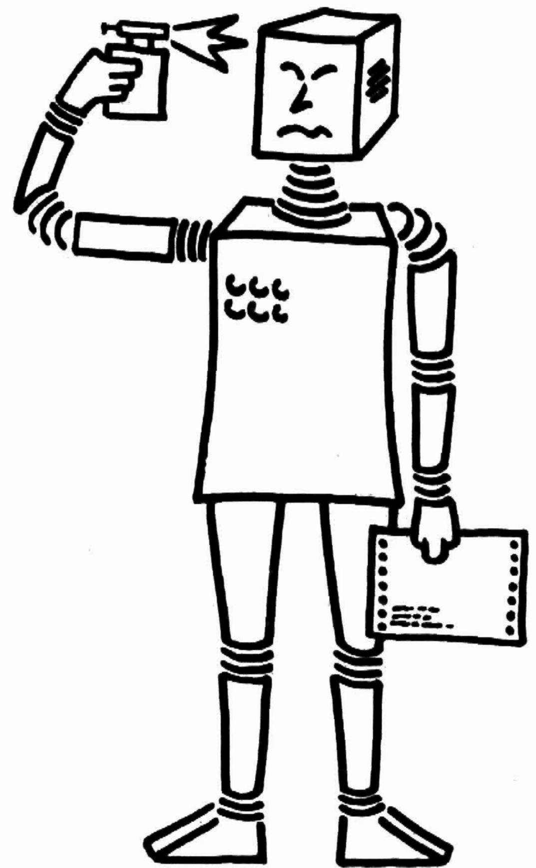
SEKI-PROJEKT

SEKI MEMO

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



Entwurf und Implementierung von CSSA..

Teil C: CSSA-Systembenutzung

MEMO-SEKI-82-03-C

Christian Beilken,
Friedemann Mattern,
Michael Spenke

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

Teil C :
CSSA-Systembenutzung

Christian Beilken
Friedemann Mattern
Michael Spenke

Druck: Abteilung Foto-Repro-Druck der Universität Kaiserslautern

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

Volume C:
User's Guide

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 language described in volume B of the CSSA-documentation 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 manual describes the use of the CSSA-system consisting of the compiler, the multiprocessor-simulation system, the command-interpreter and the interactive debugging-system. All system-components are integrated in a menue-driven program-development system.

A detailed discussion of the language concepts and the simulation system can be found in other volumes of the CSSA-documentation.

Vorwort

CSSA (Computing System for Societies of Agents) ist eine Programmiersprache für 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 Entwicklung eines Übersetzers für CSSA wurde 1981 abgeschlossen, er ist in ein Multiprozessor-Simulationssystem integriert und erzeugt Code, der von diesem Simulationssystem ausgeführt wird. Erste Anwendungen und Erfahrungen mit diesem System liegen bereits vor. Die Implementierung von CSSA auf einem realen Multi-Mikrocomputersystem befindet sich 1982 in Vorbereitung.

In diesem Handbuch wird in ausführlicher Form der Umgang mit dem unter Siemens BS2000 laufenden CSSA-System beschrieben. Nach einem kurzen Überblick über die verschiedenen Systemkomponenten (Kap. 2) wird zunächst in Kap. 3 das Menue-gesteuerte Entwicklungssystem erläutert, in das alle anderen Systemkomponenten integriert sind. In Kapitel 4 wird die Benutzung des CSSA-Übersetzers beschrieben, insbesondere finden sich dort Erläuterungen zu allen Fehlermeldungen. Der Ablauf einer interaktiven CSSA-Sitzung, in der ein Agentennetz aufgebaut und eine verteilte Berechnung durchgeführt wird, wird in Kapitel 5 geschildert. Dort wird auch der interaktive debugger und die Steuerung des Simulationssystems erläutert.

Das vorliegende Handbuch ist im wesentlichen als ein Nachschlagewerk konzipiert, eine Vielzahl von Querverweisen sollen diesen Zweck unterstützen. Es ist Teil der Diplomarbeit ("Entwurf und Implementierung von CSSA - Beschreibung der Sprache, des Compilers und des Mehrrechnersimulationssystems") von C. Beilken, F. Mattern und M. Spenke, andere Teile dieser Arbeit beschreiben genauer die CSSA-Sprachkonzepte, ein weiterer enthält eine Sammlung von Beispielprogrammen. Eine ausführliche Dokumentation des CSSA-Systems mit Quellprogrammen findet sich in Teil E.

In Bezug auf eine Diskussion der CSSA-Konzepte, einen Überblick über das Gesamtsystem und auch bezüglich weiterführender Literaturhinweise sei auf Teil A verwiesen.

BMS, im Juni 1982

Inhalt

1.	**Einleitung**	2
2.	**Überblick über das CSSA-System**	4
3.	**Das Entwicklungssystem**	8
4.	**Der Compiler**	15
4.1	**Ein- und Ausgabeverhalten des Compilers**	15
4.2	**Eingabeformat des Quellprogramms**	17
4.3	**Options und Steueranweisungen**	18
4.3.1	Options	18
4.3.2	Steueranweisungen	21
4.4	**Layout des source-listing**	23
4.4.1	Beispiel	23
4.4.2	Erläuterung	24
4.5	**Cross-reference Liste**	27
4.6	**Syntaxbäume**	28
4.7	**Fehlermeldungen**	30
4.7.1	Fehlerdiagnose	30
4.7.2	Allgemeines	32
4.7.3	Problematik der Fehlerbehandlung	34
4.7.4	Prinzip der verwendeten Methode	35
4.7.5	Syntaxfehler	36
4.7.6	Semantikfehler	37
4.7.7	Assumption-Meldungen	38
4.7.8	Allgemeine Fehlerhinweise	39
4.7.9	Liste der semantischen Fehlermeldungen	40
4.8	**Externe Funktionen und Prozeduren**	46
4.9	**Compiler-Restriktionen**	52
5.	**Die interaktive CSSA-Sitzung**	54
5.1	**Ablauf einer CSSA-Sitzung**	54
5.2	**CSSA-Teilsprache**	56
5.3	**Steuerung des Simulationssystems**	59
5.4	**Der interaktive debugger**	64
5.5	**Fehlerbehandlung**	68

Anhang A: Sitzungsprotokolle

Anhang B: Interpreter-Syntax

1. Einleitung

Das vorliegende Handbuch beschreibt als Teil C der Arbeit "Entwurf und Implementierung von CSSA" detailliert die **Benutzung** des CSSA-Systems und seiner Komponenten. Zu diesen Komponenten gehören u.a.

- das **Entwicklungssystem,** in das die anderen Systemteile integriert sind und das eine einfache aber flexible menügesteuerte Benutzung aller Systemkomponenten erlaubt. Die Möglichkeiten des Entwicklungssystems werden ausführlich beschrieben.

- der **Compiler,** der CSSA-Quellprogramme in eine Zwischensprache in SIMULA-Syntax übersetzt. Die Steuerung des Übersetzers durch parametrisierte options und Steueranweisungen wird erläutert, das Ein- und Ausgabeformat der Dateien wird beschrieben und Hinweise zu Fehlermeldungen und dem Anschluß externer Funktionen und Prozeduren werden gegeben.

- der **interface-Agent,** der eine Teilmenge der CSSA-Sprache interpretativ verarbeitet und eine integrierte, interaktive Benutzerschnittstelle darstellt. Die Teilsprache wird beschrieben, eine vollständige Syntax in erweiterter BNF-Notation befindet sich in Anhang B dieses Handbuches.

- das **Simulationssystem,** mit dem verteilte Berechnungen auf Mehrprozessorkonfigurationen simuliert werden. Die möglichen Angaben zur Konfigurationsspezifikation werden ebenso beschrieben wie die Kommandos, die der Überwachung und Steuerung des Simulationssystems dienen.

- der **interaktive debugger,** der es erlaubt, unterschiedlich detaillierte Informationen über den Zustand einzelner Agenten oder des gesamten Agentennetzes auszugeben und die Protokollierung zu steuern. Die Kommandos, mit denen der debugger gesteuert wird und die verschiedenen Formen der Protokollausgabe werden erläutert.

Da dieses Handbuch keine Funktionsbeschreibung der einzelnen Systemkomponenten darstellt und nicht auf die Realisierung der Konzepte und Komponenten eingegangen wird, sondern lediglich die (äußere) Benutzung der Komponenten behandelt wird, werden hier weitere Funktionseinheiten wie die virtuelle stack-Maschine oder der lokale Betriebssystemkern nicht behandelt. Es sei statt dessen auf die anderen Teile der vorliegenden Arbeit verwiesen:

- Die Designkriterien einzelner CSSA-Sprachkonstrukte und Systemkomponenten, eine Diskussion der CSSA-Konzepte, Vergleiche mit anderen Programmiersprachen, die Konzepte des Betriebssystemkerns, die Simulation der Konfiguration und des Betriebssystems und andere eher **globale Aspekte** werden in **Teil A** behandelt. Dort befindet sich auch ein ausführliches **Literaturverzeichnis.**

- **Teil B** stellt eine vollständige und systematische Beschreibung der vom Compiler akzeptierten **CSSA-Sprache** dar. Das Handbuch soll einerseits einen leichten Zugang zu CSSA ver-

mitteln, ist andererseits aber auch als Nachschlagewerk konzipiert.

- **Teil D** enthält neben **Testprogrammen** zu allen Sprachkonstrukten, die die Semantik, Besonderheiten und Einschränkungen demonstrieren, eine Sammlung von kommentierten **Anwendungsbeispielen** zu Problemlösungen mit CSSA. Neben einer Problembeschreibung sind die Programmlistings, teilweise mit Systemprotokollen, abgedruckt.

- Die **Dokumentation** des Systems befindet sich in **Teil E.** Dort sind die Quellprogramme der Systemkomponenten abgedruckt und dort findet man bspw. auch den Aufbau der Kontrollblöcke des (simulierten) Betriebssystems. Diese Information wird benötigt, wenn bspw. über externe Prozeduren in das System eingegriffen werden soll.

Es wird bei den nachfolgenden Benutzungshinweisen grundsätzlich davon ausgegangen, daß das CSSA-System in der für die **Siemens 7760 Anlage** der **GMD** konzipierten Version und unter Verwendung des als BS2000 Kommandoprozedur realisierten **Entwicklungssystems** benutzt wird. Auf die CSSA-Implementierung auf der IBM /370 wird hier nicht eingegangen.

Das **Entwicklungssystem** ist an die Systemumgebung der GMD angepaßt und wegen der angesprochenen speziellen Geräte (wie bspw. Laserdrucker) und der verwendeten Hilfssoftware nicht unmittelbar auf andere Installationen übertragbar. Dagegen sind der **Compiler** und das die restlichen Systemkomponenten umfassende **Laufzeitsystem** in der höheren Programmiersprache SIMULA geschrieben und daher relativ problemlos auf andere Systeme übertragbar, sofern dort ein SIMULA-Compiler zur Verfügung steht. Bemerkungen zur Portabilität des CSSA-Systems finden sich in Teil E.

Insbesondere dann, wenn Änderungen am Entwicklungssystem vorgenommen werden sollen oder externe Funktionen oder Prozeduren codiert werden, können die einschlägigen Siemens BS2000 Handbücher und die entsprechenden SIMULA Handbücher nützlich sein.

Es sei abschließend noch ausdrücklich auf das in Anhang A abgedruckte **Beispiel-Sitzungsprotokoll** hingewiesen: Es enthält fast alle Interpreterkommandos (Simulationsanweisungen, Debugginganweisungen) und einige typische Sprachelemente des interface-Agenten. Ein Studium dieses Beispiels ist sehr lohnenswert, da damit die vielfältigen Möglichkeiten, die während einer interaktiven CSSA-Sitzung zur Verfügung gestellt werden, verdeutlicht werden.

2. Überblick über das CSSA-System

CSSA ist eine interaktive Programmiersprache für asynchrone Prozesse, **Agenten** genannt, die ausschließlich durch Nachrichten miteinander kommunizieren. Der Benutzer ist über eine in das System integrierte Schnittstelle, den **interface-Agenten**, Teil des aus den Agenten als Berechnungseinheiten gebildeten heterarchischen **Agentennetzes**, mit dem die verteilten Berechnungen ausgeführt werden.

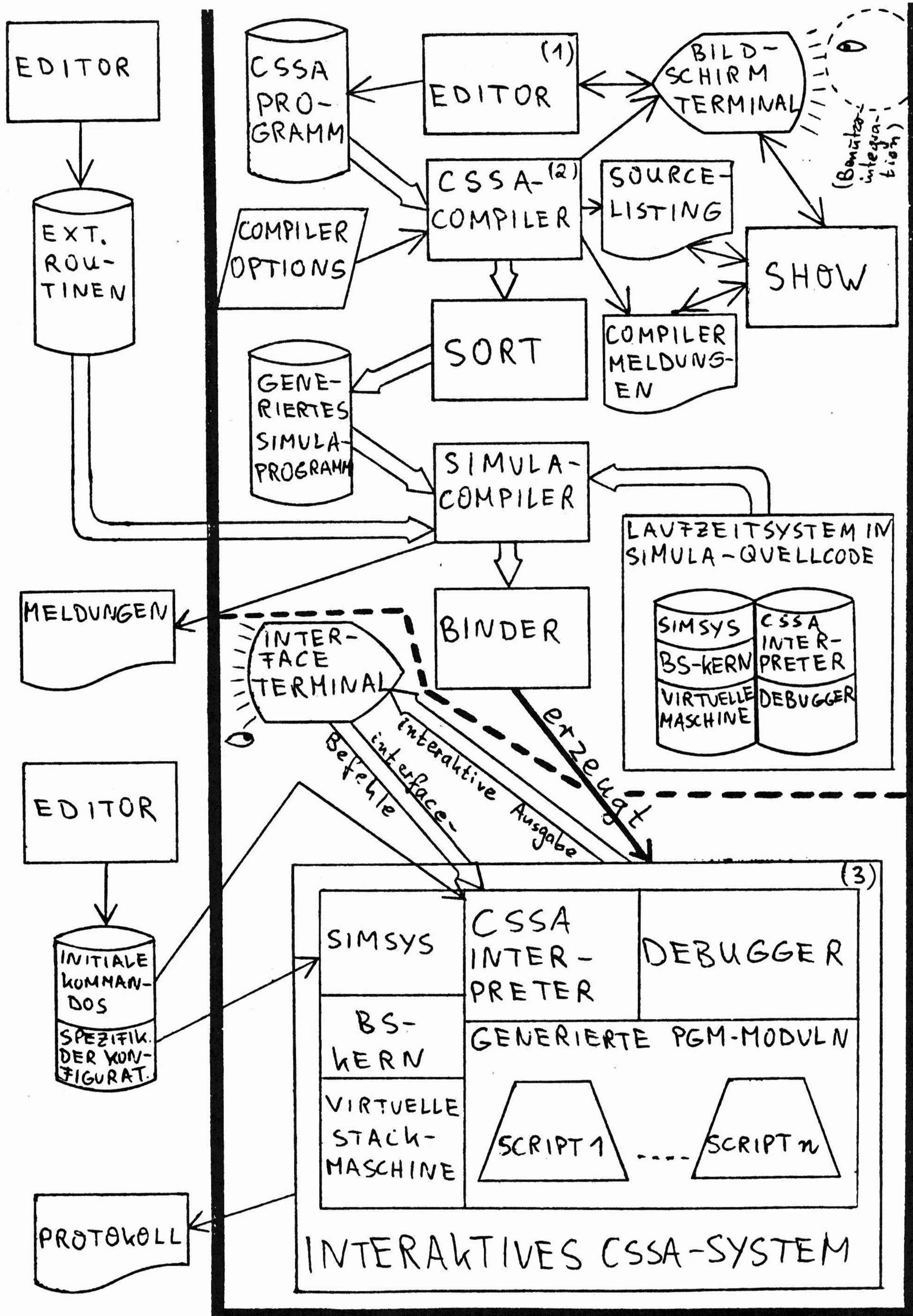
Durch den **interface-Agenten** kann der Benutzer in das Berechnungssystem eingreifen. Dieser initial im System existierende Agent, der standardmäßig allen anderen Agenten bekannt ist, verfügt im Unterschied zu diesen nicht über ein fest definiertes script, das sein Verhalten festlegt, sondern ist dynamisch programmierbar: Die über ein Bildschirmterminal eingegebenen Instruktionen werden interpretativ verarbeitet.

Das **CSSA-Konzept** sieht vor, daß der Benutzer während einer **CSSA-Sitzung** mit dem interface-Agent scripts mit einem Editor erstellen und mit einem Compiler in ausführbare Module transformieren kann oder bereits vorhandene scripts aus einer Scriptbibliothek in das System hinzuladen kann. Der Benutzer kann lokale Größen deklarieren und Zuweisungen an diese vornehmen, Nachrichten an andere Agenten senden, empfangen und verarbeiten, Agenten gründen und verteilte Berechnungen starten (——> B.1.4).

Im vorliegenden **Simulationssystem** wird von diesem Konzept etwas abgewichen, es ergeben sich einige kleinere Einschränkungen, vor allem aber auch eine Reihe von nützlichen Erweiterungen, die im wesentlichen die Steuerung und Kontrolle des Systems durch Protokollierung, tracing und spezielle privilegierte Anweisungen betreffen und das Entwickeln und Testen von verteilten Berechnungssystemen wesentlich erleichtern (——> 5.4.). Diese Testhilfen müssen bei der Implementierung von CSSA auf einem realen Mehrrechnersystem größtenteils entfallen, da es in diesem Fall keinen systemglobalen Zustand gibt.

Im hier beschrieben CSSA-System müssen vor Beginn der eigentlichen CSSA-Sitzung, in der verteilte Berechnungen ablaufen, bereits alle benötigten scripts in übersetzter Form vorliegen. Es können nicht scripts dynamisch hinzugeladen oder während des Ablaufs einer Berechnung editiert werden. Sollten Änderungen an scripts notwendig sein, so muß die CSSA-Sitzung abgebrochen werden und es müssen alle scripts nach Durchführung der Änderungen erneut übersetzt werden.

Um Benutzern ohne Kenntnisse der Siemens BS2000-Kommandosprache eine einfache Anwendung des CSSA-Systems zu ermöglichen, wurde ein komfortables Menü-gesteuertes **Entwicklungssystem** konzipiert, in das ein Datei-Editor und alle CSSA-Systemkomponenten integriert sind und das eine Umgebung bereitstellt, die vom CSSA-Anwender i.a. nicht verlassen zu werden braucht (——> 3.). Das Entwicklungssystem kapselt den Benutzer vom BS2000 Betriebssystem vollständig ab und vermittelt ihm den Eindruck, ein dediziertes CSSA-System zu benutzen.


