. Abgedruckt im Anhang des vorliegenden Bandes.

- [STR80] Stroet, J. : **An Alternative to the Communication Primitives in Ada.** Sigplan Notices, Dec. 1980.
Vergleicht ITP (-->[BPS81]) mit Ada an vielen kleinen Beispielen. ITP erlaubt asynchrones send/receive; ein ITP-Prozess enthält mehrere "tools", die nach dem matchen einer ankommenden Nachricht ausgeführt werden, die Serialisierung ist implizit gewährleistet.
- [VOSS80] Voss, H. : **Beschreibung der Mailbox-Verwaltung als Teil des Betriebssystemkerns für eine CSSA-Implementierung.** Memo SEKI-BN-80-08, Institut für Informatik III, Universität Bonn, 1980.
Detaillierte Beschreibung einiger Algorithmen, die bei der Implementierung von CSSA für eine faire Auswahl der Nachrichten aus der mailbox sorgen, unter Berücksichtigung von Randbedingungen wie mailbox-overflow, asynchrones Eintreffen von Nachrichten und Akzeptierbarkeit von Nachrichten.
- [WEI78] Weicker, R. : **Neuere Konzepte und Entwürfe bei Programmiersprachen.** Informatik Spektrum Vol. 1 pp 101-112, 1978.
Behandelt werden einige Konzepte neuerer Programmiersprachen wie goto-Problematik, ereignisgesteuerte Schleifen, Lesbarkeit von Programmiersprachen, vollständige Typbindung, abstrakte Datentypen, generische Prozeduren, information-hiding, Verifikation von Programmen. Die Programmiersprachen Pascal, LIS, Euklid, CLU, Alphard, Concurrent Pascal, Modula und Ada werden kurz vorgestellt.
- [WEL81] Welsh, J. ; Lister, A. : **A Comparative Study of Task Communication in Ada.** Software Practice and Experience Vol 11 pp 257-290, 1981.
Diskussion der Ada-Sprachkonzepte zur Kommunikation im Vergleich zu CSP von Hoare und DP von Hansen. Auf folgende Problematiken wird eingegangen: Task-Identifikation (Empfängertask soll Sendertask erkennen und identifizieren), Synchronisation, nicht-deterministische Kontrollstrukturen, busy-waiting, selective waiting (Warten auf das Eintreffen einer Teilmenge von möglichen Nachrichten). Ein Anhang enthält 16 Ada-Programmbeispiele, u.a. das Philosophenproblem und die Eratosthenes-Siebmethode zur Primzahlberechnung, die in einem früheren paper der Autoren bereits in CSP und DP programmiert wurden. Die in diesem paper diskutierten Aspekte sind interessant, auch CSSA sollte unter diesen Gesichtspunkten evaluiert und verglichen werden.
- [WHI76] White, J.R. : **Towards a New Methodology for Translator Development.** In: Proc. of the 4th Int. Conf. on the Design and Implementation of Algorithmic Languages (Ed.: Dewar, R.B.K.), June 1976.
Geht zunächst auf die übliche Strukturierung eines Compilers in lexikalische, syntaktische und semantische Analyse mit Codegenerierung ein. Stellt fest,

daß für die ersten beiden Phasen Generatoren existieren, so daß die semantische Analyse der kritische Punkt sei. Erwähnt dann kurz die Methode der top-down Programmentwicklung durch schrittweise Verfeinerung mittels Daten- und Kontrollabstraktion. Diese Methode soll auf die Programmierung der semantischen Analyse eines Compilers angewendet werden, um die Komplexität zu meistern und einen verlässlichen Compiler zu erhalten. Vorgeschlagen wird die Verwendung einer Folge von (virtuellen) Zwischensprachen, die nicht in mehreren Läufen schließlich die Zielsprache realisieren, sondern bei denen die Funktionen einer Zwischenschicht mit denen der darunterliegenden "tieferen" Schicht realisiert werden. Als Beispiel wird die Codegenerierung einer ereignisgesteuerten Schleife angegeben. Bemerkung: Im CSSA-Compiler, der in der vorliegenden Arbeit beschrieben wird, erfolgt die Codegenerierung über eine derartige Folge von virtuellen Zwischenschichten.

- [WIR77] Wirth, N. : **Compilerbau**. Teubner 1977.
Für eine einfache Sprache wird ein Parser nach der Methode des rekursiven Abstiegs entwickelt, dieser Parser wird anschließend zu einem Compiler weiterentwickelt, der Code für eine hypothetische Maschine erzeugt. Der Code wird von einem Interpreter ausgeführt. (-->[PED82])
- [WSH77] Welsh, J. ; Sneeringer, W.J. ; Hoare, C.A.R. : **Ambiguities and Insecurities in Pascal**. Software Practice and Experience Vol 7 No 6, 1977.
"If future language designs, and indeed future users, are to benefit fully from the significant advances made by Pascal, it is essential that its defects, as well as its virtues, should be carefully identified and catalogued... this criticisms may lead... to a better treatment of... problems in the languages which must inevitably follow it." Behandelt werden u.a. Typgleichheit und Typäquivalenz, explizite und implizite one-pass-Compilierbarkeit, Implementierungsrestriktionen bei Mengen (Obermenge, Größe des Basistyps), variante records und Funktionen und Prozeduren als Parameter (Typüberprüfbarkeit von Parametern).
- [WYB81] Wybranietz, D. : **Ein verteiltes Betriebssystem für CSSA**. Memo SEKI-BN-81-07, Institut f. Informatik III, Universität Bonn, 1981.
Kurzbeschreibung des in [SCHW82] beschriebenen hardwareunabhängigen Betriebssystems für eine CSSA-Implementierung auf universellen Multimicrocomputersystemen. Enthält auch einen kurzen Überblick der wesentlichen Konzepte des CSSA-Systems und der CSSA-Sprache.

- [ZIE79] Ziegler, P. : **Die interaktive Benutzerschnittstelle für das CSSA-System.** Memo SEKI-BN-79-08, Institut f. Informatik III, Universität Bonn, 1979.

- [ZIMA75] Zima, H. : **Betriebssysteme - Parallele Prozesse.** Bibliographisches Institut, 1975.
Behandelt grundsätzliche Aspekte der Parallelität, u.a. auch das Philosophenproblem als ein Beispiel.

- [ZIMA82] Zima, H. : **Compilerbau I : Analyse.** Vorabdruck, Juli 1982.
Behandelt unter besonderer Berücksichtigung von Methoden zur automatischen Generierung von Compilerkomponenten aus formalen Spezifikationen die einzelnen Phasen des Analyseteils eines Compilers, und zwar die lexikalische Analyse, Syntaxanalyse und semantische Analyse; in einem Anhang ist die semantische Analyse für Pascal beschrieben. Enthält u.a. ein Unterkapitel über Symboltabellen, in dem die verschiedenen Organisationsformen beschrieben und diskutiert werden.

- [SONST] **Handbücher zu sonstigen Programmiersprachen.**
Verschiedenen andere Programmiersprachen (wie PL/1, Fortran 77, Elan, Algol 68, Algol W, Concurrent Pascal, Modula, CLU, Gypsy), deren Sprachbeschreibungen in dieser Literaturliste nicht aufgeführt sind, wurden bei der Konzeption von CSSA untersucht.

```

. . . . .
. Each successive language incorporates with a little
. cleaning up, all the features of its predecessors plus
. a few more. ... Each new language claims new and
. fashionable features ... but the plain fact is that
. few languages make programming ... more reliable to
. justify the cost of ... learning to use them.
.   John Backus
.
. . . . .

```

Anhang

Der hier abgedruckte Aufsatz beschreibt das Prinzip der im CSSA-Compiler verwendeten Methode zur Behandlung von Syntaxfehlern. Es handelt sich um das Manuskript zu einem Vortrag, der auf der WASCO-Tagung vom 31.3-3.4.1982 in Budapest gehalten wurde.

Das für den CSSA-Compiler entwickelte Verfahren wurde auch für PASCAL implementiert, so daß ein praktischer Vergleich mit anderen Verfahren möglich war. Unter anderem wurde der PASCAL-Parser bei einem Programmierpraktikum im SS 1981 an der Uni Bonn eingesetzt und mit mehreren tausend Programmen von PASCAL-Anfängern getestet.

Obwohl das Verfahren ursprünglich für Parser nach dem rekursiven Abstieg entwickelt wurde, wird es zum besseren Verständnis anhand eines tabellengesteuerten Parsers erklärt. Mittlerweile wurde bei der GMD ein LL(1)-Parsergenerator implementiert, in den das Verfahren integriert ist.

Ein sprachunabhängiges Verfahren zur Fehlerbehandlung für LL(1)-Sprachen

(A Practical Automatic Error Recovery Method for LL(1)-Parsers)

Michael Spenke

Gesellschaft für Mathematik und Datenverarbeitung

Postfach 1240

D 5204 St. Augustin

Abstract:

An efficient and systematic LL(1) error recovery method is presented that can be used in connection with an LL(1) parser generator. Error messages with good diagnostic information are generated automatically. Symbol insertions, deletions and substitutions are simulated in order to restore the parser to a valid configuration. Among the possible corrections the one with the smallest costs is selected. Special care is taken to others than the so called "one-token" errors, typically misplaced syntactic units: they can be parsed in isolation. The recovery scheme is simple and very powerful: almost all errors are found in one scan and spurious error messages are very rare. Semantic analysis is not affected and the grammar needs not to be modified. The error-handling routines are invoked only when an error is detected and additional time for recovering from the error can be neglected. The method has been implemented with success in three recursively descent parsers for LL(1)-languages including PASCAL. It is being implemented at present in a table driven LL(1) parser generator. The PASCAL-version has been used by more than one hundred students in several thousand programs. The experimental results are summarized.

1. Einführung

Die Fehlerbehandlung gehört zu den wichtigsten Aufgaben von Compilern für höhere Programmiersprachen. Die meisten Programme, die ein Compiler analysieren und übersetzen soll, enthalten Fehler. Da moderne Übersetzer **syntax-gesteuert** arbeiten, sind gerade **Syntaxfehler** besonders kritisch. Eine gute Fehlerbehandlung ist von **großer praktischer Bedeutung**: Werden alle Fehler in einem Compilerlauf gefunden, präzise diagnostiziert und keine Folgefehler gemeldet, so kann das die Programmentwicklung wesentlich beschleunigen und damit Softwarekosten reduzieren (und gleichzeitig die Lebensqualität eines Programmierers erhöhen...).

Das Interesse an **systematischen und automatisierbaren** Fehlerbehandlungsmethoden hat in den letzten 10 Jahren stark zugenommen, vor allem bedingt durch den Automatisierungsprozess im Compilerentwurf selbst (Parsergeneratoren, Compiler-Compiler). Ein automatisch generierter Parser, in den die Fehlerbehandlung nachträglich von Hand eingebaut werden muß, wäre von geringem Nutzen!

Die meisten Papiere zu diesem Thema wurden nach 1970 veröffentlicht. Die bekannten und häufig zitierten Dissertationen von Leinius, LaFrance, Levy und Peterson (—>[9]) vom Anfang der 70er Jahre stellen einen gewissen **Meilenstein** dar, hier wird erstmals das Problem der Syntaxfehlerbehandlung in relativ systematischer Weise untersucht.

Die Situation **vor den 70er Jahren** wird am besten durch zwei Zitate belegt. Pollack schreibt in seinem Buch "Compiler Techniques": "Chapter 5 treats the most poorly documented topics in compiler writing, those of error detection and error correction. Few general techniques are currently known for these subjects, perhaps accounting for the dearth of available literature." Feldman und Gries schreiben (Translator Writing Systems, CACM 11, 1968): "There has been very little effort on the problems of automatic error detection and recovery in syntax-directed processors. Once again, even a bad system would be of great value to the users!"

In den 70er Jahren wurde dann eine Vielzahl von Fehlerbehandlungsverfahren entwickelt. Am bekanntesten ist wohl die Methode von Wirth und Amman [1], die

in den meisten PASCAL-Compilern verwendet wird. Sie ist aber nur teilweise systematisch und beinhaltet eine Reihe sprachabhängiger Heuristiken. Die Suffixanalyse von Moll [2] ist noch nicht praktisch einsetzbar und macht Überprüfungen der statischen Semantik nach dem ersten Fehler unmöglich. Fischer, Milton und Quiring [3] beschreiben ein experimentelles Verfahren, das alle Fehler durch reines Einfügen von Symbolen behebt. Das Verfahren von Pai und Kiebertz [4] ähnelt in einigen Punkten dem hier beschriebenen. Es gibt auch eine Reihe von Verfahren, die speziell für LR- und Präzedenzparser geeignet sind. Am häufigsten werden Graham und Rhodes [8] zitiert.

Ripley und Druseikis [5] haben statistisch untersucht, welche Fehler von Programmierern gemacht werden und einen Fehlerbenchmark zusammengestellt. Sie kritisieren, daß die wenigsten vorgeschlagenen Verfahren praktisch ausgetestet wurden: "Most evaluations in print consist at best of an illustration of the technique on a few examples, almost invariably leaving the reader wondering how representative these examples are, and how well the technique works in general." Aho und Ullman [6] bestätigen, daß die Beschäftigung mit dem Thema sich noch lohnt: "Although a considerable amount of theoretical and practical effort has been expended in exploring recovery and repair techniques for syntactic errors, the optimal strategy for any programming language is still an open question."

Damit ein Verfahren in der Praxis (in einem Compiler-Generator) anwendbar ist, sollten alle folgenden Punkte erfüllt sein:

- Weitgehende Sprachunabhängigkeit
- Effizienz
- Systematisierbarkeit und Automatisierbarkeit
- Automatische Generierung benutzerorientierter Fehlermeldungen. (Keine sprachabhängigen Meldungen)
- Anwendbarkeit auch in Compilern, nicht nur in Parsern. (Statische Semantik anschließbar, keine Grammatikänderungen nötig)
- Meldung aller Fehler, keine Folgefehler

Der letzte Punkt muß natürlich relativiert werden, da "Folgefehler" und "Fehlerstelle" letztlich pragmatische Begriffe sind, die sich einer vollständigen Formalisierung entziehen.

Die gängigen Analyseverfahren (LL,LR) haben die **valid-prefix property**, d.h. sie erkennen einen Fehler, sobald die gelesene Eingabe keinen Präfix der Sprache mehr darstellt. Das folgende Beispiel zeigt, daß die so definierte Fehlerstelle (**parser-defined error**) keineswegs immer mit der "tatsächlichen" Fehlerstelle übereinstimmt:

WERT:=0.3*(A+B) + 0.7*C+D);

↑

Es ist nicht entscheidbar, wo eine öffnende Klammer fehlt, oder ob die schließende Klammer zuviel ist. Was der "eigentliche" Fehler ist, kann ein Parser nicht entscheiden. Er sollte sich daher darauf beschränken, möglichst schnell wieder "Fuß zu fassen" und die Analyse des Programmes fortzusetzen, also ein **schnelles und sicheres recovery** durchzuführen. Eine Korrektur im eigentlichen Sinne kann grundsätzlich nur vom Programmierer durchgeführt werden.