EDITOR
EXT. ROU-TINEN
EDITOR
INITIALE KOMMAN-DOS
SPEZIFIK. DER KON-FIGURAT.
MELDUNGEN
PROTOKOLL
CSSA PRO-GRAMM
EDITOR (1)
BILD-SCHIRM TERMINAL
CSSA-COMPILER (2)
SOURCE-LISTING
COMPILER OPTIONS
SHOW
SORT
COMPILER MELDUN-GEN
GENE-RIERTES SIMULA-PROGRAMM
SIMULA-COMPILER
LAUFZEITSYSTEM IN SIMULA-QUELLCODE
SIMSYS
BS-KERN
VIRTUELLE MASCHINE
CSSA INTER-PRETER
DEBUGGER
BINDER
INTER-FACE TERMINAL
Befehle
interface-
interaktive Ausgabe
erzeugt
(3)
SIMSYS
BS-KERN
VIRTUELLE STACK-MASCHINE
CSSA INTER-PRETER
DEBUGGER
GENERIERTE PGM-MODULN
SCRIPT 1
SCRIPT n
INTERAKTIVES CSSA-SYSTEM

Die abgedruckte **Abbildung** zeigt überblicksartig das Zusammenspiel verschiedener **Systemkomponenten** und verwendeter Standardsoftware bei der Benutzung des CSSA-Systems. Die Komponenten innerhalb des dick umrandeten Teils werden vom Entwicklungssystem verwaltet, unterhalb der gestrichelten dickeren Linie ist das interaktive CSSA-System skizziert, mit dem in der eigentlichen CSSA-Sitzung die verteilten Berechnungen ausgeführt werden. Dickere Pfeile symbolisieren den Hauptdatenfluß.

Eine **CSSA-Sitzung** unter Benutzung des Entwicklungssystems wird etwa folgendermaßen ablaufen: Falls in speziellen Fällen externe Routinen, Initialisierungskommandos oder eine eigene Konfigurationsspezifikation verwendet werden sollen, so müssen die entsprechenen Dateien vor Aufruf des Entwicklungssystems erstellt und beim Aufruf genannt werden. Mit dem Entwicklungssystem kann an den in der Abbildung mit (1), (2) oder (3) markierten Stellen des CSSA-Systems aufgesetzt werden; es kann also ein **CSSA-Programm** (bestehend aus einer Folge von scripts) mit dem **Editor** verändert oder neu erstellt werden, ein vorhandenes oder soeben neu erstelltes bzw. verändertes Programm mit dem **CSSA-Compiler** (——> 4.) übersetzt werden und das nach einigen weiteren Schritten daraus generierte oder aus früheren CSSA-Sitzungen noch vorhandene **interaktive CSSA-System** ausgeführt werden. Die Compilermeldungen und das vom CSSA-Compiler generierte source-listing können am Terminal angezeigt werden und/oder ausgedruckt werden. Um die Compiler-options (——> 4.3.1) braucht sich der Benutzer nicht zu kümmern, diese werden vom Entwicklungssystem automatisch sinnvoll vorbesetzt. Detaillierte Informationen zu dem eben beschriebenen findet man in Kapitel 3.

Falls das CSSA-Programm fehlerfrei war, wird aus dem vom CSSA-Compiler generierten SIMULA-Programm und dem in source-Form vorliegenden Laufzeitsystem sowie evtl. vorhandenen externen Routinen nach der Anwendung des SIMULA-Compilers und Binders ein (einziges) ausführbares Maschinenprogramm erzeugt, das das **interaktive CSSA-System** darstellt. Es enthält die übrigen wesentlichen Systemkomponenten:

- Das **Simulationssystem** (——> 5.3) simuliert einerseits die Hardwarekomponenten einer Mehrrechnerkonfiguration aus Prozessoren und Bussen mit bestimmten wählbaren Charakteristika, andererseits werden die wesentlichen Aspekte eines verteilten Betriebssystems und lokaler unterbrechungsgesteuerter multiprogramming-Betriebssysteme simuliert. Statistiken über die Auslastung der simulierten Hardwarekomponenten können erzeugt werden und die wesentlichen Aktionen des Betriebssystems können protokolliert werden.

- Die **virtuellen Stackmaschinen** laufen auf den simulierten Prozessoren und stellen einfache Operationen zur Verfügung, die von den mit dem CSSA-Compiler aus den scripts generierten Programmoduln (Teilprogramme in Form von SIMULA-classes) benutzt werden.

- Der **CSSA-Interpreter** erkennt einerseits die interaktiv eingegebenen Kommandos zur Steuerung des Simulationssystems und des Debuggers und veranlaßt die notwendigen Aktionen,

kann andererseits aber auch eine Teilmenge der CSSA-Sprache interpretativ verarbeiten (——> 5.2). So kann der Benutzer etwa aus den übersetzten scripts Agenten gründen, mit ihnen kommunizieren und so die verteilten Berechnungen starten und beeinflussen.

- Der integrierte interaktive **debugger** (——> 5.4) erlaubt es, sehr detailliert und in symbolischer Form (durch Namen und Zeilennummern aus dem Quellprogramm) Informationen über den Zustand einzelner Agenten oder des gesamten Agentennetzes zu liefern. Dadurch wird die Entwicklung und der Test von CSSA-Programmen wesentlich erleichtert.

Die funktionalen Komponenten des CSSA-Systems sind allerdings nicht in jedem Fall physisch differenzierbar, sie bilden manchmal nur ab einer gewissen Abstraktionsstufe aus Benutzersicht (logisch) eigenständige Einheiten. Die Konzepte und die Realisierung dieser Komponenten ist in den Teilen A und E beschrieben.

Mit dem **interaktiven CSSA-System** kann in der eigentlichen CSSA-Sitzung die verteilte Berechnung initiiert und kontrolliert werden. Dies geschieht dadurch, daß vom interface aus Agenten gegründet werden, die auch bereits durch eine idle-Operation in der initialen Facette gewisse Aktionen ausführen können und ihrerseits etwa neue Agenten gründen oder Nachrichten versenden können. Nachdem einige Agenten gegründet wurden (Ablauf simulierter Zeit beachten!), können vom interface aus durch Versenden von Bekanntschaften in Nachrichten Operationen in den Agenten aktiviert werden, die dafür sorgen, daß die Agenten sich gegenseitig "kennenlernen" - der Graph dieser Bekanntschaftsrelation bildet das (dynamisch veränderbare) Agentennetz.

Vom interface aus kann das Agentennetz ebenfalls durch send-Befehle mit initialen Daten versehen werden. Schließlich wird mit dem run-Befehl (——> 5.3) simulierte Zeit am interface verbraucht und so die Berechnung des Agentennetzes tatsächlich gestartet. Von nun an kann auf unterschiedliche Weise vom interface aus der Ablauf der Berechnung überwacht und in die Berechnung eingegriffen werden: Etwa durch Gründen neuer Agenten, durch Kommunikation mit Agenten (——> 5.2) und durch gewaltsames stoppen und starten von Agenten (——> 5.3). Während der Berechnung können zu Zeitpunkten, an denen das interface die Kontrolle hat, debugging- und Simulationsanweisungen ausgeführt werden, die Informationen über den Zustand des simulierten Mehrrechnersystems, des Agentennetzes und einzelner Agenten liefern.

Ein Beispiel zum Ablauf einer CSSA-Sitzung befindet sich im Anhang, weitere Situngsprotokolle sind in Teil D abgedruckt.

3. Das Entwicklungssystem

Für Entwicklung und Test von CSSA-Anwendungen auf der Siemens 7760 der GMD steht ein **komfortables Entwicklungssystem** zur Verfügung, das alle Komponenten des CSSA-Systems integriert. Der Benutzer kann so ohne Kenntnisse der BS2000-Kommandosprache alle Arbeiten durchführen, die zur Entwicklung von CSSA-Anwendungen gehören, wie z.B.:

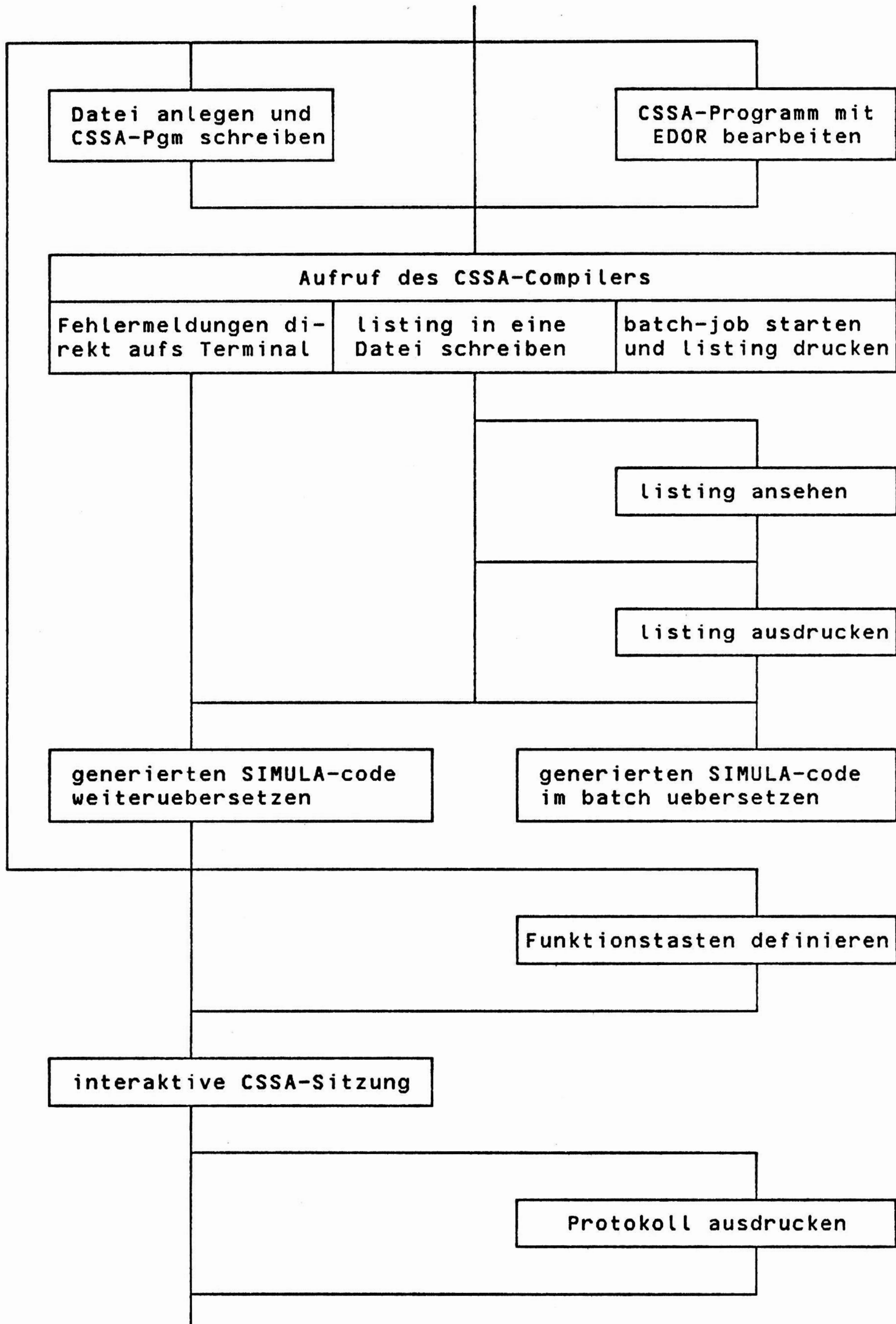
- Erstellen eines CSSA-Programms mit dem Editor
- Ändern oder Erweitern eines CSSA-Programms
- Übersetzen mit dem CSSA-Compiler
- Fehlermeldungen in einer Datei ansehen
- Programm-listing ausdrucken
- Weiterübersetzen des generierten Simula-codes
- Benutzung der übersetzten scripts in einer interaktiven Sitzung
- 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 Benutzer kann nun durch Eingeben eines einzelnen Buchstabens den gewünschten Dienst auswählen. Die bei der Benutzung anderer Compiler üblichen Fehlerquellen - wie Auswahl geeigneter Compiler-Parameter, verschiedene Druckformate , Dateiformate für Quellprogramm, 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.)


Datei anlegen und
CSSA-Pgm schreiben
CSSA-Programm mit
EDOR bearbeiten
Aufruf des CSSA-Compilers
Fehlermeldungen di-
rekt aufs Terminal
listing in eine
Datei schreiben
batch-job starten
und listing drucken
listing ansehen
listing ausdrucken
generierten SIMULA-code
weiteruebersetzen
generierten SIMULA-code
im batch uebersetzen
Funktionstasten definieren
interaktive CSSA-Sitzung
Protokoll ausdrucken

Das CSSA-Entwicklungssystem ist als BS2000-Kommandoprozedur geschrieben und ist an die Systemumgebung der Siemens 7760 Anlage der GMD angepaßt. Es wird auf Betriebssystemebene durch 'DO $SPENKE.P.CSSA' aufgerufen. Eine Kurzinformation steht in der Datei '$SPENKE.I.CSSA'. Im Laufe der Ausführung der Prozedur kann der Benutzer einen oder mehrere der folgenden Dienste aufrufen:

- **Anlegen einer Datei und Erstellen eines CSSA-Programms mit dem Dateibearbeitungssystem EDOR.**
  Es wird eine permanente SAM-Datei angelegt und der vom Benutzer angegebene Name verwendet. Das Gerüst für ein CSSA-Programm wird dabei vorgegeben und kann fast wie ein Formular ausgefüllt werden. Durch den EDOR-Befehl 'HR!H' wird die Datei geschlossen und die Kontrolle geht zurück an das Entwicklungssystem. Weitere Details über den Umgang mit EDOR können den einschlägigen Handbüchern entnommen werden.

- **Editieren eines CSSA-Programmes mit EDOR**
  Der Benutzer muß den Namen einer SAM- oder ISAM-Datei angeben, die das CSSA-Programm enthält. Die Datei muß nicht vom CSSA-Entwicklungssystem angelegt worden sein. Sie wird **virtuell** geöffnet. Änderungen schlagen daher erst nach Beendigung der EDOR-Sitzung mit 'HR!H' durch.

- **Übersetzung eines CSSA-Programms nach SIMULA**
  Aus dem folgenden Menü kann der Benutzer auswählen, in welcher Form er den CSSA-Compiler aufrufen möchte:

```
******************************************************
* - COMPILER-LISTING DIREKT AUFS TERMINAL      ( 1 ) *
* - IN EINE DATEI ZUM ANSEHEN                  ( 2 ) *
* - MIT DRUCKAUFBEREITUNG FUER LASERDRUCKER    ( 3 ) *
* - BATCH-JOB ABSCHICKEN UND DRUCKEN           ( 4 ) *
* - MIT DRUCKAUFBEREITUNG FUER REMOTE-DRUCKER  ( 5 ) *
* - MIT DRUCKAUFBEREITUNG FUER DIN A4-FORMAT   ( 6 ) *
* - BATCH-JOB FUER DIN A4 STARTEN              ( 7 ) *
* - SITZUNG BEENDEN                            ( X ) *
******************************************************
MODE=_
```

  Bei der **Übersetzungsform 1** werden Compilerlisting und Fehlermeldungen während des Übersetzungsvorgangs Zeile für Zeile direkt aufs Terminal ausgegeben. Eine Zeile des listings ist dabei nur 79 Zeichen lang, so daß das listing nicht durch Zeilenumbrüche unübersichtlich wird. Die Markierung der Blockstruktur am rechten Rand ist daher nicht möglich. Eine cross-reference Liste wird nicht ausgegeben. Der Compiler wird mit den options 'STRUCT,TERM,SOURCE' aufgerufen (——> 4.3.1).
  Diese Form der Übersetzung ist besonders dann angemessen, wenn man erwartet, daß das CSSA-Programm noch viele Fehler enthält. Die Rechenzeit des Compilers kann so schon zur Analyse der Fehlermeldungen benutzt werden. Ein Ausdrucken des listings ist allerdings hier nicht möglich.
  Bei der **Übersetzungsform 2** wird das listing in die temporäre Datei 'CSSALIST.<Name der CSSA-source Datei>' geschrieben. Die Datei wird dabei neu angelegt bzw. überschrieben.

Temporäre Dateien werden jeden Abend gelöscht. Der Compiler wird mit den options 'STRUCT,RESWD=L,SOURCE,NOXREF' aufgerufen. Schlüsselwörter werden also in Kleinbuchstaben dargestellt. Eine cross-reference Liste wird nicht ausgegeben. Direkt auf den Bildschirm meldet der Compiler lediglich, ob das Programm Fehler enthielt oder nicht.
Diese Übersetzungsform ist vor allem geeignet für längere Programme, die voraussichtlich noch einige Fehler enthalten. Man bekommt nicht das gesamte listing auf den Schirm, sondern kann sich nach der Übersetzung gezielt die fehlerhaften Programmpassagen ansehen. Zeilen, die Schlüsselwörter enthalten, sind nicht dreifach dargestellt, da kein Fettdruck vorgesehen ist. Das listing kann später ausgedruckt werden. Ein schöneres listing ergibt sich allerdings bei Verwendung von Übersetzungsform 3,4 oder 5.
Bei **Übersetzungsform 3** wird eine spätere Ausgabe auf den Laserdrucker vorbereitet. Sie unterscheidet sich von der Übersetzungsform 2 nur durch andere Compiler-options. Es wird 'STRUCT,RESWD=A,SOURCE,XREF' angegeben. Schlüsselwörter werden also fett gedruckt (bzw. in der Datei 'CSSALIST.<Name der source-Datei>' dreifach untereinander dargestellt). Eine cross-reference Liste wird ausgegeben.
Bei der späteren Ausgabe auf den Laserdrucker kann dann noch zwischen 2 verschiedenen Druckbildern gewählt werden. Es stehen der normale Zeichensatz (A1,6F) und ein etwas engerer Zeichensatz (RC,RD) zur Verfügung. Mit dem Engdruck erhält man ein gut lesbares listing, dessen wesentliche Teile dennoch auf DIN A4-Format abgedruckt werden können (vgl. auch Übersetzungsform 6 und 7).
Bei **Übersetzungsform 4** wird ein batch-job gestartet, der das CSSA-Programm nach SIMULA übersetzt, das listing auf den Laserdrucker ausgibt (options wie bei Übersetzungsform 3) und, falls das Programm fehlerfrei war, den generierten SIMULA-code weiterübersetzt (s.u.). Mit dem Abschicken des batch-jobs ist die CSSA-Sitzung beendet. Informationen über den Zustand des batch-jobs gibt das BS2000-Kommando 'STATUS LIST'. Nach Beendigung des batch-jobs steht das Compiler-listing ebenfalls in der Datei 'CSSALIST.<Name der source-Datei>'. Dort kann nachgesehen werden, ob das CSSA-Programm Fehler enthielt. War das Programm fehlerfrei, so steht das load-module in der temporären (!) Datei 'CSSAGO.<Name der source-Datei>'. Soll es längerfristig gespeichert werden, so kann es in eine andere (permanente) Datei kopiert werden. Der Name muß aber in jedem Fall die Form 'CSSAGO.<Programmname> haben: <Programmname> muß vor Ausführung des Programms dem Entwicklungssystem als Name der CSSA-source-Datei genannt werden (s.u.). Das load-module kann daher nur unter der Benutzerkennung ausgeführt werden, unter der es abgespeichert ist!
Für den Fall, daß wider Erwarten der batch-job nicht ordnungsgemäß beendet worden ist, steht ein ausführliches Protokoll und die Meldungen des SIMULA-Compilers in der temporären Datei 'CSSAMESS.<Name der source-Datei>'. Der vom Entwicklungssystem generierte batch-job steht in der Datei 'TEMPJCL.<Name der source Datei>'.
Die Benutzung des CSSA-Compilers im batch empfiehlt sich vor allem bei größeren Programmen. Als Zeitbedarf für den batch-job sind per default 300 Sekunden angesetzt. Bei grö-

ßerem Zeitbedarf muß beim Aufruf des Entwicklungssystems für den vorbesetzten Parameter TIME ein größerer Wert angegeben werden. Beispiel: 'DO $SPENKE.P.CSSA,TIME=400'
Übersetzungsform 5 bereitet eine spätere Ausgabe auf den remote-Drucker Bertha-von-Suttner-Platz vor. Es werden die Compiler-options 'NOSTRUCT,RESWD=L,SOURCE,XREF' angegeben. Dadurch wird der Tatsache Rechnung getragen, daß der remote-Drucker keinen Fettdruck zur Verfügung stellt und anstelle des senkrechten Strichs ein 'ö' ausgibt. Ansonsten unterscheidet sich Übersetzungsform 5 nicht von 2 und 3.
Übersetzungsform 6 ist zu wählen, wenn eine spätere Ausgabe auf den Laserdrucker im Kleindruck geplant ist. Die Compiler-options werden wie bei Übersetzungsform 3 gesetzt. Allerdings werden statt 65 Zeilen pro Seite 86 Zeilen ausgegeben. Das listing und auch die Fehlermeldungen können vollständig im DIN A4-Format abgedruckt werden. Diese Ausgabeform ist daher besonders für die Programmdokumentation geeignet.
Bei **Übersetzungsform 7** wird ein batch-job gestartet (vgl. Übersetzungsform 4), der das listing im Kleindruck ausgibt.