Im LL(1)-Fall werden in der Regel noch weitere Anforderungen an einen effizienten, fehlerkorrigierenden Parser gestellt, die ihm "das Leben schwer machen":

- Es wird von links nach rechts gelesen
- Der Parser kann auch im Fehlerfall nur ein Zeichen voraussehen
- Der bereits gelesenen Teil der Eingabe wird nicht mehr korrigiert
- Kein backtracking, auch nicht im Fehlerfall

Der Parser muß also in dem folgenden Beispiel sofort entscheiden, ob nach dem Semikolon bereits das nächste Statement folgt, oder ob erst noch 'then' kommt.

```
if X=5 ; //////////
```

Eine falsche Entscheidung kann nach dem Weiterlesen nicht zurückgenommen werden.

2. Informelle Darstellung der Methode

Anhand einer Mini-Grammatik für arithmetische Ausdrücke soll hier gezeigt werden, welche recovery-Aktionen ein tabellengesteuerter LL(1)-Parser vornehmen kann.

Mini-Grammatik:

- 1: $\langle \text{expr} \rangle ::= \langle \text{term} \rangle [+ \langle \text{expr} \rangle]$
- 2: $[+ \langle \text{expr} \rangle] ::= + \langle \text{expr} \rangle \mid \text{LAMBDA}$
- 3: $\langle \text{term} \rangle ::= \langle \text{factor} \rangle [* \langle \text{term} \rangle]$
- 4: $[* \langle \text{term} \rangle] ::= * \langle \text{term} \rangle \mid \text{LAMBDA}$
- 5: $\langle \text{factor} \rangle ::= \text{ID}$
- 6: $\langle \text{factor} \rangle ::= (\langle \text{expr} \rangle)$

Parsing-Tabelle:

	$\langle \text{expr} \rangle$	$\langle \text{term} \rangle$	$\langle \text{factor} \rangle$	$[+ \langle \text{expr} \rangle]$	$[* \langle \text{term} \rangle]$
ID	1	3	5	LAMBDA	LAMBDA
(	1	3	6	LAMBDA	LAMBDA
+	ERROR	ERROR	ERROR	2	LAMBDA
*	ERROR	ERROR	ERROR	LAMBDA	4
)	ERROR	ERROR	ERROR	LAMBDA	LAMBDA
EOF	ERROR	ERROR	ERROR	LAMBDA	LAMBDA

Die angegebene Grammatik hat die LL(1)-Eigenschaft. Nonterminals stehen in spitzen Klammern bzw. in eckigen, falls sie direkt nach LAMBDA abgeleitet werden können. LAMBDA-Regeln haben keine eigenen Nummern, stattdessen steht LAMBDA direkt in der Tabelle.

Der tabellengesteuerte Parser arbeitet mit der Parsing-Tabelle und einem Stack für Terminal- und Nonterminalzeichen nach dem folgenden Algorithmus:

```

PUSH(EOF); PUSH(EXPRESSION); (* Initialisierung des Stack *)
NEXT_TOKEN; (* Der scanner liest das erste Eingabesymbol *)
while not STACK_EMPTY do
  begin if IS_TERMINAL(STACKTOP)
    then begin if TOKEN = STACKTOP
      then begin POP; NEXT_TOKEN end
      else SYNTAX_ERROR;
    end
    else begin RULE:=PARSING_TABLE [TOKEN,STACKTOP];
      case RULE OF
        ERROR: SYNTAX_ERROR;
        LAMBDA: POP;
        else: APPLY(RULE);
      end;
    end;
  end;
end;

(* tabellengesteuerter LL(1)-Parser *)

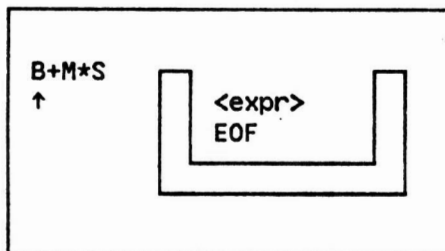
```

Ist STACKTOP ein Terminalzeichen, so wird es mit dem Eingabesymbol (TOKEN) verglichen. Bei Gleichheit wird der Stack gepoppt und das nächste Symbol gelesen. Anderfalls liegt ein Fehler vor.

Ist STACKTOP ein Nonterminal, so wird die Parsing-Aktion aus der Tabelle entnommen:

- Regelnummer ==> Anwenden der Regel auf STACKTOP, d.h. STACKTOP wird durch die rechte Seite der Regel ersetzt.
- LAMBDA ==> STACKTOP löschen durch poppen des Kellers (nach LAMBDA ableiten)
- ERROR ==> Fehlerbehandlungsroutine aufrufen

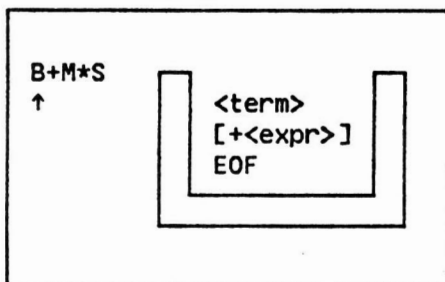
2.1 Analyse korrekter Ausdrücke



Es soll zunächst demonstriert werden, wie der (korrekte) Ausdruck $B+M*S$ analysiert wird. Links ist die Startkonfiguration gezeigt.

Mit $\uparrow$ wird jeweils das aktuelle Eingabesymbol markiert. Man beachte, daß zu jedem Zeitpunkt der Analyse der Stackinhalt aussagt, welcher Input noch erwartet wird: Alle Suffixe, die den bereits gelesenen Input zu einem korrekten Ausdruck ergänzen, sind aus dem Stackinhalt

ableitbar. Am Anfang wird eine vollständige expression gefolgt vom end-of-file Symbol erwartet.

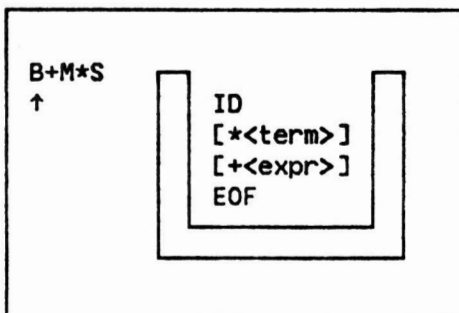


Aus der Parsing-Tabelle ergibt sich, daß zunächst Regel 1 anzuwenden ist:

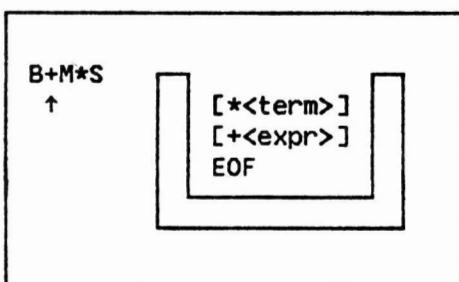
$\text{PARSING-TABLE}(\text{ID}, \langle \text{expr} \rangle) = 1$.

Das Ergebnis dieser Regelanwendung zeigt die nebenstehende Abbildung.

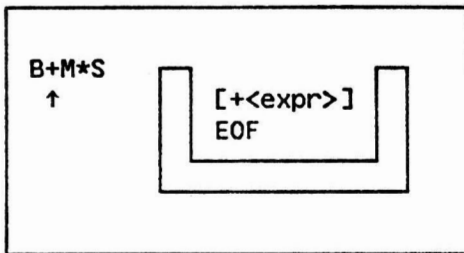
Nach Anwenden von Regel 3 und 5 ist STACKTOP ein Terminalzeichen ('ID'):



Es wird mit dem aktuellen Eingabesymbol verglichen. Da Übereinstimmung besteht (B ist ein identifizier), wird das nächste Zeichen eingelesen und STACKTOP gelöscht:



$\text{PARSING_TABLE}(+,[*\langle \text{term} \rangle]) = \text{LAMBDA}$. Daher wird nun das oberste Stacksymbol gelöscht:



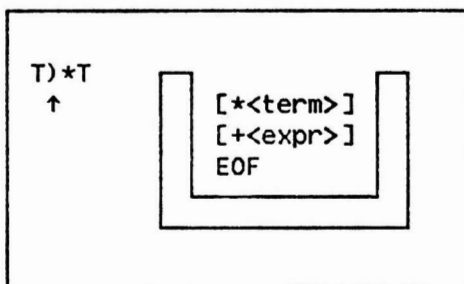
Nach einigen weiteren Schritten, die sich prinzipiell nicht von den oben beschriebenen unterscheiden, wird der Keller leer, und die Eingabe ist damit akzeptiert.

2.2 Fehlererkennung

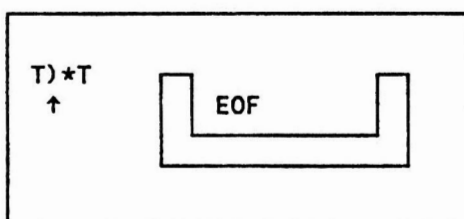
Aus dem Parsing-Algorithmus ergibt sich, daß Fehler auf zwei mögliche Arten erkannt werden:

- ◊ STACKTOP ist ein Terminalzeichen, stimmt jedoch nicht mit dem aktuellen Eingabesymbol überein.
- ◊ STACKTOP ist ein Nonterminal und die Tabellensuche ergibt ERROR.

Es soll nun demonstriert werden, wie der fehlerhafte Ausdruck $T) * T$ bis zum Erkennen des Fehlers analysiert wird. Bis zum Erreichen der folgenden Konfiguration verläuft alles genauso wie im ersten Beispiel:



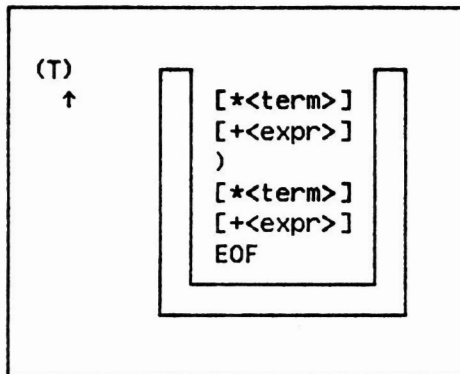
Wegen $\text{PARSING_TABLE}(')', [*<term>]) = \text{LAMBDA}$ und $\text{PARSING_TABLE}(')', [+<expr>]) = \text{LAMBDA}$ werden $[*<term>]$ und $[+<expr>]$ gepoppt, und man erreicht die folgende Fehlerkonfiguration (STACKTOP und aktuelles Eingabesymbol stimmen nicht überein).



Trotz **valid-prefix property** sind die Voraussetzungen für ein gutes error-recovery denkbar schlecht: Der Stack ist so gut wie leer. Alle wichtigen Informationen über mögliche Fortsetzungen der bisher gelesenen Eingabe sind verloren. Es wird nur noch end-of-file erwartet.

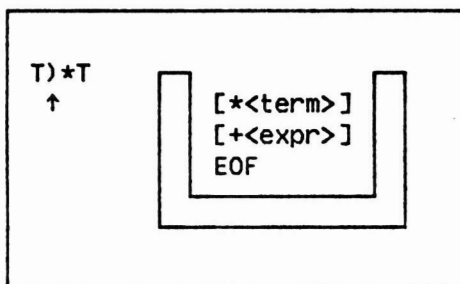
Hier wird eine noch stärkere Eigenschaft als die valid-prefix property benötigt: die sogenannte **immediate-error-detection property** (IEDP) [7]. Der Parser sollte sofort anhalten, wenn ')' das aktuelle Eingabesymbol ist. Stattdessen führt er hier noch weitere Aktionen nach Erreichen des fehlerhaften Symbols aus: $[*<term>]$ und $[+<expr>]$ werden gepoppt.

Wie kann man nun den Parser so modifizieren, daß er die IEDP bekommt, d.h. daß er stets sofort bei Erreichen eines fehlerhaften Symbols hält? Die Möglichkeit, bei ')' und $[*<term>]$ in die Tabelle ERROR einzutragen scheidet aus, da ')' im lookahead-set von $[*<term>]$ $\rightarrow$ LAMBDA liegt, d.h. weil nicht immer ein Fehler vorliegt, wenn $\text{TOKEN} = ')' \text{ und } \text{STACKTOP} = [*<term>]$. Beispielsweise ist die folgende Konfiguration gültig:



Einige der LAMBDA-Einträge in der Tabelle könnten tatsächlich durch ERROR ersetzt werden, dies stellt allerdings keine allgemeine Lösung des Problems dar. Die IEDP kann durch die aufwendigere "schwache" LL(1)-Analyse oder durch eine Erweiterung der Grammatik gewährleistet werden. Auf die damit zusammenhängende Problematik soll aber hier nicht näher eingegangen werden ("strong LL(1)-parsing" siehe [7]).

Die IEDP kann auch einfacher erreicht werden, indem stets **vor** Anwendung von LAMBDA-Regeln der Stack **inspiziert** wird: Es wird getestet, ob überhaupt das aktuelle Eingabesymbol an erster Stelle aus dem Stackinhalt abgeleitet werden kann (ob es also zur Menge der legalen Folgesymbole gehört). Das hat dann zur Folge, daß in unserem Beispiel der Fehler früher erkannt wird:



Dies ist bereits eine Fehlerkonfiguration, da ')' nicht das erste Zeichen eines aus dem Keller ableitbaren Terminalstrings darstellt.

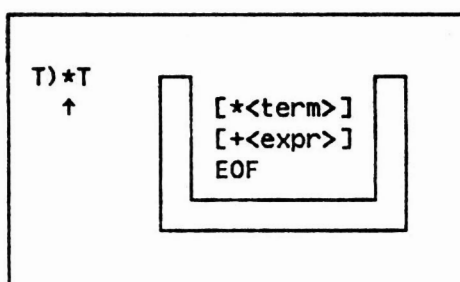
Es sei hier erwähnt, daß die zusätzliche (prophylaktische) Inspektion des Stacks bei geschickter Implementation keinen nennenswerten zusätzlichen Zeitverbrauch bedeutet. Für den Test müssen nämlich nur Tabellenaufsuchen **vorgezogen** werden, die auch beim normalen Parsen kurze Zeit später notwendig sind.

Die Modifikation des Parsers ist für solche LL(1)-Grammatiken besonders einfach, bei denen LAMBDA nur **direkt** (in einem Schritt) abgeleitet werden kann (sog. "nonnullable grammars" vgl. [7]), ist aber prinzipiell für alle LL(1)-Grammatiken möglich.