Bei allen Übersetzungsformen kann der Compiler direkt in SIMULA geschriebenen externen code übernehmen (——> 4.8). Dem Entwicklungssystem muß in diesem Falle mit 'DO $SPENKE.P.CSSA,EXTERNAL=<Dateiname>' mitgeteilt werden, wo der externe code steht.

- **Analyse der Compilermeldungen am Bildschirm**
  Im Anschluß an die Übersetzungsformen 2,3 und 5 kann sich der Benutzer mit dem GMD-Programmprodukt SHOW das Compiler-listing ansehen.
  Die wichtigsten Kommandos für SHOW sind:

++	vorblättern bis zum Ende der Datei
--	zurückblättern an den Anfang der Datei
+	eine Seite vor
-	eine Seite zurück
+<Zeilenzahl>	vorblättern
-<Zeilenzahl>	zurückblättern
C<Spalte>	Datei erst ab Spalte <Spalte> zeigen (Wichtig, da das listing nicht in voller Breite am Terminal angezeigt werden kann.)
END	beenden von SHOW. Die Kontrolle geht zurück an das Entwicklungssystem.

  Als one-pass Compiler trennt der CSSA-Compiler listing und Fehlermeldungen, indem er sie in zwei verschiedene Dateien ausgibt. Vor dem Aufruf von SHOW werden die beiden Dateien daher vom Entwicklungssystem konkateniert.

- **Compiler-listing ausdrucken**
  Das Compiler-listing kann auf Wunsch auf den Laserdrucker oder den remote-Drucker S2M06DSC am Bertha-von-Suttner-Platz ausgegeben werden. Um einen eindeutigen Namen für die print-Datei generieren zu können, wird vom Benutzer ein Identifikations-character angefordert. Dadurch wird sichergestellt, daß kurz hintereinander mehrfach dasselbe CSSA-Programm übersetzt und ausdruckt werden kann. (Im BS2000 sind nämlich print-Dateien solange gesperrt, bis sie tatsächlich gedruckt worden sind.) Als Identifikations-

character kann eine Ziffer oder ein Buchstabe gewählt werden. Vor dem Ausdrucken werden die vom SIMULA-Laufzeitsystem ausgegebenen IBM-Druckervorschubzeichen in den SIEMENS-Standard übersetzt. Für den Laserdrucker wird noch eine weitere Aufbereitung vorgenommen: Fettdruck wird hier nicht durch mehrfaches Drucken, sondern durch einen speziellen Zeichenvorrat erreicht.
Bei der Ausgabe auf den Laserdrucker kann zwischen 3 verschiedenen Ausgabeformen gewählt werden, wodurch sich unterschiedliche Druckbilder ergeben: Normaldruck (L), Engdruck (E) und Kleindruck für DIN A4-Format (4). Die Ausgabeform '4' sollte stets in Verbindung mit der Übersetzungsform 6 gewählt werden, bei der 86 Zeilen pro Seite ausgegeben werden (vgl. Testprogramme in Teil D). Listings im Engdruck (siehe Beispiel-script im Anhang) können ebenfalls auf DIN A4-Format zurechtgeschnitten werden, wobei allerdings einige (unwesentliche) Informationen verlorengehen.

- **Weiterübersetzen des generierten SIMULA-codes**
  Der vom Compiler generierte code wird zunächst entsprechend seiner Numerierung **sortiert.** Anschließend werden die Routinen des CSSA-Laufzeitsystems und das Simulationssystem, sowie vom Benutzer spezifizierter externer SIMULA-code hinzukopiert. SIMULA-Compiler und linkage-editor erzeugen daraus ein ausführbares Programm. Es steht in der temporären Datei 'CSSAGO.<Name der source-Datei>', kann aber auf eine permanente Datei kopiert werden (s.o.). Für das Übersetzen des SIMULA-codes kann ebenfalls auf Wunsch ein batch-job gestartet werden. Dabei sind die gleichen Randbedingungen zu beachten, wie bei der Ausführung des **gesamten** Übersetzungsvorgangs im batch (s.o.).
  Die Meldungen des SIMULA-Compilers werden in die Datei 'CSSAMESS.<Name der source-Datei>' geschrieben. Ob der SIMULA-Compiler Fehler entdeckt hat, kann man bei der interaktiven Übersetzungsform am condition-code erkennen (0=fehlerfrei, sonst fehlerhaft). Der sortierte code wird nach dem Aufruf des SIMULA-Compilers sofort wieder gelöscht. Wer sich den generierten code ansehen (oder ausdrucken) will, muß die interaktive Übersetzung wählen und nach der Meldung 'Der generierte code ist sortiert worden und wird jetzt vom CSSA-Compiler übersetzt' mit der K2-Taste unterbrechen. Der generierte code steht dann in der Datei 'CSSACODE.<Name der source-Datei>'.

- **Start einer interaktiven CSSA-Sitzung**
  Während der interaktiven Sitzung interpretiert der interface-Agent die vom Benutzer eingegebenen Kommandos (——> 5.). Dabei können die vom Compiler übersetzten Scripts verwendet werden.
  Auf Wunsch werden speziell für CSSA geeignete **Funktionstasten** definiert:

```
P1:  'DISPLAY' <EINGABETASTE>
P2:  'STATUS' <EINGABETASTE>
P3:  'SYSSTATUS' <EINGABETASTE>
P4:  'VAR AGENT : '
P5:  'OPER : '
P6:  'PORT : '
```

P7: 'SEND '
P16: ';RUN' <EINGABETASTE>
P17: ';RUN' <EINGABETASTE> + Schirm loeschen

Die Funktionstastenbelegung bleibt auch nach dem Ende der Sitzung erhalten und braucht daher für mehrere aufeinanderfolgende Sitzungen nicht mehrfach definiert zu werden. Während der Ausführung eines CSSA-Programmes arbeitet das SIMULA-Laufzeitsystem mit 512K Speicherplatz. Reicht das nicht aus, so muß beim Aufruf des Entwicklungssystems mit dem SIZE-Parameter mehr Platz angefordert werden (ein Vielfaches von 4K). Beispiel: 'DO $SPENKE.P.CSSA,SIZE=768K'. Sollen zunächst Befehle für den interface-Agenten anstatt vom Terminal aus einer Datei (SAM-Datei !!) gelesen werden (——> 5.1), so muß 'DO $SPENKE.P.CSSA,INIT=<Datei>' angegeben werden. Es kann auch eine eigene Konfiguration für das Simulationssystem (——> 5.3) aus einer (SAM-) Datei gelesen werden. Diese Datei wird durch 'DO $SPENKE.P.CSSA,CONFIG=<Datei>' angegeben.

- **Drucken eines Sitzungsprotokolls**
  Falls in der interaktiven CSSA-Sitzung ein Protokoll erzeugt wurde (——> 5.4), so kann es auf Wunsch auf dem Laserdrucker oder dem remote-Drucker ausgegeben werden. Um einen eindeutigen Namen für die print-Datei erzeugen zu können, wird vom Benutzer wieder ein Identifikations-character angefordert. Nach dem Ende der CSSA-Sitzung steht das Protokoll noch in der temporären Datei 'PROT.<Name der source-Datei><IDCHAR>'.
  Bei der Ausgabe auf dem Laserdrucker kann wiederum zwischen verschiedenen Zeichensätzen gewählt werden. Protokolle können problemlos im DIN A4-Format abgedruckt werden, wenn Engdruck (vgl. Protokolle im Anhang) oder Klein- und Engdruck gewählt werden.

Im Normalfall werden alle Meldungen des BS2000 vom Entwicklungssystem unterdrückt. Der Benutzer wird nur mit den ihm verständlichen Menüs und anderen CSSA-spezifischen Meldungen konfrontiert. Ist jedoch eine ausführliche Protokollierung aller Kommandos gewünscht - z.B. weil ein unvorhergesehener Fehler aufgetreten ist - so muß der Aufruf 'DO $SPENKE.P.CSSA,MSG=A' lauten.

Achtung: Andere Eingaben als die in den Menüs jeweils geforderten führen zum Abbruch der CSSA-Sitzung. Das Entwicklungssystem kann stets durch Eingabe von 'X' verlassen werden, auch wenn das nicht explizit im Menü angegeben ist.

4. Der Compiler

In diesem Kapitel werden Hinweise zur **Benutzung** des BMS-CSSA-Compilers gegeben. Sie umfassen das Eingabeformat, die Steuerung des Compilers und die Beschreibung seiner Ausgaben - wie source-listing, cross-reference Liste (XREF), Fehlermeldungen etc. Die Eingabesprache CSSA, die im wesentlichen implementierungs-unabhängig ist, wird dagegen in [BMS-B] dargestellt.

4.1 Ein- und Ausgabeverhalten des Compilers

Die **Eingabe** des BMS-CSSA-Compilers besteht hauptsächlich aus einem CSSA-Programm, dessen Eingabeformat in 4.2 beschrieben wird; zusätzlich können in den Programmtext noch **Steueranweisungen** an den Compiler eingestreut werden; sie betreffen meist die Steuerung der Ausgabe und werden in 4.3.2 erläutert. Außerdem können bei Aufruf des Compilers verschiedene **options** angeben werden, die gewisse Voreinstellungen bezüglich der Ausgabe erwirken oder Begrenzungen für die Übersetzung darstellen; sie werden in 4.3.1 beschrieben.

Die **Ausgabe** des Compilers besteht aus dem (aufbereiteten) **CSSA-source-listing** mit **cross-reference Liste**, den **Fehlermeldungen** mit Korrekturannahmen (sowie evtl. Ableitungsbäume zu ausgewählten Zeilen ——> 4.6) und dem **generierten code**. Die verschiedenen Ausgaben können unterdrückt bzw. modifiziert werden (z.B. zur Anpassung an das Ausgabemedium Terminal bzw. Drucker). Dies geschieht durch Angabe von options oder dynamisch mit Hilfe von Steueranweisungen (siehe 4.3). Das Layout des source-listing wird in 4.4 erläutert, die cross-reference Liste in 4.5, die Fehlermeldungen und Korrekturannahmen in 4.7.

Der **generierte code** ist reentrant und für eine virtuelle stack-Maschine konzipiert (auf niedriger Ebene - jedoch hardware-unabhängig) und wird in SIMULA-Syntax erzeugt, damit er zusammen mit dem Simulationssystem vom SIMULA-Compiler weiter-übersetzt werden kann. Das Einbringen von - in SIMULA geschriebenem - code für **externe Funktionen und Prozeduren** geschieht bei dem zweiten Übersetzungsschritt und wird in 4.8 beschrieben.

Übersicht über Ein- und Ausgabe des BMS-CSSA-Compilers:

Da der Compiler in das Entwicklungssystem integriert ist, braucht der Benutzer diesen Abschnitt nicht zu beachten. Lediglich Anwender, die den Compiler ohne Entwicklungssystem aufrufen wollen, sollen hier über die Ein- und Ausgabefiles informiert werden.

Eingabe:

- Compiler-options
- CSSA-Quellcode und Compiler-Steuerkarten auf dem Standard-Eingabefile
- Code für externe Funktionen und Prozeduren werden nicht dem CSSA-Compiler übergeben, sondern erst in einem späteren Schritt zu dem übersetzten code hinzugebunden.

Ausgabe:

- CSSA-source-listing und XREF auf dem Standard-Ausgabefile
- Fehlermeldungen, Korrekturannahmen (und evtl. Ableitungsbäume, interner trace und der Inhalt interner stacks) auf dem file "PROTOCOL"
- generierter code auf dem file "GENCODE"

Falls die TERM-Option gesetzt wurde, werden die Fehlermeldungen zusammen mit dem source-listing auf dem Standard-Ausgabefile ausgegeben, so daß Meldungen direkt unter der fehlerhaften Zeile im listing stehen.

Bei Benutzung des Entwicklungssystems werden die Dateien mit dem source-listing und den Fehlermeldungen automatisch konkateniert.

4.2 Eingabeformat des Quellprogramms

Das CSSA-Programm, das der Compiler als Eingabe erhalten soll, muß in Form einer SAM- oder ISAM-Datei vorliegen. Die Sätze dürfen höchstens 80 Zeichen lang sein.

Das **CSSA-Programm** kann **formatfrei** in die Spalten 1 bis 72 geschrieben werden. Das Zeilenende wirkt dabei wie ein Leerzeichen als Begrenzer, daher dürfen syntaktische Einheiten (insbesondere Kommentare und string-Konstanten) nicht über das Zeilenende hinausgehehen (——> B 2.2). Die **Spalten 73 bis 80** können zur Numerierung der Sätze benutzt werden. Sollten die Spalten 73-80 eines Satzes beschrieben sein, jedoch die Spalte 73 keine Ziffer enthalten (z.B. weil das CSSA-Programm versehentlich über Spalte 72 hinausgeschrieben wurde), so erhält der Benutzer zu der entsprechenden Zeile eine Warnung:

- WARNING IN LINE <n>: TEXT IN COL 73-80: '<text>' IGNORED.

Sätze mit **Steueranweisungen** zeichnen sich durch ein Prozentzeichen in Spalte eins aus, im Anschluß folgt eine der in 4.3.2 genannten Steueranweisungen. Das CSSA-Programm kann erst im nächsten Satz fortgesetzt werden.

Für die Eingabe von CSSA-Programmen und Steueranweisungen könne beliebig **Groß- und Kleinbuchstaben** verwendet werden. Relevant ist die Schreibweise jedoch nur in Textkonstanten. Im sourcelisting erscheint die gewählte Schreibweise nur innerhalb von Textkonstanten und Kommentaren. Über das Erscheinungsbild der Ausgabe gibt 4.4 Auskunft.

4.3 Options und Steueranweisungen

Das Verhalten des Compilers, insbesondere seine Ausgabe, kann je nach Verwendungszweck gesteuert werden. Dies kann in Form von **options** geschehen, die **einmalig bei Aufruf** des Compilers angegeben werden, oder **dynamisch** mit Hilfe von **Steueranweisungen**, die in den Programmtext eingestreut werden.

4.3.1 Options

Bei Benutzung des Entwicklungssystems werden dem Compiler je nach Aufrufmodus **automatisch geeignete options** übergeben. Deshalb braucht sich der Benutzer des Entwicklungssystems nicht um die Angabe der options zu kümmern. Zum besseren Verständnis werden hier jedoch alle options erläutert. Am Ende dieses Kapitels werden Hinweise zur Übergabe der options gegeben, die nur für diejenigen Anwender interessant sind, die den Compiler ohne Entwicklungssystem aufrufen wollen.

Die für die aktuelle Compilerversion gültigen **default-options** sowie die im aktuellen Compilerlauf **aktiven options** werden am Anfang eines jeden CSSA-listings ausgegeben. Dazwischen erscheinen gegebenenfalls Fehlermeldungen zu falsch eingebenen options, die dann ignoriert werden.

Liste aller options für den BMS-CSSA-Compiler:

- **NOOBJECT**
  Keine Generierung von code.

- **OBJECT** (default)
  Ausgabe von generiertem code.
  (Nur solange das analysierte Programm fehlerfrei ist.)

- **SOURCE** (default bei NOTERM)
  Das eingegebene Quellprogramm wird protokolliert.
  (Dynamisch änderbar durch Steueranweisungen ——> 4.3.2.)

- **NOSOURCE** (default bei TERM)
  Das eingegebene Quellprogramm wird nicht protokolliert (dynamisch änderbar durch Steueranweisungen ——> 4.3.2); Zeilen zu denen Fehler gemeldet werden, erscheinen dann jedoch in der Liste der Fehlermeldungen.

- **STRUCT** (nur sinnvoll in Verbindung mit SOURCE)
  Klammerstrukturen im Programm werden automatisch durch senkrechte Linien im source-listing hervorgehoben (——> 4.4).

- **NOSTRUCT** (default):
  Keine Erzeugung von senkrechten Linien im source-listing.

- **RESWD** (Nur relevant in Verbindung mit SOURCE)
  Zur Hervorhebung von Schlüsselworten im source-listing:
  RESWD=B: Durch Fettdruck [boldface],
  RESWD=L: Durch Kleinbuchstaben [lower case],

RESWD=A: Durch Fettdruck und Kleinbuchstaben [all],
RESWD=N: Keine Hervorhebung vom Schlüsselwörtern [none].
Default bei NOTERM: RESWD=A, default bei TERM: RESWD=L.