Die IEDP bringt zwei wesentliche Vorteile:

- Beim Erkennen des Fehlers können **alle** anstelle des falschen Symbols erlaubten Symbole ("legale Folgesymbole") gemeldet werden. Z.B.: "'+' '*' or end-of-file expected". Das ist besonders im Hinblick auf **automatisch** generierte Fehlermeldungen wichtig. Ein Parser ohne IEDP kann nur melden: "end-of-file expected".
- Es ist mehr Information vorhanden, auf deren Grundlage nach einer sinnvollen recovery-Aktion gesucht werden kann: Aus dem Stackinhalt können **alle** Terminalstrings abgeleitet werden, die die bereits gelesene Eingabe richtig verlängern.

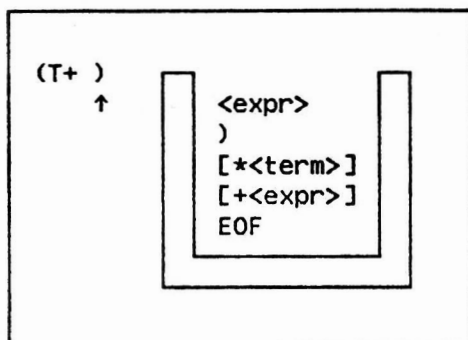
2.3 Recovery durch Überlesen



Im Fehlerfall sind verschiedene Reaktionen denkbar, die einfachste besteht im Überlesen von einem oder mehreren Eingabesymbolen.

In dem oben genannten Beispiel erreicht man durch Überlesen von ')' die nebenstehende gültige Konfiguration und der restliche Input kann ohne weitere Fehlermeldungen analysiert werden.

2.4 Recovery durch Einfügen von Symbolen



Gegeben sei die abgebildete Fehlerkonfiguration.

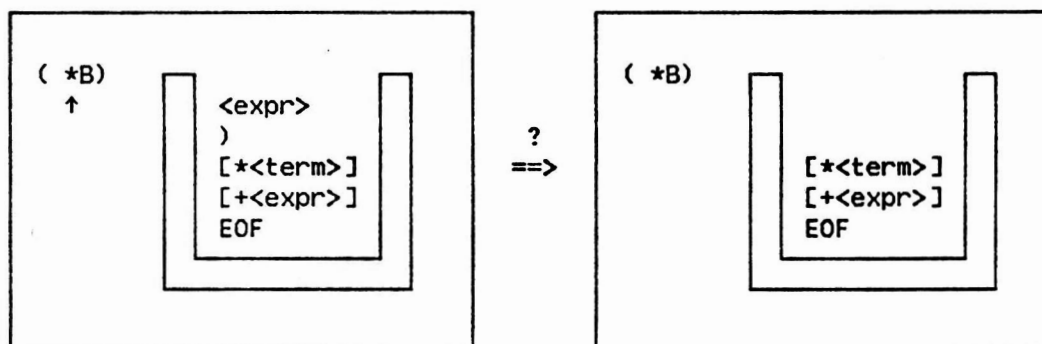
Der Stack sagt aus, was vom restlichen Input noch **erwartet** wird: Zunächst eine expression, dann ')'. Da das aktuelle Eingabesymbol ')' ist, liegt die Annahme nahe, daß die expression (obwohl erwartet) fehlt. Sie kann (virtuell) **eingefügt** werden, indem der Stack **gepoppt** wird. Die **Erwartung**, daß noch eine expression kommt, wird **fallengelassen**. Mit der so erreichten Konfiguration kann wiederaufgesetzt werden.

Durch Entfernen von Symbolen aus dem Stack wird eine **gültige Konfiguration** erreicht, wenn das neue oberste Stacksymbol dem aktuellen Eingabesymbol entspricht. Es genügt aber auch, wenn an der Kellerspitze ein Nonterminal erscheint, aus dem das aktuelle Eingabesymbol an erster Stelle **abgeleitet** werden kann, wenn also gilt: $TOKEN \in FIRST(STACKTOP)$. Das ist genau dann der Fall, wenn $PARSING-TABLE(TOKEN, STACKTOP)$ eine Regelnummer liefert!

In dem hier beschriebenen Verfahren werden nur solche insertions vorgenommen, die durch poppen des Stacks simuliert werden können. Es werden also nur diejenigen Symbole als **Aufsetzpunkte** benutzt, die mit Sicherheit einmal in einer zukünftigen Konfiguration **legale Folgesymbole** sein werden, ganz gleich wie der Fehler letztlich behoben wird. Für das recovery wird also eine **dynamisch** berechnete Menge von Symbolen verwendet, die in dem oben genannten Sinne schon 'erwartet' werden.

2.5 Immediate Prediction

Wir betrachten die folgende Fehlerkonfiguration.



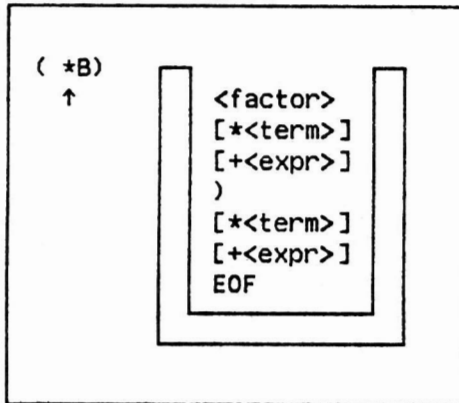
Nach der bisher geschilderten Methode könnte **<expr>** und **)** aus dem Stack entfernt werden, also virtuell eingefügt werden. Bei Erreichen von **)** in der Eingabe gäbe es einen Folgefehler, da angenommen wurde, daß **)** schon vor ***** fehlte.

Das Problem liegt hier offenbar darin, daß der Parser nicht 'weiß', daß auch schon in **<expr>** (dem **obersten** Stacksymbol) ein ***** vorkommen kann. Andererseits wird **<expr>** stets nach **<factor>** **[*<term>] [+<expr>]** abgeleitet. Wenn also eine expression erwartet wird, so werden mit Sicherheit einmal ***** und **+** legale Forsetzungszeichen sein.

Um die aus dem Stack **direkt** ablesbare Information zu erhöhen, kann der Parser sofort bei Einbringen eines Nonterminals in den Stack eine weitere Regel anwenden, wenn diese eindeutig gegeben ist (**immediate-prediction property**).

Beim Eintragen von $\langle \text{expr} \rangle$ wird dann erkannt, daß $\langle \text{expr} \rangle$ nur nach Regel 1 abgeleitet werden kann. Daher wird gleich $\langle \text{term} \rangle [+ \langle \text{expr} \rangle]$ anstelle von $\langle \text{expr} \rangle$ eingetragen. Beim Eintragen von $\langle \text{term} \rangle$ wird wiederum (rekursiv) erkannt, daß $\langle \text{term} \rangle$ nur nach Regel 3 abgeleitet werden kann und daher statt $\langle \text{term} \rangle$ gleich $\langle \text{factor} \rangle [* \langle \text{term} \rangle]$ eingetragen.