- **XREF** (default bei NOTERM)
  Eine cross-reference Liste wird erstellt. (——> 4.5)

- **NOXREF** (default bei TERM)
  Es wird keine cross-reference Liste erstellt.

- **NOTERM** (default)
  Für die Ausgabe auf dem Drucker. Die Ausgabe wird allgemein dem größeren Platz auf dem Papier angepaßt: Es werden Seitenüberschriften (mit Seitenzahlen) und eine Tabulatorzeile pro Seite ausgegeben, fehlerhafte Zeilen werden mit einem Pfeil ('===>') gekennzeichnet, die Markierung der Blockverschachtelung und der Inhalt von Spalte 73 bis 80 werden am rechten Rand ausgegeben. Der Umbruch des Textes von Fehlermeldungen wird der größeren Zeilenlänge angepaßt.

- **TERM**
  Für die Ausgabe am Bildschirm: Es werden folgende options gesetzt: NOSOURCE, NOXREF, RESWD=L; diese options können abgeändert werden, wenn **hinter** TERM eine andere option angegeben wird (Z.B: TERM-option mit Ausgabe des source: TERM,SOURCE). Außerdem wird die Ausgabe den kürzeren Zeilen des Terminals angepaßt: Es wird nur Spalte 1 bis 72 des Quellprogramms (zwischen senkrechten Strichen) ausgegeben, Ableitungsbäume werden enger dargestellt (siehe Steueranweisung %DERIV, 4.3.2) und Fehlermeldungen werden ab Spalte 70 umgebrochen. Fehlermeldungen und source-listing werden auf dem gleichen file ausgegeben, so daß Meldungen direkt unter der zugehörigen Zeile erscheinen.

- **LINECOUNT=n** oder **LC=n** (nur sinnvoll mit NOTERM und SOURCE)
  Anzahl der Zeilen, die pro Seite gedruckt werden. n darf zwischen 1 und 99 liegen; default ist LINECOUNT=65.

- **MAXPAGES=n** oder **MAXP=n**
  Zur Begrenzung der Seitenzahl des source-listings (nur sinnvoll in Verbindung mit NOTERM und SOURCE). Bei Überschreitung wird die Ausgabe des source-listings unterdrückt, der Übersetzungsvorgang und die Ausgabe der Fehlermeldungen werden nicht abgebrochen (fehlerhafte Zeilen werden von nun ab bei den Fehlermeldungen ausgegeben). MAXPAGES hat keinen Einfluß auf die Länge der Liste der Fehlermeldungen. n darf zwischen 0 und 9999 liegen; default: MAXPAGES=99.

- **MAXERROR=n** oder **MAXE=n**
  Der Compiliervorgang wird abgebrochen, wenn mehr als n Fehler gemeldet werden. Das listing wird (hinter einer deutlichen Trennlinie) fortgesetzt. n darf zwischen 0 und 9999 liegen; default: MAXERROR=99.

- **MAXDERIV=n** oder **MAXD=n**
  Zu maximal n Zeilen wird ein Ableitungsbaum gedruckt (siehe Steueranweisung %DERIV, 4.3.2). Bei Überschreiten wird

lediglich kein Ableitungsbaum mehr gedruckt; listing und Übersetzung werden fortgesetzt. n darf zwischen 0 und 9999 liegen; default ist MAXDERIV=99.

Die folgenden options sollen nur für Testzwecke benutzt werden.

- **CHECK** (default)
  Der Compiler wendet Fehlerbehandlung mit error-recovery an.

- **NOCHECK**
  Compilerlauf ohne Fehlerbehandlung. Nur für fehlerfreie CSSA-Programme geeignet.

- **NOTEST** (default) Keine Testausgaben.

- **TEST**
  Ausgabe über verbrauchten Speicherplatz beim Compilieren etc. (nur für Implementatoren).

Die foldenden Informationen sind nur für Anwender von Bedeutung, die nicht das Entwicklungssystem benutzen. Das Entwicklungssystem übergibt jeweils automatisch geeignete options.

Options müssen als erste Karte im source-text hinter '%PARM=' erscheinen, sie werden durch Kommata voneinander getrennt und als letzte option muß '*' codiert werden. Alle options müssen in Großbuchstaben geschrieben werden.

Beispiel:

- %PARM=STRUCT,MAXERROR=20,*

Bei der Angabe der options sollte man bedenken, daß die ersten options an das SIMULA-Laufzeitsystem übergeben werden, bis eine für das Laufzeitsystem unverständliche option erreicht wird, da der BMS-CSSA-Compiler ein SIMULA-Programm ist. Das Entwicklungssystem übergibt an das SIMULA-Laufzeitsystem folgende options:

- %PARM=NOTEST,SIZE=(,,512k),...

Anstelle von '...' werden CSSA-Compiler-options angegeben. (Vorsicht: 'NOTEST' ist SIMULA- und CSSA-option!)

4.3.2 Steueranweisungen

Steueranweisungen dienen zur **dynamischen** Steuerung der Ausgabe. Sie werden in das Quellprogramm eingestreut. Eine Steueranweisung an den Compiler beginnt mit einem Prozentzeichen ('%') in Spalte eins eines Satzes. Darauf folgt die Steueranweisung und evtl. ein Parameter. Der Rest des Satzes wird vom Compiler ignoriert und darf also nicht für eine weitere Steueranweisung oder die Fortsetzung des CSSA-Programms verwendet werden.

Fehlerhafte Steueranweisungen bewirken eine Fehlermeldung (in der Liste der Syntaxfehler) und werden ansonsten ignoriert.

Folgende Steueranweisungen können angegeben werden:

- **%SOURCE**
  Ab dieser Anweisung soll das source-listing des Programms ausgeben werden.
  (Voreinstellung durch option SOURCE, NOSOURCE bzw. TERM)

- **%NOSOURCE**
  Ab dieser Anweisung wird das source-listing unterdrückt. Zeilen zu denen Fehler gemeldet werden, erscheinen jedoch in der Liste der Fehlermeldungen.

- **%SKIP** [n]
  Es werden n Leerzeilen im listing erzeugt. Falls die Angabe von n fehlt, wird **eine** Leerzeile erzeugt.

- **%PAGE**
  Das listing wird auf der nächsten Seite fortgesetzt.

- **%TITLE** <text>
  Das listing wird auf der nächsten Seite fortgesetzt, und in der Überschrift erscheint der Text, der hinter %TITLE angegeben ist (Spalte 8 bis 55). Der Text erscheint auch solange auf den folgenden Seiten (bei expliziten Seitenvorschub durch %PAGE und bei impliziten durch Erreichen der maximalen Zeilenzahl pro Seite), bis der Titeltext durch eine neue %TITLE-Anweisung geändert wird. (Insbesondere ist auch ein Text, der nur aus Leerzeichen besteht, zulässig.)

- **%DERIV=1**
  Ab dieser Anweisung Ableitungsbäume ausgeben. (——> 4.6)

- **%DERIV=2**
  Ableitungsbäume mit Terminalsymbolen ausgeben.

- **%DERIV=3**
  Ableitungsbäume mit fettgedruckten Terminalsymbolen ausgeben. (Nur für Ausgabe auf einem Zeilendrucker geeignet.)

- **%DERIV=0**
  Ab dieser Anweisung keinen Ableitungsbaum mehr ausgeben.
  (Voreinstellung: %DERIV=0)

- **%RESWD=B/L/A/N**
  Wie option RESWD, siehe 4.3.1.

Die folgenden Steueranweisungen sind nur für Testzwecke vorgesehen:

- **%RECOVERY / %NORECOVERY**
  Zum Ein- und Ausschalten der Protokollierung des recovery-stacks.

- **%TRACE / %NOTRACE**
  Zum Ein- und Ausschalten von SIMULA-trace.

Die Steueranweisungen %SKIP, %PAGE und %TITLE erscheinen nicht im listing; die übrigen werden protokolliert, wenn SOURCE eingeschaltet ist; insbesondere wird %NOSOURCE nie und %SOURCE immer protokolliert.

4.4 Layout des source-listing

In diesem Kapitel soll der Aufbau des source-listing erläutert werden, das der BMS-CSSA-Compiler erzeugt. Manche Angaben beziehen sich nur auf die Ausgabe auf dem Drucker mit der NOTERM-option; welche das sind, entnehme man der Beschreibung in 4.3.1.

4.4.1 Beispiel

Hier ein Beispiel für ein listing, wie es der BMS-CSSA-Compiler erzeugt. Aus drucktechnischen Gründen mußte das Beispiel 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
```

4.4.2 Erläuterung

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

Auf Seite eins folgt eine Liste der **default-options** der aktuellen Compilerversion, sowie eine Liste der für diesen Übersetzungslauf **aktiven options.** Zwischen den beiden Listen erscheinen gegebenenfalls Fehlermeldungen zu falsch eingegebenen options. (options ——> 4.3.1)

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 Strichen ausgegeben. Insbesondere wird auf keinen Fall die Anordnung der Zeichen innerhalb der Sätze verändert. Die Fehlermeldungen erscheinen nicht zwischen den Zeilen, sondern im Anschluß an das source-listing (Fehlermeldungen ——> 4.7). Folgende Änderungen werden vorgenommen:

- **Schlüsselworte** werden GROSS (bei RESWD=N), klein (bei RESWD=L), **FETT** (bei RESWD=B) bzw. **klein und fett** (bei RESWD=A) ausgegeben - nur abhängig von der gesetzten option, unbeeinflußt von der (beliebig wählbaren) Schreibweise in der Eingabe (——> 4.2).

- Alle übrigen Buchstaben außerhalb von Kommentaren und Textkonstanten (i.w. in Identifiern) werden - ebenfalls unabhängig von der Schreibweise in der Eingabe - in Großbuchstaben ausgegeben.

- **Fehlerhafte Symbole** werden (durchgezogen) unterstrichen, überlesene Symbole werden gestrichelt unterstrichen (Fehlermeldungen ——> 4.7).

- Wenn die **STRUCT-option** gesetzt ist, werden in den sourcetext senkrechte Linien eingetragen, die - bei geeigneter Anordnung der Eingabe - die Klammerstrukturen optisch unterstützen (siehe unten).

- Die **Steueranweisungen** %PAGE, %TITLE und %SKIP werden nicht protokolliert.

Der Inhalt der **Spalten 73-80** wird - getrennt vom Text der Spalten 1-72 - ganz rechts außen ausgeben, da dieser nicht analysiert wird. **Kommentarzeilen** (Zeilen, die mit Dollar ('$') beginnen) werden durch ein Dollar rechts neben dem Text gekennzeichnet. Links neben den Zeilen erscheint eine **Zeilennumerierung,** die mit der Satzzahl innerhalb der Datei übereinstimmt, da Zeilen, deren listing durch %NOSOURCE unterdrückt werden und Zeilen mit Steueranweisungen, die nicht ausgegeben werden (%SKIP, %PAGE und %TITLE ——> 4.3.2), trotzdem mitgezählt werden. Zeilen, zu denen Fehler gemeldet werden und Zeilen, in denen sich überlesene Symbole (gestrichelt unterstrichen) befinden, werden mit einem **Pfeil** am linken Rand gekennzeichnet (Fehlermeldungen ——> 4.7).

Unter der Überschrift **BLOCKNESTING** wird rechts neben dem Quellprogramm noch eine Information über die Blockstruktur des Programmes ausgegeben. Diese Information kann insbesondere im Fehlerfall darüber Auskunft geben, wie der Compiler die Blockstruktur "verstanden" hat. Achtung: Hier werden nicht die für die Sichtbarkeitsregeln relevanten scopes gekennzeichnet, sondern syntaktische Strukturen, die durch eindeutige Schlüsselworte geklammert werden:

- script ... endscript
- facet ... endfacet
- operation ... endoperation
- procedure ... endprocedure
- function ... endfunction
- record ... endrecord
- receive ... endreceive
- find ... endfind
- begin ... end
- if ... endif
- loop ... endloop
- select ... endselect

Die versetzte senkrechte Linie neben dem Quellcode vermittelt einen anschaulichen Eindruck von der syntaktischen Struktur des Programmes: Sie ist entsprechend der Schachtelungstiefe nach rechts eingerückt. Der Anfang eines Blocks ist durch +n und das Ende durch -n gekennzeichnet (n ist die laufende Nummer des Blocks). Es läßt sich so leicht zu jeder syntaktischen Klammer die korrespondierende Klammer finden. Die Rückkehr auf das Niveau von Block n wird durch *n markiert. Es läßt sich daher auf einen Blick zu jeder Zeile des Quellcodes feststellen, welches die direkt umgebende syntaktische Klammer ist, indem man die senkrechte Linie bis zum nächsten +n oder *n nach oben verfolgt.

Wenn die **STRUCT-option** gesetzt ist, dann wird die Blockstruktur auch durch senkrechte Linien im source-listing hervorgehoben. Dazu muß der Programmierer dafür sorgen, daß

- öffnendes und schließendes Schlüsselwort einer Klammerstruktur (s. oben) in der gleichen Spalte beginnen, und daß
- der Text zwischen den (klammernden) Schlüsselworten um mindestens ein Zeichen eingerückt wird, das heißt jeweils weiter rechts als die zugehörigen klammernden Schlüsselworte beginnt.

Wenn die STRUCT-option eingeschaltet ist, wird in Zeilen zwischen klammernden Schlüsselworten (siehe Liste) auf der Höhe des ersten Buchstabens des öffnenden Schlüsselwortes eine senkrechter Linie generiert. Die senkrechte Linie wird in Spalte n nur dann erzeugt, wenn die Spalten eins bis n der entsprechenden Zeile unbeschrieben sind.

Hinter dem source-listing erscheinen einige **Statistiken** über den Übersetzungslauf. Dabei wird gemeldet:

- Die Anzahl der entdeckten **Fehler**
- Das Datum, zu dem der benutzte BMS-CSSA-Compiler generiert wurde (release)
- Datum und Uhrzeit, zu dem der Übersetzungsvorgang beendet wurde
- Der **returncode** (7 bei fehlerhaften, 0 bei fehlerfreien Programmen)
- Die reine **Rechenzeit** (CPU-time) und die reale **Ausführungszeit** (execution-time) für den Übersetzungslauf
- Die Anzahl der eingelesenen Zeilen, der ausgegebenen Sätze mit generiertem code und die Anzahl der Symbole (tokens) des Quellprogramms.

4.5 Cross-reference Liste

Zu jedem CSSA-Programm wird eine attributierte cross-reference Liste ausgedruckt, wenn die option XREF angegeben wird (options ——> 4.3.1). In dieser Tabelle werden alle Bezeichner alphabetisch sortiert ausgegeben mit der Angabe der Typklasse und evtl. des Daten- und Subtypes (Typkonzept ——> B 4.) sowie eine Liste der Zeilennummern, in denen der Bezeichner verwendet wurde. Die erste Zeilennummer ist immer die Deklarationsstelle des Bezeichners, da in CSSA jeder Bezeichner textuell vor seiner Verwendung deklariert sein muß. Tritt ein Bezeichner in einer Zeile mehrfach auf, so wird dies mit '+' hinter der Zeilennummer gekennzeichnet. Gleichnamige Bezeichner, die in verschiedenen Blöcken deklariert wurden, werden getrennt aufgeführt (siehe Bezeichner PHILOSOPH im Beispiel)

Die cross-reference Liste zu dem Programm aus 4.4 sieht folgendermaßen aus:

```
                  *** 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
```

4.6 Syntaxbäume

Mit Hilfe der Steueranweisung %DERIV=n kann zu ausgewählten Zeilen eines CSSA-Programmes ein Ableitungsbaum ausdruckt werden. Man erhält so detaillierte Information, wie der Compiler die syntaktische Struktur des Quellprogramms aufgefaßt hat.

Wird im Quellprogramm die Steueranweisung %DERIV=1 angegeben, so wird zu den folgenden Zeilen ein Ableitungsbaum in preorder bezüglich der in [BMS-B] abgedruckten erweiterten BNF-Grammatik generiert, der nur Nichtterminale enthält. Schreibt man %DERIV=2, so werden zusätzlich die Terminalsymbole des Quellprogramms ausgegeben; bei %DERIV=3 in Fettdruck. Der Baum wird nicht mehr gedruckt, wenn eine Steueranweisung %DERIV=0 erreicht wird. (Der Ableitungsbaum wird auch nicht mehr gedruckt, wenn die Grenze, die durch die option MAXDERIV gesetzt wurde, erreicht ist.) Der Ableitungsbaum wird in der Liste der Fehlermeldungen im Anschluß an das source-listing ausgegeben. (Options und Steueranweisungen ——> 4.3)

Hier als Beispiel der erste Teil des mit %DERIV=3 erzeugten Ableitungsbaumes zu dem in 4.4 abgedruckten Programm:

```
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

LINE   5
                    PUBLIC
                    :
                    <IDENTIFIER_LIST>
                        PHILOSOPH
                    ;
```

```
LINE   6
                    <LOCAL_DECL_LIST>
                    <OPERATION_DECL>
                        OPERATION
                        PHILOSOPH
                        IS

LINE   7
                        <LOCAL_DECL_LIST>
                            <DECLARATION>
                                <OPER_DECL>
                                    OPER
                                    :
                                    <IDENTIFIER_LIST>
                                        PICKUP
                                        ,
                                        PUTDOWN
                            ;

LINE   8
                            <DECLARATION>
                                <PORT_DECL>
                                    PORT
                                    :
                                    <IDENTIFIER_LIST>
                                        FIRST
                                        ,
                                        SECOND
                            ;

LINE   9
                        _____________________

LINE  10
                        <STATEMENT_LIST>
                            <STATEMENT>
                                <SEND_STATEMENT>
                                    SEND
                                    <OPER_DENOTATION>
                                        <DENOTATION>
                                            PICKUP
                                    <MESSAGE>
                                    TO
                                    <AGENT_EXPRESSION>
                                        <FACTOR>
                                            <DENOTATION>
                                                RFORK

LINE  11
                                    REPLY
                                    TO
                                    FIRST
                            ;
```

4.7 Fehlermeldungen

Der BMS-CSSA-Compiler versucht, **alle Fehler** in einem Compilerlauf zu finden und möglichst keine Folgefehler ("spurious errors") zu erzeugen. Die dabei zur Anwendung kommende systematische und automatische Syntaxfehlerbehandlungsmethode **(BMS error-recovery)** wird an anderer Stelle beschrieben, hier soll lediglich auf die Fehlerbehandlung des Compilers aus Benutzersicht eingegangen werden; die Beschreibung der Methode beschränkt sich daher auf die für den **Anwender** relevanten Eigenschaften.

Neben der Beschreibung der Fehlermeldungsformen und einer **Auflistung aller Fehlermeldungen** mit Hinweisen auf mögliche **Fehlerursachen** werden in diesem Kapitel auch einige kurze Bemerkungen zur generellen **Problematik** der Fehlerbehandlung und zu dem im BMS-CSSA-Compiler verwendeten **Prinzip** gegeben ; diese Hinweise sollen zu einem besseren Verständnis der Fehlerbehandlungsmethode und der Fehlermeldungen beitragen.

Die **Laufzeitfehler** des CSSA-Systems werden in Kapitel 5.5 behandelt.

4.7.1 Fehlerdiagnose

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

- Markierung der Fehlerstelle im **source-listing:** 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; die entsprechende Zeile wird ebenfalls am linken Rand markiert. Oft genügt daher ein Blick auf das source-listing, um den Fehler zu erkennen.

- Eine **genaue Diagnose** der Fehler erfolgt im Anschluß an das source-listing in den "parsing-diagnostics". Hier werden neben detaillierteren Fehlerhinweisen, die weiter unten erläutert werden und 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 Folgefehlern 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.).

Das folgende kleine **Beispiel** zeigt typische Fehler und die zugehörigen Meldungen des Compilers. (Die Markierung der statischen Programmstruktur im listing mittels der struct-Option, die Angaben zur Blockverschachtelungstiefe und die cross-reference Liste werden hier nicht gezeigt.)