Ein Parser, der die **immediate-prediction property** hat, erreicht daher folgende Fehlerkonfiguration:

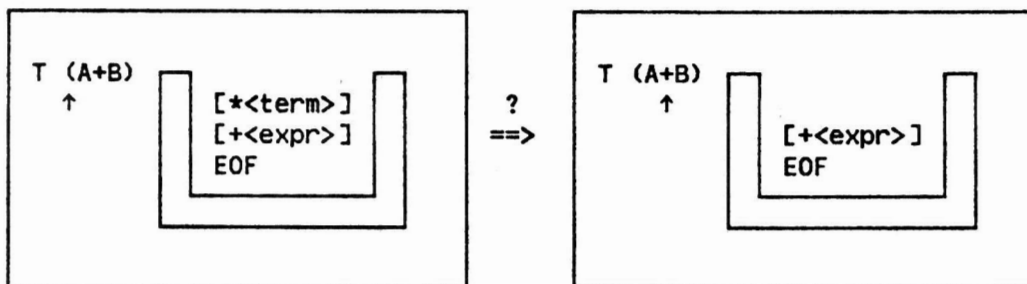


Der Fehler kann jetzt durch Einfügen von $\langle \text{factor} \rangle$ – also durch Entfernen aus dem Keller – behoben werden, ohne daß Folgefehler auftreten.

Bei der Analyse von PASCAL-Programmen findet **immediate-prediction** beispielsweise bei $\langle \text{compound-stmt} \rangle$ statt: Anstelle von $\langle \text{compound-stmt} \rangle$ wird immer schon **begin** $\langle \text{stmt-list} \rangle$ **end** erwartet. Dadurch wird es möglich, mit einem Statementanfang (z.B. 'if') in die nun erwartete Statementlist einzusteigen, falls das 'begin' (z.B. wegen eines Schreibfehlers) nicht erkannt wurde.

2.6 Heuristisches Parsen

In der gegebenen Fehlerkonfiguration ist nach der bisher beschriebenen Methode das Einfügen eines Operators nicht möglich, da es nicht durch Entfernen ganzer syntaktischer Einheiten aus dem Stack simuliert werden kann.



Stattdessen könnte der Parser 'A' überlesen, $[* \langle \text{term} \rangle]$ entfernen und mit dem erwarteten '+' wieder aufsetzen. Es gäbe dann einen Folgefehler bei ')'. Offenbar ist es aber sinnvoller, nicht alle Zeichen einzeln zu überlesen, sondern beim Erreichen von '(' eine ganze (eingeklammerte) expression zu überlesen. Der Parser pusht daher $\langle \text{expr} \rangle$ auf den Stack: '(A+B)' wird **heuristisch geparkt**. Die expression wird als ganzes überlesen und analysiert. Das eigentliche Wiedereinsteigen geschieht erst bei EOF.

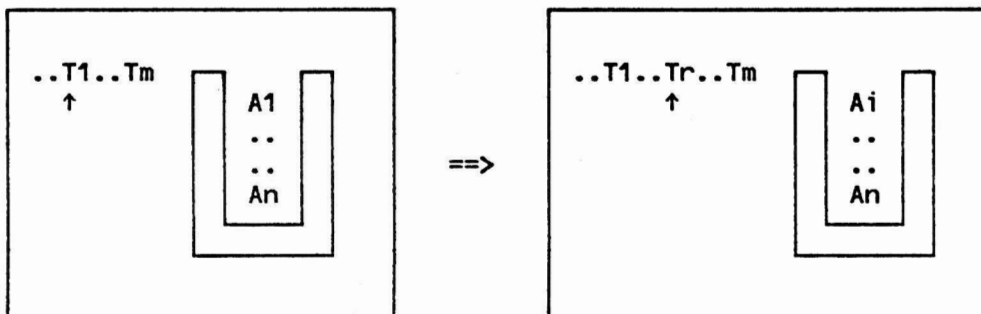
In PASCAL werden für das heuristische Parsen typischerweise wichtige Schlüsselwörter verwendet, die eine eindeutige Phrase einleiten:

- **begin** ==> push($\langle \text{compound-stmt} \rangle$)
- **if** ==> push($\langle \text{if-stmt} \rangle$)
- **do** ==> push($\langle \text{do-stmt} \rangle$)

So können syntaktische Einheiten auch dann analysiert werden, wenn sie völlig deplaziert stehen, wie z.B. eine Deklaration im Statementteil. Es werden dadurch etwaige Fehler in der Deklaration gefunden und ein falsches Einsteigen in den Statementteil mit Symbolen, die zur Deklaration gehören, wird vermieden.

3. Der recovery-Algorithmus

Der recovery-Algorithmus überführt eine Fehlerkonfiguration in eine gültige Konfiguration:



Eine **gültige** Konfiguration liegt genau dann vor, wenn gilt:

$Tr=A_i$ oder $Tr \in \text{FIRST}(A_i)$ ($\Leftrightarrow \text{PARSING-TABLE}(Tr, A_i)$ ist eine Regelnummer)

Unter diesen Voraussetzungen kann nämlich der tabellengesteuerte Parser die Analyse – um mindestens ein Zeichen – fortsetzen. Um eine gültige Konfiguration zu erreichen, werden zwei mögliche Arten von Modifikationen in Betracht gezogen:

- Überlesen einiger Eingabezeichen ($T_1 \dots Tr-1$), evtl. mit heuristischer Analyse
- Poppen einiger Symbole aus dem Stack ($A_1 \dots A_{i-1}$)

Ziel des recovery-Algorithmus ist es nicht allein, irgendeine gültige Konfiguration herzustellen: Es soll **richtig** eingestiegen werden, so daß keine **Folgefehler** gemeldet werden. Ein richtiges Einsteigen liegt dann vor, wenn die restliche Eingabe – sofern sie keine weiteren Fehler enthält – akzeptiert wird, also aus dem Stackinhalt **ableitbar** ist ($A_i \dots A_n \Rightarrow^* Tr \dots T_m$). Sicher ist aber lediglich, daß das aktuelle Eingabesymbol an erster Stelle aus dem Stackinhalt ableitbar ist; die restliche Eingabe ist noch nicht bekannt. Es muß also versucht werden, an einer Stelle einzusteigen, wo möglichst sichere **Annahmen** über die weitere Eingabe möglich sind.

Eine grundlegende Annahme jedes recovery-Verfahrens ist, daß ein falsches Programm ein "near miss" ist, also einem richtigen Programm stark ähnelt. Eine hohe Wahrscheinlichkeit, daß eine recovery-Aktion richtig ist, ist daher gegeben, wenn die folgenden drei Punkte erfüllt sind:

- Es wurden nur wenige Eingabesymbole überlesen.
- Die "Kosten" für die eingefügten Symbole sind klein.
- Es wurde mit einem **verlässlichen** Symbol eingestiegen, das stets eine eindeutige Phrase einleitet und mit hoher Wahrscheinlichkeit nicht zufällig oder fälschlich in der Eingabe steht.

Dem ersten Punkt wird Rechnung getragen, indem grundsätzlich erst dann ein Symbol überlesen wird, wenn alle anderen Möglichkeiten ausscheiden.

Damit die Kosten für eingefügte Symbole berechnet werden können, muß der Compilerentwickler zu jedem Terminalsymbol der Sprache die **INSERT-COSTS** spezifizieren. Wichtigen Schlüsselwörtern werden hohe Kosten zugeordnet, Symbolen, die leicht vergessen werden (z.B. Semikolon), werden geringe Kosten zugeordnet. Das Aufstellen der Kostentabelle wird zwar von Hand vorgenommen, es kann dabei aber schematisch nach einfachen Heuristiken vorgegangen werden. Die Funktion **INSERT-COSTS** kann in natürlicher Weise auf **Terminalstrings** erweitert werden:

$$\text{INSERT-COSTS}(ab) = \text{INSERT-COSTS}(a) + \text{INSERT-COSTS}(b)$$

Einem **Nonterminal** X werden die **INSERT-COSTS** des billigsten aus X ableitbaren Terminalstrings zugeordnet ($\text{INSERT-COSTS}(\text{LAMBDA})=0$).