```
   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 letzen, handelt es sich um Syntaxfehler. In diesem Beispiel hat sich der Parser stets sehr schnell wieder gefangen, es wurden keine Programmteile überlesen.

4.7.2 Allgemeines

Die folgenden Punkte mögen zu einem besseren Verständnis der Fehlerbehandlung des Compilers und damit zu einem schnelleren Auffinden der eigentlichen Fehlerursache beitragen:

- Folgende vier **Typen von Meldungen** werden unterschieden:
  1.) Syntaxfehlermeldungen
  2.) Semantikfehlermeldungen
  3.) Korrekturannahmen (Assumption-Meldungen)
  4.) Allgemeine Fehlerhinweise

- An eigentlichen Programmierfehlern werden **Syntaxfehler, Semantikfehler** und **lexikalische Fehler** unterschieden. Die vom scanner entdeckten lexikalischen Fehler haben in der Meldung die gleiche Form wie Semantikfehler und sind zusammen mit diesen in Abschnitt 4.7.9 aufgeführt. **Korrekturannahmen** zeigen die Reaktion des Parsers auf entdeckte Fehler ; bei den **allgemeinen Fehlerhinweisen** handelt es sich um Compilerbedienfehler.

- **Syntaxfehlermeldungen** werden **automatisch generiert.** Das genaue Format ist weiter unten beschrieben. Die Meldung erfolgt in syntaktischen Kategorien; hierbei wird auf die **Grammatik** von CSSA, wie sie im Anhang von Teil B beschrie-

ben ist, Bezug genommen. Syntaxfehler ziehen i.a. **"assumption-Meldungen"** nach sich, diese Korrekturannahmen geben Aufschluß über das error-recovery.

- Der Compiler arbeitet syntaxgesteuert nach dem **one-pass** Prinzip und besitzt als LL-Parser die **"valid prefix property"**. Das heißt, daß Syntaxfehler zwar in dem Sinne zum **frühest möglichen Zeitpunkt** erkannt werden, als das Programmstück bis zum Fehler einen (syntaktisch) korrekten Präfix darstellt ; die eigentliche Fehlerursache jedoch schon viel früher liegen kann. (Triviales Beispiel: Eine fehlende öffnende Klammer in einem arithmetischen Ausdruck kann erst bei einer schließenden Klammer erkannt werden). Streng genommen gilt dies nur für den ersten Syntaxfehler; da der Parser sich jedoch meistens sehr schnell wieder fängt, werden i.a. **alle** Fehler "frühest möglich" gemeldet. Man beachte auch, daß beispielsweise der typische Fehler des fehlenden Semikolons am Ende einer Zeile natürlich erst am Anfang der **nächsten** (nicht leeren) Zeile entdeckt werden kann!

- Die verschiedenen **Semantikfehler** sind numeriert (BMSnnn); eine Liste der möglichen Fehler zusammen mit Erklärungen befindet sich weiter unten (——> 4.7.9). Man beachte, daß bei Semantikfehlern das fehlerhafte Zeichen i.a. im source-listing nicht unterstrichen ist und der Markierungspfeil sich in der darauffolgenden Zeile befindet, falls der Fehler am Ende einer Zeile entdeckt wurde. Dies hat seinen Grund darin, daß der Parser der semantischen Analyse immer ein Zeichen vorauseilt ("one symbol lookahead").

- Im Gegensatz zum sonst üblichen Sprachgebrauch und im Unterschied zur Grammatik von CSSA handelt es sich beim beschriebenen Compiler bei **deklarierten Bezeichnern** und **undeklarierten Bezeichnern** um **syntaktische Kategorien** im Sinne einer attributierten Grammatik. Dies hat (für den Compilerentwickler) den Vorteil, daß i.a. eine fehlerhafte Redeklaration bzw. eine fehlende Deklaration von Bezeichnern als **Syntaxfehler** (automatisch) gemeldet wird. Dem Benutzer mag die Syntaxfehlermeldung "ILLEGAL SYMBOL ... CHECK, IF <GLOBAL IDENTIFIER> | <LOCAL IDENTIFIER> SHOULD HAVE BEEN USED" mit der darauffolgenden assumption "DECLARATION OF ... ASSUMED TO BE MISSING" zunächst etwas ungewohnt vorkommen. Man beachte daher:
  <GLOBAL IDENTIFIER> bezeichnet einen global definierten Bezeichner ;
  <LOCAL IDENTIFIER> bezeichnet einen lokal definierten Bezeichner ;
  <NEW IDENTIFIER> bezeichnet einen noch nicht definierten Bezeichner.
  Beim **angewandten Auftreten** muß ein Bezeichner deklariert sein, d.h. vom Typ <GLOBAL IDENTIFIER> oder <LOCAL IDENTIFIER>, und darf nicht vom Typ <NEW IDENTIFIER> sein. Beim **definierenden Auftreten** muß ein Bezeichner vom Typ <NEW IDENTIFIER> oder <GLOBAL IDENTIFIER> sein, nicht aber vom Typ <LOCAL IDENTIFIER>.

- **Stringkonstanten** und **Kommentare** dürfen nicht über eine Zeile hinausgehen. Fehlt eine schließende Kommentarklammer '*)', so wird der Rest der Zeile als Kommentar aufgefaßt. Fehlt dagegen ein quote-Zeichen, so wird vom scanner keine Stringkonstante erkannt und der string als Programmteil aufgefaßt. Dies resultiert zwar i.a. in einigen Folgefehlern, verhindert aber andererseits, daß zu lange Textpassagen überlesen werden.

- Nach einem Syntaxfehler kann der Parser u.U. einen kleinen Teil des Programms **überlesen**, bevor er sich wieder fängt. Dieser Teil wird im source-listing **gestrichelt unterstrichen.** Da hierin eventuell vorhandene Fehler nicht entdeckt werden, sollte diesem Teil besondere Aufmerksamkeit gewidmet werden! **Achtung:** Werden sehr kurze Passagen oder etwa nur ein einziges Zeichen überlesen, so kann ein "gestricheltes" Unterstreichen von einem durchgehenden Unterstreichen nicht unterschieden werden. Man sollte sich in diesem Fall nicht irreleiten lassen und nach vermeintlichen Fehlern suchen!

- Der Parser ist **im** Vergleich zu Compilern anderer blockorientierter Sprachen recht robust gegenüber **Fehlern in der Blockstruktur.** Da derartige Fehler jedoch eine Reihe von (vorwiegend semantischen) Folgefehlern verursachen können, sollte in einem derartigen Fall die vom Parser angenommene Blockstruktur anhand der am rechten Rand ausgedruckten **Markierung der Blockstruktur** (——> 4.4) oder der Korrekturannahmen überprüft werden. Die **struct-option** kann hierbei ebenfalls sehr hilfreich sein; ihre Verwendung wird auch deswegen empfohlen, da dann ein strukturiertes Aufschreiben des Programms quasi erzwungen wird und so Fehler in der Blockstruktur von vornherein vermieden werden.

4.7.3 Problematik der Fehlerbehandlung

Einige kurze Bemerkungen zur **Problematik** und zum **Prinzip** der verwendeten Syntaxfehler-Behandlung mögen zum weiteren Verständnis der Reaktion des Parsers und der erzeugten Meldungen bei fehlerhaften Programmen beitragen.

- Der Parser erkennt einen Syntaxfehler daran, daß ein Symbol nicht zu der Menge der an dieser Stelle **erwarteten Symbole** gehört. Falls der "eigentliche Fehler" tatsächlich an dieser Stelle (und nicht schon früher !) begangen wurde, so sind zunächst drei einfache Fälle möglich:
  (1) Das Symbol ist **zuviel.**
  Bsp: **procedure** P(**int:** M) **is** ; **var string** : A ; ...
  Das Semikolon nach 'is' ist falsch. Die richtige Aktion bestünde im Überlesen dieses Zeichens.
  (2) Ein Symbol **fehlt.**
  Bsp: **var int** : A,B,C D,E ;
  Zwischen 'C' und 'D' fehlt ein Komma (oder ist vielleicht das 'D' zuviel ?) Die richtige Aktion bestünde darin, daß das Komma (virtuell) "eingefügt" würde.
  (3) Ein Symbol ist **falsch.**

Bsp: **if** X>5 THAN **replace by** FULL ; **endif** ;
Es muß 'then' statt 'than' heißen. Die richtige Aktion bestünde im Ersetzen von 'than' durch 'then'.

- Nicht alle Syntaxfehler sind einfache **"one-token" Fehler**, die lokal "korrigiert" werden könnten und mit denen der Parser i.a. sehr gut fertig wird. Fehler, die vom Programmierer aus Unkenntnis der genauen Syntax begangen werden, sind oft besonders schwierig zu behandeln. Beispiel: Deplazierte Prozedurdeklaration.

- Der Parser muß als **one-pass one-symbol-lookahead** Analysator allein auf Grund des bisher analysierten Programmteils und des fehlerhaften Symbols zu einer vernünftigen (nicht notwendig richtigen!) Aktion kommen, die ein Weiteranalysieren des Programms garantiert!

- Das **one-pass-Verhalten** bedeutet hier, daß der Parser nicht zurückgehen kann ("backtracking") und das Programm an einer schon analysierten Stelle "verbessern" kann, auch wenn erkannt wurde, daß der eigentliche Fehler an einer früheren Stelle begangen wurde.

- **One-symbol-lookahead** bedeutet hier, daß der Parser nicht über das fehlerhafte Symbol hinaussehen kann, um eine geeignete Aktion einzuleiten.

- Es sollen möglichst **alle Fehler** gefunden werden und **keine Folgefehler** gemeldet werden. Dies ist nur möglich, wenn das fehlerhafte Programm "richtig" (virtuell) repariert wird und möglichst **frühzeitig** ein **"recovery"** durchgeführt wird.

- Syntaxfehler können zum Verlust semantischer Attribute führen ; dies kann eine Reihe von **semantischen Folgefehlern** auslösen.

4.7.4 Prinzip der verwendeten Methode

Um mit den geschilderten Schwierigkeiten fertig zu werden, wurden eine Reihe von allgemeinen Maßnahmen entwickelt, die teilweise theoretischer Natur sind ("immediate error detection property" durch sogen. "starke" LL-Analyse ; "immediate prediction property" ; Verwendung von sogen. "nonnullable" Grammatiken, dynamische Verwaltung aller möglichen Synchronisationssymbole etc.) und an anderer Stelle beschrieben werden; andererseits kommen **verallgemeinerbare Heuristiken** zum Einsatz, von denen einige relevante hier genannt werden sollen:

- **Spelling correction:** 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, ein Zeichen zuviel vorhanden ist oder zwei benachbarte Zeichen vertauscht sind.

Beispiel für eine Meldung des spelling-correctors : 'THAN' ASSUMED TO BE A MISSPELLED 'THEN'.

- **Kostenrechnung:** Hiermit wird gesteuert, ob ein Symbol überlesen werden soll oder ob mit einem Symbol recovery versucht werden soll. Die Kostenrechnung steuert die "Überlegungen" des Parsers und äußert sich bei gewichtigeren Entscheidungen in assumption-Meldungen der Art "<symbol> ASSUMED TO BE PART OF <synt-unit> STARTING IN LINE <n>" oder "<symbol> ASSUMED TO BE NOT PART OF <synt-unit> BECAUSE...".
  Mit der Kostenrechnung 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.

- **Chamäleon/anytype-Prinzip:** Bezeichner, die fehlerhaft oder unvollständig definiert wurden, werden als "Chamäleons" behandelt. Ihr Typ ist zu allen anderen Typen kompatibel, eine Reihe von Attributen werden bei der semantischen Analyse nicht überprüft. Ergeben sich Datentyp und Typklasse später eindeutig aus dem Kontext, so werden diese Attribute nachträglich zugeordnet. Diese Maßnahme reduziert die Anzahl der semantischen Folgefehler.

4.7.5 Syntaxfehler

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.

Eine Syntaxfehlermeldung hat folgendes Format:

```
==> SYNTAX-ERROR IN LINE <n> : ILLEGAL SYMBOL '<token>'
    IN <synt-unit> [STARTING IN LINE <m>]
    CHECK, IF <list of expected symbols> SHOULD HAVE
    BEEN USED INSTEAD [OF (OR BEFORE) '<token>']
```

Darin bedeuten:
<n> : Die aktuelle Zeilennummer
<token> : Das fehlerhafte Eingabesymbol
<synt-unit> : Die gerade analysierte syntaktische Einheit aus der CSSA-Grammatik. Es wird die äußerste syntaktische Einheit genannt von denen, die an der Fehlerstelle Terminalsymbole erwarten.
<list of expected symbols> : Eine Liste von Terminalsymbolen aus der CSSA-Grammatik. Diese Liste stellt die Menge aller (syntaktisch) legalen Folgezeichen dar, die an dieser Stelle erwartet wurden und den korrekten Programmpräfix verlängern würden.
Der Teil "STARTING IN LINE <n>" der Meldung wird nur ausgedruckt, wenn die Zeilennummer <m> der Zeile, in der <synt-unit> beginnt, verschieden von der aktuellen Zeilennummer <n> ist.
Der letzte Teil der Meldung erscheint nur, wenn <token> ein Schlüsselwort ist.
Falls die Syntaxfehlermeldung mit "==> SYNTAX ERROR(n)" beginnt, so wurde sie vom HEP-Parser erzeugt ; n zeigt dabei die Verschachtelungstiefe des (evtl. rekursiv aufgerufenen) HEP-Parsers an. Dieser Fall deutet auf dicht aufeinanderliegende oder schwerwiegende Fehler wie ein systematischer Mißbrauch der Syntax hin. Gleichzeitig zeigt er, daß die Fehlerbehandlung mit Schwierigkeiten zu kämpfen hat; eine weniger gute Fehlerdiagnose und Fehlerbehandlung durch falsche Annahmen könnte der Fall sein.
Die Meldung "SYNTAX ERROR IN LINE <n>: END OF SOURCE TEXT" wird generiert, wenn der Parser auf ein vorzeitiges Programmende gestoßen ist.

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).

4.7.6 Semantikfehler

Die **Semantikfehlermeldungen** haben folgende Form:

```
==> ERROR BMSnnn IN LINE <m> (BEFORE '<token>') : <text>
```

Darin bedeuten:
nnn : Die Fehlernummer
<m> : Die Zeilennummer, in der der Fehler festgestellt wurde
<token> : Das Symbol, das vom Parser gerade gelesen wurde. Die Fehlermeldung muß sich auf die semantische Überprüfung von Attributen von Symbolen beziehen, die vor diesem Symbol liegen. Achtung : <token> muß nicht in der Zeile <m> stehen, falls der Semantikfehler beim letzten Symbol einer Zeile ausgelöst wurde!

<text> : Eine Fehlererläuterung. Zu verschiedenen Nummern nnn kann der gleiche Fehlertext vorhanden sein.
Die Semantikfehlermeldungen werden in Abschnitt 4.7.9 genauer erläutert.

4.7.7 Assumption-Meldungen

Assumption-Meldungen (Korrekturannahmen) werden bei fehlender Variablendeklaration, bei der spelling-correction und beim error-recovery erzeugt. Sie stellen keine Fehlermeldungen dar, sondern geben Hinweise darauf, wie der Parser auf eine vorangegangene Fehlersituation reagiert hat.

Assumption-Meldungen haben die Form :

```
- ASSUMPTION IN LINE <n> : <text>
```

Es können folgende Arten von Korrekturannahmen erzeugt werden :

DECLARATION OF <identifier> ASSUMED TO BE MISSING
Diese Meldung wird erzeugt, wenn an einer Stelle, wo <local identifier> oder <global identifier> erwartet wurde, ein <new identifier> gefunden wurde, und dieser Bezeichner durch die spelling-correction Routine nicht verbessert werden konnte. Der Bezeichner wird (lokal) als "Chamäleon" in die Symboltabelle eingetragen. Der Meldung geht stets eine Syntaxfehlermeldung voran ; mit einer weiteren assumption-Meldung wird das sofort durchgeführte recovery bestätigt.

<tok1> ASSUMED TO BE A MISSPELLED <tok2>
Meldung der spelling-correction Routine, die nach einem Syntaxfehler aufgerufen wurde. Token <tok1> wurde in <tok2> verändert, da entweder <tok2> ein (ähnlich geschriebener) deklarierter Bezeichner ist und ein Bezeichner an dieser Stelle syntaktisch erlaubt ist, oder <tok2> ein (ähnliches) an dieser Stelle erwartetes Schlüsselwort darstellt. Einige leicht zu verwechselnde Sonderzeichen können ebenfalls korrigiert werden. Durch die darauffolgende assumption-Meldung sollte das auf diese Korrektur hin erfolgte recovery bestätigt werden. Da die Korrektur nur virtuell erfolgt, wird in dieser Meldung noch auf das nicht korrigierte token <tok1> Bezug genommen.

CONCATENATION ERROR ASSUMED: '<string1>' CHANGED TO '<string2>'
Meldung der spelling-correction Routine, die nach einem Syntaxfehler aufgerufen wurde. Da ein an der Fehlerstelle erwartetes Symbol einen Präfix der gefundenen Zeichenfolge <string1> darstellt, wurde angenommen, daß ein Leerzeichen vergessen wurde. Dieses Leerzeichen wurde eingefügt.
Achtung: Es wurde nicht überprüft, ob die Zeichenfolge nach dem eingefügten Leerzeichen ein an dieser Stelle gültiges Symbol darstellt ; evtl. kommt es hierbei zu Folgefehlern !

<symbol> [(COL.<n>)] ASSUMED TO BE PART OF <synt-unit> [EXPECTED IN <unit>] STARTING IN LINE <m>

Diese Meldung wird bei (hoffentlich) erfolgreichem recovery erzeugt. Falls der HEP-Parser aktiv war, wird er (zumindest in einer Rekursionsstufe) verlassen und der Parsingprozess auf einer Stufe fortgeführt, die vor dem letzten Syntaxfehler aktiv war. Der letzte Fehler ist damit "repariert": Es wird angenommen, daß das genannte Symbol zu der bereits teilweise analysierten oder zumindest erwarteten syntaktischen Einheit <synt-unit> gehört. In diese wird nun wieder "eingestiegen".

<symbol> ASSUMED TO BE MISSING IN <synt-unit> [EXPECTED IN <unit>] STARTING IN LINE <m>

Es ist ein Symbol gefunden worden, mit dem recovery versucht werden soll. Dazu müssen allerdings einige syntaktische Einheiten als fehlend angenommen werden, d.h. einige Symbole müssen (virtuell) "eingefügt" werden. Die gewichtigeren Terminalsymbole werden hier genannt.

<symbol> [(COL.<n>)] ASSUMED TO BE NOT PART OF <synt-unit> [EXPECTED IN <unit>] STARTING IN LINE <m> BECAUSE ESSENTIAL PARTS OF <unit2> ARE STILL MISSING

Es ist ein Symbol gefunden worden, mit dem prinzipiell recovery durchgeführt werden könnte. Die Kostenrechnung kam allerdings zu dem Ergebnis, daß auf ein recovery verzichtet werden soll, da sonst wichtige erwartete Symbole als fehlend angenommen werden müssen.
Bemerkung: Die Entscheidung kann natürlich falsch sein ; Folgefehler wären der Fall.

4.7.8 Allgemeine Fehlerhinweise

Allgemeine Fehlerhinweise erfolgen, wenn Compiler-Parameter falsch gesetzt werden oder Beschränkungen überschritten werden, die im Grunde nichts mit der CSSA-Sprache zu tun haben ("Compilerbedienfehler").

Es gibt folgende Fehlerhinweise :

- ==> TOO MANY ILLEGAL CHARACTERS. NO FURTHER MESSAGE
  (Keine weitere Meldung zu "illegal characters", diese werden im source-listing jedoch weiterhin unterstrichen. Keine weiteren Konsequenzen)

- ==> TOO MANY ILLEGAL CHARACTERS. COMPILATION STOPPED
  (Keine weitere Analyse des Programms, auch keine Untersuchung auf Fehler. Das source-listing wird weiter ausgedruckt)

- ==> WARNING IN LINE <n> : TEXT IN COL 73-80 'xxxxxxxx' IGNORED
  (Keine weiteren Konsequenzen, zählt nicht als Fehler; Folgefehler sind wegen des überlesenden Programmteils möglich)

  ==> MAXDERIV LIMIT EXCEEDED

(Der Ableitungsbaum wird nicht weiter gedruckt. Zählt nicht als Fehler, keine weiteren Konsequenzen)

- ==> MAXPAGES LIMIT EXCEEDED
  (Kein source-listing mehr. Ansonsten keine Konsequenzen)

- ==> MAXERROR LIMIT EXCEEDED. COMPILATION STOPPED
  (Keine weitere Analyse des Programms, source-listing wird allerdings weiter ausgedruckt)

- ==> WRONG CONTROL-CARD AT LINE <n> IGNORED
  (Keine weiteren Konsequenzen, zählt nicht als Fehler)

- ==> INCORRECT OPTION '<symbol>' IN THE PARM FIELD REJECTED
  (Keine weiteren Konsequenzen, zählt nicht als Fehler)

- ==> WRONG SPECIFICATION FOR <parmfield> : '<symbol>' IN THE PARM_FIELD IGNORED
  (Keine weiteren Konsequenzen, zählt nicht als Fehler)

- ==> EMPTY PARM_FIELD FOR <parmfield>
  (Keine weiteren Konsequenzen, zählt nicht als Fehler)

4.7.9 Liste der semantischen Fehlermeldungen

Die Bedingungen und Restriktionen der statischen Semantik werden in der CSSA-Sprachbeschreibung [BMS-B], insbesondere in den Unterkapiteln 3 und 4 des entsprechenden Sprachkapitels, angegeben. Werden diese Bedingungen verletzt, so erfolgt eine Semantikfehlermeldung.

BMS001 ILLEGAL SIDE-EFFECT TO GLOBAL VARIABLE <id> IN FUNCTION
In Funktionen dürfen globale Variablen nicht verändert werden, dies wäre ein unerlaubter Seiteneffekt.

BMS002 LEAVE <id> IS MISPLACED
Das Statement "leave <id>" muß sich statisch verschachtelt in der Facette <id> befinden.

BMS003 MISPLACED IDLE OPERATION
Eine idle-Operation kann nur auf Facettenebene definiert werden, nicht aber auf Scriptebene.

BMS004 OPERATION <id> IS STILL UNDEFINED
Eine Operation, die mit 'public' oder 'private' deklariert wurde, wurde im script nicht definiert.

BMS005 FACET <id> IS STILL UNDEFINED
Eine nach 'facethead' deklarierte Facette wurde nicht definiert.

BMS006 FACET <id> IS REDEFINED
Eine Facette wurde im gleichen scope mehrfach definiert.

BMS007 OPERATION <id> IS STILL UNDEFINED
Eine Operation, die mit 'public' oder 'private' deklariert

wurde, wurde in der Facette nicht definiert.

BMS008 FACET <id> IS STILL UNDEFINED
Eine nach 'facethead' deklarierte lokale Facette wurde nicht definiert.

BMS009 PROCEDURE <id> IS REDEFINED
Eine Prozedur wurde im gleichen scope mehrfach definiert.

BMS010 FUNCTION <id> IS REDEFINED
Eine Funktion wurde im gleichen scope mehrfach definiert.

BMS011 MORE THAN 5 DIMENSIONS
Bei einem array-Zugriff ist die Index-Liste zu groß : Ein array darf höchstens 5 Dimensionen haben.

BMS012 NUMBER OF INDICES NOT EQUAL TO NUMBER OF DIMENSIONS
Die Zahl der verwendeten Indizes bei einem array-Zugriff stimmt nicht mit der bei der Definition des arrays angegebenen Anzahl überein.

BMS013 UNMATCHED IN-PARM OF TYPE <type> (PARMS WERE CHECKED FROM RIGHT TO LEFT!)
Die aktuelle Inparm-Liste bei 'setup', 'replace', 'initial', 'delete', 'find', 'call' oder Funktionsaufruf ist länger als die formale Inparm-Liste. Achtung: Die Parameter wurden einander von rechts nach links zugeordnet.

BMS014 IN-PARMS MISSING (PARMS WERE CHECKED FROM RIGHT TO LEFT)
Die aktuelle Inparm-Liste bei 'setup', 'replace', 'initial', 'delete', 'find', 'call' oder Funktionsaufruf ist kürzer als die formale Inparm-Liste. Achtung: Die Parameter wurden einander von rechts nach links zugeordnet.

BMS015 OUT-PARMS MISSING
Bei einem call-Statement fehlt das arrowsymbol '-->' mit nachfolgenden aktuellen out-Parametern ; die gerufene Prozedur besitzt out-Parameter.

BMS016 <id> IS A CONSTANT
Ein out-Parameter einer Prozedur darf keine Konstante sein.

BMS017 UNMATCHED OUT-PARM OF TYPE <type> (PARMS WERE CHECKED FROM RIGHT TO LEFT!)
Bei einem Prozeduraufruf durch 'call' wurden zu viele aktuelle out-Parameter angegeben. Achtung: Die formalen/aktuellen Parameter wurden einander von rechts nach links zugeordnet.

BMS018 OUT-PARMS MISSING (PARMS WERE CHECKED FROM RIGHT TO LEFT!)
Bei einem Prozeduraufruf durch 'call' wurden zu wenig aktuelle out-Parameter angegeben. Achtung: Die formalen/aktuellen Parameter wurden einander von rechts nach links zugeordnet.

BMS019 <id1> IS NOT A FIELD OF <id2>
Es wurde mit Punkt-Notation auf einen record <id2> zugegriffen ; die record-Definition enthält jedoch kein Feld <id1>.

BMS020 <id> IS A CONSTANT
Ein Bezeichner in einem globalen Feld eines patterns darf keine Konstante sein.

BMS021 PROC/FUNCTION <id> IS STILL UNDEFINED
Eine in <procedurehead> deklarierte Prozedur oder in <functionhead> deklarierte Funktion wurde nicht definiert.

BMS022 MISPLACED PORT DECLARATION
Die port-Deklaration steht nicht direkt in einer Operation oder Prozedur.

BMS023 ILLEGAL TYPE IN ARRAY DECLARATION : <type>. USE INT OR ENUM
Die <expression> in "array (<expression>..<expression>...)" müssen vom Typ INT oder Aufzählungstypen sein.

BMS024 ILLEGAL TYPE IN ARRAY DECLARATION : <type>. USE INT OR ENUM
Fehler bei der Deklaration eines mehrdimensionalen arrays : Die <expression> in "array (<expression>..<expression>...)" müssen vom Typ INT oder Aufzählungstypen sein.

BMS025 MORE THAN 5 DIMENSIONS
Fehler bei der Deklaration eines arrays : Ein array darf höchstens 5 Dimensionen haben.

BMS026 ACQUAINTANCE BEFORE ':=' MUST BE TYPED
Fehler bei der Initialisierung in einer Konstanten- oder Variablendeklaration : Bei der Verwendung von AGENT als Typ muß ':-' statt ':=' verwendet werden.

BMS027 TYPED ACQUAINTANCE NOT ALLOWED BEFORE ':-'
Fehler bei der Initialisierung in einer Konstanten- oder Variablendeklaration : Bei typisierten Agenten muß ':=' statt ':-' verwendet werden.

BMS028 MISPLACED RETURN STATEMENT
Ein return-Statement wurde nicht in einer Funktion verwendet.

BMS029 <id> IS A CONSTANT
Bei "put...into <id>" darf <id> keine Konstante sein.

BMS030 'PUT' FOR SET <id> IS PLACED IN A LOOP FOR THE SAME SET
In einer loop-Schleife darf für die durchlaufene Menge kein 'put' ausgeführt werden. Da die Zugriffsreihenfolge undefiniert ist, wäre sonst die Determiniertheit nicht gewährleistet.

BMS031 <id> IS A CONSTANT
Bei "remove...from <id>" darf <id> keine Konstante sein.

BMS032 <id> IS A CONSTANT
In einer Zuweisung darf die linke Seite keine Konstante sein.

BMS033 TYPED ACQUAINTANCE NOT ALLOWED BEFORE ':-'
Fehler bei einer Zuweisung : Bei typisierten Agenten muß ':=' statt ':-' verwendet werden.

BMS034 ACQUAINTANCE BEFORE ':=' MUST BE TYPED
Fehler bei einer Zuweisung: Bei der Verwendung von AGENT als Typ muß ':-' statt ':=' verwendet werden.

BMS035 ILLEGAL TYPE OF LOOP_INDEX. USE "INT" OR "ENUM"
Bei "loop...for...in <expr1>..<expr2>..." ist <expr1> kein Aufzählungstyp oder nicht vom Typ INT.

BMS036 NESTED LOOPS FOR ONE SET NOT ALLOWED!
Geschachtelte loop-for-of-Schleifen für die gleiche Menge sind nicht möglich. (Implementierungsrestriktion)

BMS037 EXIT <id> IS REDEFINED
Ein exit-Name wurde innerhalb des aktuellen scopes mehrfach definiert.

BMS038 EXIT <id> IS STILL UNDEFINED
Bei "loop...until" wurde ein event-Bezeichner deklariert, für den kein exit definiert wurde.

BMS039 ESCAPE MUST BE NESTED IN A LOOP
'escape' wurde außerhalb einer loop-Schleife verwendet.

BMS040 CONTINUE MUST BE NESTED IN A LOOP
'continue' wurde außerhalb einer loop-Schleife verwendet.

BMS041 ILLEGAL OPERATOR
Bei einem Booleschen Vergleich der Typen AGENT, SCRIPT und OPER kann nur '=' oder '/=' als Operator verwendet werden (und nicht '<', '>', '<=', '>=', 'in'). Records, sets, arrays und Boolesche Ausdrücke können gar nicht verglichen werden.

BMS042 IDENTIFIER OF CLASS <cl> NOT ALLOWED IN EXPRESSIONS
Diese Typklasse ist in Expressions nicht zulässig. Zulässig sind höchstens (je nach Kontext) VARIABLE, LITERAL, OPERATION, FUNCTION oder USER-DEFINED-TYPE vom Typ SCRIPT; auf keinen Fall aber LOOP, PORT, EVENT, FACET, PROCEDURE, RELATION.

BMS043 IN-PARMS MISSING
Ein Funktionsaufruf enthält keine Parameter, obwohl die Funktionsdefinition Parameter besitzt.

BMS044 INTERNAL ERROR: TABLE OVERFLOW IN <module>
Bei einem internen Fehler wird der Analyseprozeß abgebrochen. Im source-listing erscheint eine "===...===" Zeile, der Rest der Eingabe wird noch gelistet, aber nicht mehr analysiert. Dieser Fehler kann eventuell bei extrem tief geschachtelten Strukturen auftreten.

BMS045 <id> IS NOT OF TYPE <type>
Der angegebene Bezeichner ist nicht vom erwarteten Datentyp. Es gibt folgende Datentypen: SET, ARRAY, RECORD, OPER, SCRIPT, ENUM, INT, AGENT, BOOL, STRING, REAL.

BMS046 <id> IS NOT A <type-class>
Der angegebene Bezeichner hat nicht die aus dem Kontext erwartete Typklasse. Welche Typklasse er tatsächlich hat, läßt sich bspw. aus der cross-reference Liste entnehmen. Es gibt folgen-

de Typklassen: USER_DEFINED_TYPE, LOOP, LITERAL, PORT, VARIABLE (umfaßt auch CONSTANT), EVENT, FACET, OPERATION, PROCEDURE, FUNCTION, RELATION.

BMS047 <type1> IS USED INSTEAD OF <type2>
Bei der Typüberprüfung von expressions wurde <type1> gefunden, obwohl aus dem Kontext <type2> erwartet wurde.

BMS048 ILLEGAL TYPE <type> INSTEAD OF INT OR REAL
Es wurde der Typ <type> bei der Typüberprüfung einer expression gefunden, obwohl nur INT oder REAL zulässig sind.

BMS049 ONLY A BUILT-IN TYPE OR 'ENUM' ALLOWED HERE
Bei arrays ("array...of <type>"), sets ("set of <type>"), in records (Recordfelder) und im print-Statement sind nur die einfachen Datentypen INT, BOOL, AGENT, OPER, SCRIPT, STRING, REAL oder Aufzählungstypen (als einzige selbstdefinierte Typen) erlaubt, nicht jedoch SET, ARRAY oder RECORD.

BMS050 ILLEGAL DATA TYPE IN MESSAGE
In einer message (send- oder reply-statement) kann kein set oder Aufzählungstyp verwendet werden, auch nicht verborgen in einem array oder record.

BMS051 ARRAY-VALUED FUNCTION NOT ALLOWED IN A MESSAGE
Es können zwar ganze arrays verschickt werden, in einer message darf jedoch kein Funktionsaufruf stehen, der ein array liefert. (Implementierungsrestriktion)

BMS052 RECORD-VALUED FUNCTION NOT ALLOWED IN A MESSAGE
Es können zwar ganze records verschickt werden, in einer message darf jedoch kein Funktionsaufruf stehen, der einen record liefert. (Implementierungsrestriktion)

BMS053 ILLEGAL DATA TYPE IN PATTERN
In einem pattern darf kein SET, Aufzählungstyp oder typisierter Agent verwendet werden, auch nicht verborgen in einem array oder record.

BMS054 INCOMPATIBLE SUBTYPES
Dieser Fehler wird bei der Überprüfung der Typgleichheit zweier expressions entdeckt (linke/rechte Seite einer Zuweisung ; aktuelle/formale Parameterlisten ; Boolescher Vergleich zweier expressions etc.). In diesen Fällen muß völlige Typgleichheit herrschen, d.h. die generischen Typen RECORD, ENUM, ARRAY, SET und AGENT (typisiert) müssen sogar vom gleichen Subtyp sein.

BMS055 TYPE OF ACQUAINTANCE CANNOT BE DETERMINED
Der script-Typ einer untypisierten agent-expression (z.B. freie agent-Variable) kann nicht ermittelt werden. Abhilfe : Bei einer Zuweisung auf der rechten und linken Seite beidesmal typisierte oder aber untypisierte Agenten verwenden.

BMS056 TEXT AFTER END OF PROGRAM
Der Parser hat bereits das Programmende erwartet. Dieser Fehler sollte bei der Analyse eines CSSA-Programms nicht auftreten.

BMS057 UNMATCHED '<symbol>' FOUND
Diese Fehlermeldung erscheint, wenn der Parser durch einen vorangegangenen Syntaxfehler 'außer Tritt' geraten ist und versucht, wieder einzusteigen. Dabei wurde das Schlüsselwort <symbol> gefunden, das an dieser Stelle deplaziert scheint, da es nur innerhalb einer syntaktischen Einheit auftreten darf, die nicht gefunden wurde (z.B. wegen eines vorangegangenen Syntaxfehlers).

BMS058 INCOMPLETE PROGRAM : <tokenlist> MISSING IN <synt-unit> [EXPECTED IN <unit>] STARTING IN LINE <n>
Bei Programmende wurde festgestellt, daß einige wichtige tokens, die noch erwartet wurden, nicht gekommen sind. I.a. deutet dies entweder auf schwerwiegende frühere Fehler in der Blockstruktur hin oder auf ein vorzeitiges Programmende.

BMS059 '*)' MISSING: '*)' ASSUMED AT END-OF-LINE
Es fehlt die schließende Kommentarklammer in dieser Zeile. Der Rest der Zeile wird als Kommentar behandelt. (Lexikalischer Fehler)

BMS060 ILLEGAL CONSTANT
Nach dem Exponentzeichen 'E' einer real-Konstanten fehlt der Exponent. (Lexikalischer Fehler)

BMS061 EXPONENT OVERFLOW IN REAL CONSTANT : <exponent>
Real-Zahl größer als 10**74 (Lexikalischer Fehler)

BMS062 INTEGER CONSTANT TOO LARGE
Integer-Konstante hat mehr als 9 Stellen. (Lexikalischer Fehler)

BMS063 <n> ILLEGAL CHARACTERS FOUND
Es wurden Zeichen gefunden, die nicht zum erlaubten Eingabealphabet gehören. Die Zeichen werden vom scanner überlesen. (Lexikalischer Fehler)

BMS064 IDENTIFIER <id> TOO LONG. (MAX. 30 CHAR.)
Ein Bezeichner darf höchstens 30 Stellen lang sein. (Lexikalischer Fehler)

4.8 Externe Funktionen und Prozeduren

Mit externen Funktionen und Prozeduren besteht die Möglichkeit, Unterprogramme, die in **anderen Programmiersprachen** (z.B. SIMULA) geschrieben wurden, von CSSA-Programmen aus aufzurufen. Dadurch ist es möglich, in gezielter Weise auf interne Größen des Laufzeitsystems (das heißt der abstrakten stack-Maschinen des Betriebssystemkerns) zuzugreifen und diese eventuell zu ändern, um so **Pseudospracherweiterungen** oder **Erweiterungen der Systemsteuerung** vorzunehmen.

Damit eröffnen sich vielfältige Möglichkeiten, einige **Beispiele** seien angedeutet:

- Erweiterung der Trace-Funktion durch Zugriff auf den Laufzeitkeller und den aktuellen Aktivierungssatz
- Programmabhängiges Ein- und Ausschalten von Trace-Funktionen
- Erweiterte Statistiken über die Auslastung einzelner Komponenten des simulierten Mehrrechnersimulationssystems durch Zugriff auf die SIMULA-eventqueue
- Erweitertes nichtdeterministisches Verhalten einzelner Agenten durch Verwendung der SIMULA-Zufallsfunktion
- Pseudoerweiterung der CSSA-Sprache durch Realisierung einer externen Funktion, die den aktuellen Facettennamen zurückliefert mittles Zugriff auf den ACB (agent-control-block)
- Absetzen von BS 2000-Betriebssystemkommandos
- Hardwareabhängige Ein-/Ausgabeprogrammierung (z.B. Terminalsteuerung)
- Verwendung von numerischen Berechnungsverfahren aus Programmbibliotheken und Verfahren aus anderen Methodenbanken, die in Form von (Fortran)-Unterprogrammen vorliegen.

Der **Kopf** externer Funktionen und Prozeduren wird entsprechend den built-in Funktionen als 'procedurehead' bzw. 'functionhead' mit der Angabe 'external' im CSSA-Programm deklariert, wie in B 14.4 und B 14.5 beschrieben. Es muß dort also insbesondere der Typ eventueller Parameter und im Falle externer Funktionen auch der Datentyp des zurückgelieferten Wertes spezifiziert werden. Die in den formalen Parameterlisten der Kopfdeklaration genannten Namen der Parameter haben keine weitere Bedeutung.

Die externen Funktionen und Prozeduren selbst müssen in **SIMULA** geschrieben sein und in eine **Datei** gestellt werden, die neben diesen Funktionen und Prozeduren auch andere Deklarationen und Hilfsprozeduren enthalten kann. Diese Datei muß beim Aufruf des CSSA-Entwicklungssystems genannt werden:

- DO $SPENKE.P.CSSA,EXTERNAL=<Dateiname>

Der Inhalt der Datei wird auf oberster Ebene in das Laufzeitsystem einkopiert und zusammen mit diesem und dem aus dem CSSA-Programm generierten SIMULA-Programm sowie den Routinen des Mehrrechnersimulationssystems weiterübersetzt. Zur Vermeidung von Namenskonflikten generiert der CSSA-Compiler aus dem Aufruf einer externen Funktion oder Prozedur einen Aufruf einer gleichnamigen SIMULA-Prozedur, der die Buchstabenfolge 'CSSA' vorangestellt wird. Bei der Verwendung von Hilfsprozeduren und anderen selbst definierten Größen in der externen Datei ist auf die Vermeidung von Namenskollisionen mit vorhandenen Bezeichnern des Laufzeitsystems zu achten.

Syntaxfehler in externen Routinen oder fehlende externe Routinen werden vom CSSA-Compiler nicht entdeckt, sie werden erst im zweiten Übersetzungsschritt vom SIMULA-Compiler gemeldet, wenn das generierte SIMULA-Programm zusammen mit dem Laufzeitsystem und den externen Routinen weiterübersetzt wird. Die Meldungen des SIMULA-Compilers werden in die Datei CSSAMESS.<Name der source-Datei> geschrieben (——> 3.). Eventuelle **Laufzeitfehler** führen bei der Ausführung der Programme während der interaktiven CSSA-Sitzung zu einer Unterbrechung mit einer Meldung des SIMULA-Laufzeitsystems.

Als **Beispiel** sei hier die Definition einer externen Funktion angegeben, die eine Zufallszahl vom Typ INT aus einem bestimmbaren Bereich liefert. Im **CSSA-Programm** wird an geeigneter Stelle definiert:

- **functionhead** RANDOM (**int:** LOW,UP) **returns int external;**

Der Aufruf könnte dann lauten:

- ZUF:=RANDOM(1,10);

In die externe Datei werden folgende SIMULA-Definitionen geschrieben:

```
integer RANDOM_MEMORY;
integer procedure CSSARANDOM(LOW,UP); integer LOW,UP;
   begin if RANDOM_MEMORY=0 then RANDOM_MEMORY:=999;
         CSSARANDOM:=Randint(LOW,UP,RANDOM_MEMORY);
   end;
```

Bezüglich des Aufrufs externen Funktionen gelten folgende **Konventionen:**

- Einer CSSA-Prozedur entspricht eine procedure in SIMULA
- Einer CSSA-Funktion entspricht eine <type>-procedure in SIMULA
- Der CSSA-Name der Funktion oder Prozedur erhält in SIMULA den Präfix 'CSSA'
- out-Parameter von Prozeduren entsprechen name-Paramtern in SIMULA

In der folgenden Tabelle ist aufgeführt, welche **CSSA-Datentypen** (in der linken Spalte) welchen SIMULA-Datentypen (rechte Spalte) entsprechen.

- INT - INTEGER
- BOOL - BOOLEAN
- REAL - LONG REAL
- STRING - TEXT
- OPER - TEXT
- ENUM - INTEGER
- SCRIPT - INTEGER
- AGENT - REF(ACB)
- ARRAY - REF(FIELD) bzw. REF(<type>_ARRAY)
- SET - REF(HEAD) bzw. REF(<type>_SET)
- RECORD - REF(RECORD) bzw. REF(RECORD<n>)

Bemerkungen:

- Der zugeordnete integer-Wert eines Wertes eines **CSSA-Aufzählungstyps** entspricht dem Wert, der durch Anwendung der Funktion POS auf den Enum-Wert geliefert wird. (Dem ersten Literal in der Aufzählung ist der integer-Wert 0 zugeordnet.)

- Der zugeordnete integer-Wert eines **scripts** ergibt sich aus der vom CSSA-Compiler vorgenommenen Durchnumerierung aller scripts. Das Interface hat die Nummer 1, das erste übersetzte script die Nummer 2. NOSCRIPT ist der Wert 0 zugeordnet.

- Bei **arrays** wird ein Verweis auf ein Objekt geliefert, das aus der Oberklasse **FIELD** und einer entsprechend dem Typ des arrays spezifischen Unterklasse <type>_ARRAY gebildet wurde. Der Aufbau dieses Kontrollblockes ist in Teil E dokumentiert. Er enthält beispielsweise einen Verweis auf den Deskriptor des arrays und eine Prozedur PUSH_ELEMENT, mit dem auf die Werte des arrays zugegriffen werden kann.

- Bei **sets** wird ein Verweis auf ein Objekt geliefert, das aus der Oberklasse **HEAD** und einer entsprechend dem Typ des sets spezifischen Unterklasse <type>_SET gebildet wurde. Der Aufbau dieses Kontrollblockes ist in Teil E dokumentiert.

- Bei **records** wird ein Verweis auf ein Objekt geliefert, das aus der Oberklasse **RECORD** und einer jeweils speziell generierten Unterklasse gebildet wurde. Genaueres entnehme man dem Aufbau der class RECORD, die in Teil E dokumentiert ist.

- Bei Agenten wird ein Verweis auf einen agent-control-block (ACB) geliefert. Die wichtigsten Felder des ACB seien hier aufgeführt, um mögliche Manipulationen anzudeuten:
  ref(MODULE) ASS_SCRIPT - Verweis auf den ausführbaren Code des zugehörigen script-Modul. Dieses enthält unter anderem ein Feld PRINT_NAME vom Typ TEXT, das den Namen des scripts angibt, und ein Feld OWNMODE vom Typ INTEGER, das die eindeutige Nummer des zugehörigen scripts angibt.
  ref(PROCESSOR) ASS_PROCESSOR - Verweis auf den zugehörigen Prozessor. Dieser enthält beispielsweise ein Feld ID vom Typ INTEGER, das ihn identifiziert, des weiteren beispielsweise Verweise auf die ACB-queue, den aktiven ACB, die auf den Prozessor geladenen script-Module etc.
  ref(HEAD) MAILBOX - Verweis auf die Mailbox des Agenten
  text CURRENT_FACET - Benennt die aktuelle Facette
  text CURRENT_OPER - Benennt die aktuelle Operation
  boolean TRACE - gibt an, ob der Agent tracen soll
  integer CURRENT_LINE - Aktuelle Zeile im Quellprogramm
  Des weitern sind unter anderem Verweise auf den Laufzeitkeller der Aktivierungssätze (damit besteht Zugriff auf alle Variablen eines Blocks mit deren Werten) und die verschiedenen Arbeits-stacks der Datentypen der abstrakten Maschine vorgesehen. Den Aufbau der Keller, weiterer Felder des ACB und anderer Kontrollblöcke, globaler Größen und Listen entnimmt man der Dokumentation (Teil E).

Als kleine **Anwendung** sei demonstriert, wie das **tracing** dynamisch ein- und ausgeschaltet werden kann und wie der **Name der aktuellen Facette** ausgedruckt werden kann.

Im CSSA-Programm definiert man an geeigneter Stelle etwa:

```
functionhead FACETNAME returns string external;
procedurehead TRACEON ( agent: A) external;
procedurehead TRACEOFF( agent: A) external;
```

Die Proceduren TRACEON und TRACEOFF können beispielsweise mit dem Parameter 'self' in einem CSSA-Programm aufgerufen werden.

Als externe Prozeduren definiere man:

```
text procedure CSSAFACETNAME;
CSSAFACETNAME:- Copy(Current qua MODULE.
  ASS_PROCESSOR.ACTIVE_ACB.CURRENT_FACET);

procedure CSSATRACEON(A); ref (ACB) A;
  A.TRACE:= true;

procedure CSSATRACEOFF(A); ref (ACB) A;
  A.TRACE:= false;
```

Die Realisierung der Funktion FACETNAME zeigt, wie unter Verwendung von Kontrollblöcken und Kenntnis über deren Felder und Verpointerung auf interne Steuergrößen zugegriffen werden kann. In ähnlicher Weise kann der Zugriff auf andere Daten des Laufzeitsystems und des Betriebssystemkerns erfolgen.

Mit Hilfe von externen Prozeduren und Funktionen kann auch die in CSSA nur rudimentär vorhandene Ein-/Ausgabe recht einfach erweitert werden. So zeigt das folgende Beispiel, wie das **Lesen von Daten aus Dateien** programmiert werden kann; eine Funktion, die in CSSA bisher noch nicht vorgesehen ist. Dieses Beispiel könnte in erweiterter Form dazu dienen, einen **E/A-Agenten** als Service-Agenten bereitzustellen und damit die Ein-/Ausgabe in das CSSA-System zu integrieren.

```
CSSA-code:

procedurehead OPENINPUT external;
 functionhead END_OF_FILE returns bool external;
 functionhead READREC returns string external;
procedurehead CLOSEINPUT external;
-------------------------------------------------
call OPENINPUT;
loop while not END_OF_FILE do
|   ...
|   ... READREC ...
|   ...
endloop;
call CLOSEINPUT;
```

```
SIMULA-code:

 ref (Infile) INPUTFILE;

 procedure CSSAOPENINPUT;
 begin INPUTFILE:- new Infile("BMSINPUT");
       INPUTFILE.Open(Blanks(80));
 end;

 text procedure CSSAREADREC;
 begin INPUTFILE.Inimage;
       CSSAREADREC:-INPUTFILE.Intext(80);
 end;

 boolean procedure CSSAEND_OF_FILE;
    CSSAEND_OF_FILE:=INPUTFILE.Endfile;

 procedure CSSACLOSEINPUT;
    INPUTFILE.Close;
```

Vor Aufruf des CSSA-Systems muß die Eingabedatei dem Linknamen BMSINPUT zugeordnet werden.

Sollen externe **Fortran-** oder **Assembler-Routinen** an CSSA-Programme angeschlossen werden, so muß zunächst - wie oben beschrieben - eine externe SIMULA-Prozedur bereitgestellt werden, die ihrerseits die externe Fortran- oder Assembler-Routine aufruft. Die externen Routinen werden als

- **external assembly** [<type>] **procedure** <name>

bzw.

- **external fortran** [<type>] **procedure** <name>

in diesen externen SIMULA-Prozeduren oder global in der externen Datei deklariert. Bezüglich möglicher Parametertypen und der Aufrufkonventionen sei auf die Kapitel 5.15 ("External Procedures and Classes"), 5.16 ("Assembly and Fortran Structures"), 13 ("Internal Representation of Data Structures") und 14 ("external Assembly or Fortran Procedures") des Handbuches "Siemens BS2000 Simula Programmer's Guide" (Mertens/Lattrell, Rechenzentrum der Universität Mannheim, 1981) verwiesen.

Falls **systemabhängige und hardwareabhängige Funktionen** realisiert werden sollen, so können vordefinierte externe Simula-Prozeduren verwendet werden, die in "External Procedure Library Siemens BS2000 Simula User's Guide" (Mertens/Lattrell, s.o.) beschrieben sind. Unter anderem besteht damit die Möglichkeit, BS2000 Betriebssystemkommandos abzusetzen, andere Object-Module zu laden und auszuführen und die Ein-/Ausgabe für Datensichtgeräte in spezifischer Weise zu programmieren.

4.9 Compiler-Restriktionen

Mögliche **Einschränkungen** an zu übersetzende CSSA-Programme ergeben sich daraus, daß zum einen der CSSA-Compiler in einer bestimmten Systemumgebung ausgeführt wird, die gewisse Grenzen aufweist (wie beispielsweise begrenzten Hauptspeicher), zum zweiten der der Compiler selbst einige (allerdings recht großzügig ausgelegte) statische Grenzen für die Größe von stacks und Speicherbereiche besitzt, und schließlich die vom Compiler generierten recht umfangreichen SIMULA-Programme den Restriktionen des SIMULA-Compilers unterliegen.

Implementierungsabhängig ist der Wertebereich bestimmter Datentypen. Diese Wertebereiche sind in B 6. (einfache Datentypen) aufgeführt, im wesentlichen handelt es sich um

- **REAL-Zahlen:** -16**63..16**63 mit einer Genauigkeit von 14 Hexadezimalziffern (entspricht ca. 16 Dezimalziffern)

- **INT-Zahlen:** -2**31..2**31-1

- **Strings:** EBCDIC-Zeichensatz, max 32748 Zeichen lang

Implementierungsrestriktionen der CSSA-Sprache wie beispielsweise die maximale Anzahl von Dimensionen eines arrays oder die Tatsache, daß verschiedene for/of-Schleifen für ein und denselben set nicht erlaubt sind, werden in Teil B (CSSA-Sprachbeschreibung), meist in den Unterkapiteln 4 der entsprechenden Sprachkonzepte, behandelt.

Die **durch den CSSA-Compiler festgelegten Restriktionen** an CSSA-Programme sind unwesentlich:

- Die Schachtelungstiefe von syntaktischen Einheiten (das heißt die maximale Tiefe des Ableitungsbaumes zur erweiterten BNF-Grammatik) ist begrenzt; eine genaue Aussage über die Grenze ist nicht möglich, eine Ineinanderverschachtelung von mindestens 30 Blöcken ist allerdings garantiert (bei extrem tief verschachtelten Strukturen erscheint die BMS044-Fehlermeldung).

- Unabhängig von der Verschachtelung können maximal 997 scripts übersetzt werden.

- Ein script darf höchstens 9999 verschiedene scopes, unabhängig von deren Verschachtelung, besitzen.

Die letzten beiden Einschränkungen dürften nie erreicht werden, da dann die generierten SIMULA-Programme so groß sind, daß diese nicht mehr vom SIMULA-Compiler übersetzt werden können. Um die erste Restriktion abzuändern, müssen eine Reihe von array-Grenzen im CSSA-Compiler erhöht werden (RECOVERY_TOKEN, PROC_NAME, BLOCK_NR etc. ——> [BMS-E]).

Die generierten **SIMULA-Programme** sind recht groß, aus einem CSSA-Programm wird ein SIMULA-Programm mit ca. 15 mal mehr Statements generiert. Bei großen CSSA-Programmen können daher Probleme bei der Übersetzung des generierten SIMULA-Programms

mit dem SIMULA-Compiler auftreten. In Bezug auf den SIMULA-Compiler sei auf den "Siemens BS2000 Simula Programmers Guide" (Mertens/Lattrell, Rechenzentrum der Universität Mannheim, 1981) verwiesen. In den Kapiteln 4.2 bis 4.4 werden dort zwei Restriktionen genannt, die von Bedeutung sein könnten:

- NUMBER OF DIFFERENTLY SPELLED INDETIFIERS, INDEPENDENT OF CORE AREA = 3072
  (Man beachte, daß das generierte SIMULA-Programm recht viele identifier, z.B. für labels, enthält. Bei zuvielen Bezeichnern erzeugt der SIMULA-Compiler eine Fehlermeldung mit der Nummer SIM10B, SIM120 oder SIM554.)

- MAXIMUM SIZE OF BLOCKINSTANCES: 4095 BYTE
  (Jeder Aktivierungssatz wird als class-Objekt repräsentiert. Bei Überschreiten des Bereichs erzeugt der SIMULA-Compiler die Meldung SIM565.)

Der verfügbare **Hauptspeicherplatz** kann sowohl die Übersetzung als auch die Ausführung von CSSA-Programmen beeinflussen. Für den CSSA-Compiler werden in jedem Fall mehr als 256K Byte benötigt; der SIMULA-Compiler reagiert bei zuwenig Speicherplatz evtl. mit den Meldungen SIM801 oder SIM802. Bei knappem Speicher können die Übersetzungszeiten wesentlich vergrößert werden. Sollte der Speicherplatz dem CSSA-Compiler nicht genügen, so wird das im allgemeinen durch die SIMULA-Laufzeitfehlermeldung

- ZYQ017 STORAGE EXHAUSTED

gemeldet. Auch bei einer evtl. "ZYQ077"-Meldung handelt es sich vermutlich um Speicherplatzprobleme. Eventuelle Fehlermeldungen des SIMULA-Compilers werden in die Datei CSSAMESS.<Name der source-Datei> geschrieben.