Die **Verlässlichkeit** (**RELIABILITY**) von Terminalzeichen wird ebenfalls (sprachabhängig) vom Compilerschreiber spezifiziert (Die **RELIABILITY** von EOF sei unendlich). Es wird nur dann eingestiegen, wenn die Kosten für die eingefügten (gepoppten Stack-) Symbole kleiner sind als die **RELIABILITY** des gefundenen Eingabesymbols.

Der recovery-Algorithmus findet das kleinste r ($1 \leq r \leq m$), für das es ein i ($1 \leq i \leq n$) gibt, so daß gilt:

- $Tr \in \text{FIRST}(Ai)$ ($\Leftrightarrow \text{PARSING-TABLE}(Tr, Ai)$ ist eine Regelnummer) oder $Tr = Ai$
- $\text{INSERT-COSTS}(A1..Ai-1) < \text{RELIABILITY}(Tr)$

Bemerkungen: $A1..Ai-1$ bezeichne für $i=1$ den leeren string LAMBDA. Da $Tm = \text{EOF}$ und $An = \text{EOF}$ und außerdem die RELIABILITY von EOF unendlich ist, gibt es stets ein solches r .

Zu dem gefundenen r wird das kleinste gültige i gewählt und die folgende recovery-Aktion vorgenommen:

- $T1..Tr-1$ überlesen (deletion).
- $A1..Ai-1$ aus dem Stack entfernen (insertion).

Das heuristische Parsen ist hier ausgeklammert, wird aber im folgenden Algorithmus berücksichtigt.

Algorithmus zur Rekonfiguration des Parsers

$A1 \dots An$	ist der Stackinhalt. $\text{STACKTOP} = A1$
TOKEN	ist jeweils das aktuelle Eingabezeichen
UNIQUE(TOKEN)	sagt aus, ob TOKEN eine eindeutige Phrase einleitet
UNIQUE_PHRASE(TOKEN)	gibt die Phrase an, die TOKEN eindeutig einleitet
PREDICT(PHRASE)	schreibt eine Phrase als oberstes in den Keller

```

MESSAGE("illegal symbol:",TOKEN);
loop until RECOVERY do
    loop for I in 1..N
        while INSERT_COSTS(A1...Ai-1) < RELIABILITY(TOKEN)
            do
                if TOKEN=Ai or TOKEN in FIRST(Ai) then signal RECOVERY;
            endloop;
        if UNIQUE(TOKEN)
            then begin (* Heuristic Parsing *)
                    PREDICT(UNIQUE_PHRASE(TOKEN)); (* push *)
                    PARSE; (* Recursive Call *)
                end
            else NEXT_TOKEN; (* Deletion *)

    exit RECOVERY is
        POP(A1...Ai-1); (* Insertion *)
        MESSAGE(TOKEN," assumed to be part of ",Ai);
    endloop;

```

Die äußere Schleife wird solange durchlaufen, bis das Ereignis RECOVERY signalisiert wird und der zugehörige exit-Block ausgeführt wird. Anschließend wird die normale Analyse fortgesetzt. (Das Schleifenkonstrukt wurde in Anlehnung an die Sprache CSSA gewählt.)

Immer wenn eine Möglichkeit zur Rekonfiguration des Parsers gefunden ist, wird durch die Kostenrechnung die folgende Entscheidung getroffen: Ist es wahrscheinlicher, daß die einzufügenden Symbole tatsächlich vom Programmierer vergessen wurden (oder wegen eines Schreibfehlers nicht erkannt wurden), oder ist es wahrscheinlicher, daß das gefundene Symbol fälschlicherweise in der Eingabe steht oder in einen anderen Zusammenhang gehört.

Trotz der restriktiven LL(1)-Voraussetzungen und obwohl der vorgestellte Algorithmus einfach und leicht implementierbar ist, hat der praktische Vergleich mit anderen Verfahren gezeigt, daß er mindestens gleichwertige Ergebnisse liefert (siehe 5.).

4. Ein Beispielprogramm

Die vorgestellte Methode wurde unter anderem für PASCAL implementiert. Zu jedem Terminalzeichen von PASCAL wurden dabei die INSERT-COSTS und die RELIABILITY spezifiziert. Darüberhinaus wurde die Methode noch um zwei nicht völlig sprachunabhängige Techniken erweitert, die sich aber auf alle Sprachen mit Schlüsselwörtern und deklarierten Bezeichnern anwenden lassen:

- **spelling-correction** (—> Zeile 14,15,16)
- **vorgezogene Symboltabellenaufsuche** (—> Zeile 15)

Das Beispiel zeigt, wie der PASCAL-Parser auf ein **Demonstrationsprogramm** reagiert. Die Fehlerzeilen sind durch einen Pfeil markiert, das fehlerhafte Zeichen ist unterstrichen. Überlesene Passagen werden gestrichelt unterstrichen. Im Protokoll wird zwischen den eigentlichen Fehlermeldungen und den Annahmen des Parsers für das recovery unterschieden. Man beachte, daß alle Fehlermeldungen **automatisch** generiert werden und **nicht** speziell auf PASCAL zugeschnitten sind.

Erläuterungen zu den Fehlermeldungen

- **Zeile 14:** In der Meldung werden alle legalen Fortsetzungszeichen angegeben (nur wegen IEDP möglich!). Eines davon ('else') ähnelt dem falschen Symbol, daher **spelling-correction**.
- **Zeile 15:** Schon der scanner untersucht, ob und wo ein identifier deklariert wurde, und liefert einen entsprechenden token: <new-identifier>, <local-identifier> bzw. <global-identifier> (**Vorgezogene Symboltabellenaufsuche**). Der Fehler wird daher nicht erst bei '=' erkannt und kann durch spelling-correction behoben werden.
- **Zeile 16:** Spelling-correction durch Vergleich mit der Menge aller deklarierten identifier.
- **Zeile 17:** Ein nicht deklariertes identifier bewirkt einen **Syntaxfehler** (vgl. Zeile 15).
- **Zeile 14-17:** Korrektur auf Tokenebene.
- **Zeile 18:** Recovery durch Überlesen (**deletion**).
- **Zeile 19:** Wichtige überlesene Symbole werden gemeldet.
- **Zeile 20:** 'AUTO=KAPUTT' ist bereits eine vollständige expression. Einstieg mit 'or' ist nur wegen IEDP möglich!
- **Zeile 18-20:** Reine deletion.
- **Zeile 21:** Die RELIABILITY von 'then' ist größer als die INSERT-COSTS für die fehlenden Teile der expression ==> insertion.
- **Zeile 22:** RELIABILITY('else') > INSERT-COSTS('then' <statement>) ==> insertion. Wichtige eingefügte Symbole werden gemeldet.
- **Zeile 21-22:** Reine insertion.
- **Zeile 23:** Kombinierte deletion/insertion. Beachte: Kein Einstieg mit 'FUSSMARSCH' in <statement> - obwohl dies richtig wäre - da INSERT-COSTS von 'then' zu hoch sind.
- **Zeile 26:** INSERT-COSTS('then') > RELIABILITY(';',') ==> kein Einstieg mit ';',
- **Zeile 28:** Kein Einstieg mit 'until' in <repeat-statement> wegen Kostenrechnung: RELIABILITY('until') < INSERT-COSTS ('to' <expr> 'do' <statement> 'end').
- **Zeile 33:** Durch **heuristisches Parsen** der deplazierten label-Deklaration wird der ','-Fehler entdeckt, Folgefehler werden vermieden.
- **Zeile 35:** Begrenzung von Kommentaren und string-Konstanten auf eine Zeile ermöglicht rechtzeitiges recovery (**Sprachdesign**).