5. Die interaktive CSSA-Sitzung

CSSA ist als **interaktive** Programmiersprache konzipiert. Während einer interaktiven Sitzung kann der Benutzer eine **Teilmenge** der CSSA-Sprache verwenden, die **interpretativ** verarbeitet wird: Durch Gründen von Agenten kann ein Agentennetz aufgebaut werden; über Nachrichten kann der Benutzer mit den Agenten kommunizieren und so eine verteilte Berechnung starten. Darüberhinaus stehen spezielle Befehle zur **Steuerung des Simulationssystems** zur Verfügung. Eine dritte Gruppe von Kommandos dient zur Kontrolle des **interaktiven debuggers**, der eine Reihe von Zustandsinformationen über die Agenten und das Simulationssystem liefert, sowie unterschiedliche Grade der Systemprotokollierung - bis hin zum statementweisen tracing - erlaubt.

Im Anhang dieses Handbuchs ist ein ausführliches **Sitzungsprotokoll** abgedruckt, in dem alle Kommandos und Systemausgaben demonstriert werden.

5.1 Ablauf einer CSSA-Sitzung

Am Anfang einer interaktiven CSSA-Sitzung existiert zunächst nur ein vom System zur Verfügung gestellter, ausgezeichneter Agent, der sogenannte **interface-Agent.** Er verfügt nicht über ein fest definiertes script, sondern sein Verhalten kann vom Benutzer dynamisch gesteuert werden: Die interaktiv eingegebenen Kommandos werden interpretativ ausgeführt.

Die vom Compiler übersetzten scripts sind dem interface-Agenten bekannt. Er kann auf Anweisung des Benutzers aus den scripts neue Agenten erzeugen, Nachrichten verschicken und empfangen und so eine **verteilte Berechnung** durchführen. Allen anderen Agenten ist der interface-Agent durch das Schlüsselwort 'interface' bekannt.

Der Interpreter zeigt jeweils durch die Meldung 'ENTER CSSA COMMAND -' und einen Stern in Spalte 1 an, daß er auf Eingabe wartet. Die Eingabe darf nicht über eine Zeile hinausgehen und wird durch <DUE> oder <EM> <DUE> abgeschlossen. Im ersten Fall wird der Bildschirm gelöscht.

Es dürfen mehrere Kommandos in einer Zeile stehen. Jedes Kommando muß durch ein Semikolon abgeschlossen werden. Allerdings wird das Zeilenende vom scanner ebenfalls als Semikolon erkannt, so daß hinter dem letzten Befehl einer Zeile kein Semikolon stehen muß. Da das leere Statement zugelassen ist, darf dort aber ein Semikolon stehen.

Da das Zeilenende als Semikolon erkannt wird, kann ein Kommando nicht ohne weiteres über mehrere Zeilen gehen. Zeilen, die fortgesetzt werden sollen, müssen mit '#' abgeschlossen und eingegeben werden; der Rest dieser Zeile wird als Kommentar aufgefaßt. In der oder den folgenden Zeilen kann dann das Kommando fortgesetzt werden. Der Interpreter kennt eine Teilmenge des vom CSSA-Compiler akzeptierten **Zeichensatzes** (——> B.2). Sie umfaßt nur die folgenden Sonderzeichen: ; := :- : , ? - ( ) #

Für Bezeichner, Zahlenkonstanten und strings gelten die in B.2 beschriebenen Regeln unverändert.

Zwischen (* und *) dürfen **Kommentare** stehen, die aber nicht über eine Zeile hinausgehen dürfen. Sie sind vor allem dann sinnvoll, wenn die Kommandos aus einer Datei gelesen werden (s.u.).

Mit Hilfe des CSSA-Entwicklungssystems (——> 3.) können vor dem Beginn einer interaktiven CSSA-Sitzung speziell für den Interpreter geeignete **Funktionstasten** definiert werden, die die Benutzung der häufigsten Kommandos erleichtern:

P1: 'DISPLAY' <EINGABETASTE>
P2: 'STATUS' <EINGABETASTE>
P3: 'SYSSTATUS' <EINGABETASTE>
P4: 'VAR AGENT : '
P5: 'OPER : '
P6: 'PORT : '
P7: 'SEND '
P16: ';RUN' <EINGABETASTE>
P17: ';RUN' <EINGABETASTE> + Schirm löschen

Bei **Syntaxfehlern** gibt der Interpreter das fehlerhafte Symbol und die Menge aller stattdessen erlaubten Symbole aus. Die error-recovery Routine überliest die Eingabe bis zum nächsten Semikolon, bzw. bis zum Zeilenende. Man beachte, daß der Kommandointerpreter analysierte Kommandoteile teilweise sofort ausführt, noch bevor das Kommando vollständig analysiert ist. So wird beispielsweise bei einer fehlerhaften Initialisierung die Variable bereits deklariert und mit dem default-Wert belegt. Ein nochmaliges Eingeben der gesamten korrigierten Deklaration wird zu einem Deklarationsfehler führen.

Es ist weiterhin zu beachten, daß der Scanner zwei verschiedene Token für deklarierte und undeklarierte Bezeichner liefert, so daß fehlende Deklaration und Redeklaration als **Syntaxfehler** gemeldet werden. Bei einer Redeklaration ist beispielsweise das fehlerhafte Symbol <OLD-IDENTIFIER>, und in der Menge der erwarteten Symbole liegt <NEW-IDENTIFIER> (vgl. auch Punkt 6 aus 4.7.2).

Das **read-Kommando** ermöglicht es, Kommandofolgen anstatt vom Terminal aus einer Datei zu lesen. Die Datei muß vor Aufruf des Entwicklungssystems einem symbolischen file-Namen zugeordnet worden sein, der hinter 'read' genannt wird. Achtung: Fehlerhafte file-Namen führen zu einem Fehler, der vom CSSA-System nicht abgefangen werden kann. Die Kommando-Datei muß eine SAM-Datei (!) sein. Sie darf keine Kleinbuchstaben enthalten und die Satzlänge darf 72 nicht überschreiten. Die gelesenen Kommandos werden auf dem Bildschirm protokolliert. Bei Erreichen des Dateiendes oder eines fehlerhaften Kommandos wird die Eingabe zum Terminal zurückgeschaltet. Es kann auch durch ein erneutes read-Statement auf eine weitere Datei umgeschaltet werden, eine Rückkehr zur ersten Datei ist aber nicht möglich.

Beim Aufruf des Entwicklungssystems (——> 3.) kann durch die Angabe 'INIT=<Dateiname>' bewirkt werden, daß die angegebene Datei den file-Namen INIT erhält und sofort am Anfang der CSSA-Sitzung eingelesen wird. Sie kann auch nochmals mit 'READ INIT' gelesen werden.

Im CSSA-System gibt es für jeden Agenten eine **eindeutige Bezeichnung** bestehend aus Scriptname gefolgt von einer laufenden Nummer in Klammern. Beispielsweise bezeichnet PHILOSOPH(3) den Agenten, der als dritter zu dem script PHILOSOPH generiert wurde. Solche Standardbezeichnungen (<AGENT-DENOTATION>) werden in allen Systemmeldungen verwendet. In einer Reihe von Kommandos können auch Agenten, auf die der interface-Agent keine acquaintance durch eine Variable vom Typ AGENT hat, durch solche Standardbezeichnungen angesprochen werden.

Das **help-Kommando** gibt eine Liste aller vom Interpreter akzeptierten Kommandos aus.

Eine CSSA-Sitzung wird durch **'terminate'** beendet. Es werden zum Schluß einige Statistiken über die interaktive CSSA-Sitzung ausgegeben.

5.2 CSSA-Teilsprache

Der interface-Agent kann - wie jeder andere Agent - Agenten gründen, sowie Nachrichten versenden und verarbeiten. Der Interpreter kennt entsprechende Kommandos, die weitgehend der CSSA-Compilersprache entsprechen. Der interface-Agent verfügt auch über einen lokalen Speicher: Durch Deklarationen, Initialisierungen und Zuweisungen können Variablen angelegt und deren Wert verändert werden.

Da der interface-Agent allerdings nicht nach vordefinierten Algorithmen arbeitet, sondern seine Aktionen von Fall zu Fall vom Benutzer bestimmt werden, kennt der Interpreter keine komplexen Kontrollstrukturen wie Prozeduren, Funktionen oder Schleifen. Ebensowenig sind Datenstrukturen wie Mengen, Relationen, arrays oder records notwendig. Facetten und Operationen entfallen ebenfalls, stattdessen kann der Benutzer nach Belieben die mailbox inspizieren und Nachrichten löschen oder beantworten.

Im folgenden werden alle Kommandos der CSSA-Teilsprache des interface beschrieben. Dabei wird besonders auf Unterschiede zur Compilersprache hingewiesen. Ausführliche Beispiele finden sich in dem in Anhang A abgedruckten Sizungsprotokoll. Die komplette Syntax mit Erläuterungen zur Metasprache ist in Anhang B angegeben.

```
<CONST-VAR-DECL> ::= ( const|var ) <TYPE> ':'
                          <IDENTIFIER> || ','
                          [(':='|':-') <EXPRESSION>]
<TYPE> ::= bool|int|string|real|agent|oper|script
<ASSIGNMENT> ::= <IDENTIFIER> (':='|':-') <EXPRESSION>
<EXPRESSION> ::= <FACTOR>
<FACTOR> ::= ['-']
             ( <INT-CONST> | <REAL-CONST> | <STRING-CONST>
              | ownmode | self | interface
              | noscript | noagent | nooper
              | <IDENTIFIER>
              | new <SCRIPT-EXPRESSION> <MESSAGE>
             )
<DISPLAY-STATEMENT> ::= ( display | '?') [<FACTOR>]
```

Durch eine <CONST-VAR-DECL> können einfache Variablen oder Konstanten lokal im interface-Agenten angelegt werden. Als Datentyp sind nur die 7 Standard-Datentypen zulässig. Eine Initialisierung durch einen Ausdruck passenden Typs ist möglich.

Bei **Zuweisungen** wird ebenfalls die übliche Typgleichheit gefordert. Agenten werden mit ':-' zugewiesen, bei allen anderen Datentypen wird ':=' verwendet. Der Interpreter kennt nur sehr einfache Ausdrücke. Arithmetische und logische Berechnungen sind nicht möglich - aber wohl auch nicht notwendig. Neben Variablen und Konstanten der verschiedenen Datentypen (evtl. mit negativem Vorzeichen) sind auch Generatorausdrücke möglich: Mit 'new' können aus bekannten scripts neue Agenten erzeugt werden.

Durch **'display'** bzw. **'?'** kann die gesamte **Variablentabelle** des interface ausgegeben werden. Zu jedem Bezeichner werden Typ und Wert gezeigt. Alle Werte werdem im CSSA-Standardformat des jeweiligen Typs (——> B 16.) ausgegeben. Es kann auch gezielt der Wert einer bestimmten Variablen ausgegeben werden.

```
<OPER-DECL> ::= oper ':' <IDENTIFIER> || ','
<PORT-DECL> ::= port ':' <IDENTIFIER> || ','
<SEND-STATEMENT> ::= send <OPER-EXPRESSION> <MESSAGE>
                     [ to <AGENT-EXPRESSION>
                       [ reply to <IDENTIFIER> | inherit ]]
<MESSAGE> ::= '(' <EXPRESSION> || ',' ')'
          ::= <EMPTY>
<MAILBOX-STATEMENT> ::= mailbox
<DELETE-STATEMENT> ::= delete <INT-CONST>
<RECEIVE-STATEMENT> ::= receive <INT-CONST> [<PATTERN>]
<PATTERN> ::= '(' [<IDENTIFIER>] || ',' ')'
<REPLY-STATEMENT> ::= reply [<INT-CONST>] <MESSAGE>
```

Bevor Nachrichten versandt werden können, müssen zunächst die benötigten ports und Operationsbezeichner vereinbart werden.

Das **send-Statement** stimmt weitgehend mit dem entsprechenden Konstrukt aus der Compilersprache überein. Allerdings kann der 'to'-Teil weggelassen werden, wenn die Nachricht wiederum an denselben Zielagenten gesandt werden soll, wie schon die letzte Nachricht. Wird 'inherit' angegeben, so wird die Antwortverpflichtung der letzten mit 'receive' entgegengenommenen Nachricht weitervererbt.

Mit dem **mailbox-Statement** kann überprüft werden, welche Nachrichten dem interface-Agenten zugesandt wurden. Zu jeder Nachricht in der mailbox werden eine laufende Nummer, der Operationsbezeichner und zu allen Feldern der Nachricht Datentyp und Wert ausgegeben. Eventuell wird außerdem angegeben, für welchen port eine Antwortverpflichtung vorliegt. Die allgemeine Form des mailbox-Statement ist in 5.4 beschrieben.

Mit Hilfe des **delete-Statements** kann unter Angabe der laufenden Nummer eine Nachricht aus der mailbox gelöscht werden. Achtung: Durch das Löschen einer Nachricht verschieben sich die laufenden Nummern. Durch 'DELETE 1; DELETE 1' werden beispielsweise die beiden ersten Nachrichten aus der mailbox entfernt.

In der Regel wird es genügen, sich die empfangenen Nachrichten und die darin übermittelten Werte in der mailbox anzusehen. Es kann aber auch notwendig sein, mit einem **receive-Statement** den Wert eines Feldes einer Nachricht einer lokalen Variablen zuzuweisen. Hinter 'receive' steht die laufende Nummer der Nachricht. In dem eventuell folgenden pattern können lokale Variablen des interface oder aber auch leere Felder stehen. Beispielsweise werden durch 'RECEIVE 3 (,I,,J)' der zweite und vierte Parameter der dritten Nachricht an I und J zugewiesen. Bei mismatch erfolgt eine Fehlermeldung. Das pattern darf kürzer sein als die Nachricht. Wenn auf die empfangene Nachricht eine Antwort erwartet wird, erfolgt eine entsprechende Meldung.

Antwortnachrichten werden mit dem **reply-Statement** automatisch an den richtigen Agenten und an den richtigen port versandt. Wird hinter reply die laufende Nummer einer Nachricht genannt, so wird die Antwort auf die so bezeichnete Nachricht versandt. Wird keine Nummer angegeben, so bezieht sich das reply-Statement auf die zuletzt mit 'receive' entgegengenommene Nachricht.

Bemerkung: Über ownmode kann im Interpreter das dem interface-Agenten zugrundeliegende script bezeichnet werden. Es kann daher vom interface oder von irgendeinem anderen Agenten ein **zweiter interface-Agent** gegründet werden. Der zweite interface-Agent kennt ebenfalls alle vom Compiler übersetzten scripts. Er bewirbt sich wie alle anderen Agenten um die Prozessorzuteilung (---> 5.3) und meldet sich daher von Zeit zu Zeit mit 'INTERFACE(2): ENTER CSSA-COMMAND -'. Er ist dann bereit Kommandos zu interpretieren. Insbesondere kann durch ein run-Kommando (---> 5.3) der Verbrauch von Simulationszeit bewirkt werden und so die Kontrolle an das initiale interface zurückgegeben werden. Über das Schlüsselwort 'interface' wird stets der erste interface-Agent bezeichnet. Zusätzliche interface-Agenten eignen sich vor allem dazu, um in der Testphase die Rolle von noch nicht zur Verfügung stehenden Agenten zu übernehmen und so eine geeignete Umgebung für den Test der vorhandenen scripts zu

schaffen.

5.3 Steuerung des Simulationssystems

Das CSSA-System simuliert die Ausführung der Agenten auf einer Mehrprozessorkonfiguration. So können Informationen über den Grad der Parallelität einer verteilten Berechnung sowie über die Auslastung der simulierten hardware-Konfiguration gewonnen werden. Eine ausführliche Beschreibung des Simulationssystems findet sich im Teil A der CSSA-Dokumentation.

Vor Beginn einer interaktiven CSSA-Sitzung liest das Simulationssystem zunächst folgende Angaben über die gewünschte hardware-Konfiguration:

- **Die Anzahl der Prozessoren**
  Es können zunächst bis zu 10 Prozessoren angegeben werden. Sollen mehr als 10 Prozessoren simuliert werden, so ist eine geringfügige Änderung des Simulationssytems notwendig (array-Grenze erhöhen; siehe Teil E).

- **timeslice**
  Die Maximalzeit, die ein Agent rechnen kann, ohne von seinem Prozessor per timer-interrupt unterbrochen zu werden.

- **Die routing-Matrix**
  Diese Matrix ordnet je zwei Prozessoren einen Prozessor zu. 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, weil dorthin eine Busverbindung besteht. Wie man sieht, unterstützt das simulierte verteilte Betriebssystem ein sogenanntes **statisches routing:** Genau einer der möglichen Verbindungswege zwischen zwei Prozessoren wird bei der Systemgenerierung vorgegeben. Ein dynamisches routing in Abhängigkeit von der Auslastung der Kommunikationswege ist nicht vorgesehen.

- **Die Anzahl der Busse**
  Die Anzahl der Busse ist ebenfalls zunächst auf 10 beschränkt.

- **Die Übertragungszeit**
  Die Übertragungszeit der Busse für eine Nachricht kann gewählt werden. Man beachte dabei, daß pro CSSA-Statement 0.1 Sekunden Simulationszeit verbraucht werden. Die Übertragungszeit ist unabhängig von der Länge der Nachricht. Eine entsprechende Erweiterung des Simulationssystems wäre allerdings leicht möglich.

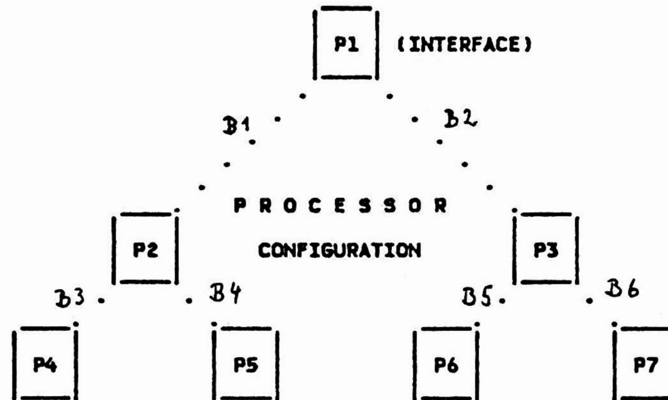
- **Die Abfertigungsstrategie**
  'F' bedeutet, daß die Nachrichten von den Bussen in der Reihenfolge ihres Eintreffens bearbeitet werden (FIFO). 'R' bedeutet, daß die Busse jeweils per Zufall eine Nachricht aus der Warteschlange auswählen (RANDOM), so daß Überholungen möglich sind, falls es wegen hoher Übertragungszeiten vor einem Bus zum