DEFAULT-OPTIONS: NOTERM,NOTEST, CHECK, SOURCE,NOSTRUCT, XREF,RESWD=A,MAXPAGES= 99,MAXERROR= 99,MAXDERIV= 99
 OPTIONS IN USE: NOTERM,NOTEST, CHECK, SOURCE, STRUCT, XREF,RESWD=A,MAXPAGES= 99,MAXERROR= 99,MAXDERIV= 99

```

1  -----1-----2-----3-----4-----5-----6-----7-----
2  program FEHLER_DEMO(OUTPUT);
3
4      type WOCHENTAG=(MO,DI,MI,DON,FR,SA,SO);
5      var TAG:          WOCHENTAG;
6          AUTO,KAPUTT,SCHROTTREIF,ALTER:  INTEGER;
7          OELKRISE,CRASH,URLAUB:          BOOLEAN;
8
9      procedure MALOCHEN;          begin          end;
10     procedure AUTOFAHRT;          begin          end;
11     procedure FUSSMARSCH;          begin          end;
12
13     begin
14         if AUTO=KAPUTT              then FUSSMARSCH else AUTOFAHRT;
15         if AUTO=KAPUTT              then FUSSMARSCH ELZE AUTOFAHRT;
16         if IFAUTO=KAPUTT            then FUSSMARSCH else AUTOFAHRT;
17         if BUS=KAPUTT                then FUSSMARSCH else AUTOFAHRT;
18         if AUTO=KAPUTT,SCHROTTREIF then FUSSMARSCH else AUTOFAHRT;
19         if AUTO=KAPUTT until CRASH then FUSSMARSCH else AUTOFAHRT;
20         if AUTO=KAPUTT or OELKRISE then FUSSMARSCH else AUTOFAHRT;
21         if AUTO=                    then FUSSMARSCH else AUTOFAHRT;
22         if AUTO=KAPUTT              else AUTOFAHRT;
23         if AUTO=KAPUTT              DANN FUSSMARSCH else AUTOFAHRT;
24
25     repeat
26         if not URLAUB;
27         then begin
28             for TAG:=MO until FR do MALOCHEN;
29             write(' WOCHENENDE');
30         end;
31     until ALTER=65;
32
33     label 1,2;
34
35     write('0815 ) ???????  (* KOMMENTAR-ENDE FEHLT
36     end.
  
```

B L O C K N E S T I N G

```

+1
+2+3-3-2
+4+5-5-4
+6+7-7-6
+1
+8
+9
+10
-10
-9
+8
-8-1
  
```



"THOSE ARE JUST THE ERROR
 MESSAGES. THE PROGRAM
 IS OVER HERE."

[illegible]

LL(1)-ERROR-RECOVERY

5. Praktische Ergebnisse

Daß eine Syntaxfehlerbehandlungsmethode "gut" ist, kann letztlich nicht durch theoretische Überlegungen bewiesen werden, sondern muß durch praktische Experimente gezeigt werden. Die PASCAL-Version des vorgestellten Verfahrens wurde 1981 von über hundert Studenten in mehr als tausend Programmen getestet. Es zeigte sich, daß das Verfahren tatsächlich nur sehr selten Folgefehler produziert und fast nie Fehler übersieht. Die erzeugten Fehlermeldungen waren auch für PASCAL-Anfänger verständlich.

Darüberhinaus wurde der PASCAL-Parser mit dem Fehlerbenchmark von Ripley und Druseikis [5] getestet. Das Material stammt aus 237 fehlerhaften PASCAL-Programmen (etwa 12000 Zeilen), die von Studenten geschrieben wurden. Die Fehlerstelle wurde isoliert und mit einem minimalen Kontext versehen. Gleiche Fehler wurden nur einmal aufgenommen, aber mit einem Gewichtungsfaktor versehen. Es ergaben sich so 127 kleine Beispielprogramme, die man bei Ripley anfordern kann.

Die erzeugten Fehlerdiagnosen wurden in 3 Kategorien eingeordnet:

- ◊ "excellent": Ein Mensch würde die gleiche Diagnose liefern
- ◊ "good": Falsche Annahmen des Parsers, aber keine Folgefehler
- ◊ "poor": Ein oder mehrere Folgefehler

Es ergab sich die mit 'BMS' bezeichnete Spalte in der Tabelle. Der PASCAL/VS-Compiler von IBM führt ein error-recovery nach der Methode von Wirth [1] durch. Er lieferte das mit 'IBM' bezeichnete Ergebnis. Das zweitbeste uns bekannte Verfahren, das ebenfalls mit dem Fehlerbenchmark evaluiert wurde, ist das "Global Context Recovery" von Pai und Kiebertz [4]. Sie erhielten das mit GCR bezeichnete Ergebnis.

	GCR	IBM	BMS
excellent	52%	68%	77%
good	26%	9%	14%
poor	22%	23%	9%

Das vorgestellte Verfahren führt also in 90% der Fälle richtige recovery-Maßnahmen durch und vermeidet so Folgefehler. Falsche Reaktionen des Parsers waren in der Mehrzahl auf falsch verwendete Schlüsselwörter oder fehlende Kommentarklammern und Hochkommata zurückzuführen. Solche Fehler stellen für

alle Methoden ein großes Problem dar. Sie sollten daher durch ein geeignetes Sprachdesign nach Möglichkeit von vorneherein vermieden werden (vgl. Kommentare in Ada).

6. Literatur

- [1] Wirth, N.: "Die Behandlung von syntaktischen Fehlern" in: Compilerbau Teubner-Verlag 1977
- [2] Moll, Karl-Rudolf: "Suffixanalyse, ein Konzept zur Behandlung von syntaktischen Fehlern", Informatik Spektrum 4, 1981, pp 82-89
- [3] Fischer, Milton, Quiring: "Efficient LL(1) Error-Correction and Recovery using Only Insertions", Acta Informatica, Vol.13 No.2, 1980, pp141-154
- [4] Pai, Kiebertz: "Global Context Recovery: A New Strategy for Syntactic Error Recovery by Table-Driven Parsers", ACM TOPLAS, Vol.2, no.1, 1980, pp18-41
- [5] Ripley, Druseikis: "A Statistical Analysis of Syntax Errors", Computer Languages, Vol.3 No.4, pp 227-239, 1978
- [6] Aho, Ullman: "Error Detection and Recovery", in: Principles of Compiler Design, Chapter 11, pp382-405, 1977, Addison-Wesley
- [7] Fischer, Tai, Milton: "Immediate Error Detection in Strong LL(1) Parsers", Information Processing Letters, Vol.8 No.5, pp261-266, 1979
- [8] Graham, Rhodes: "Practical Syntactic Error Recovery", CACM, Vol.18 No.11, pp639-650, 1975
- [9] Beilken, Mattern, Spenke: "Bibliography of Error Handling in Compilers". Die sehr umfangreiche Bibliographie (über 200 Titel) enthält abstracts und Querverweise und kann beim Autor angefordert werden.

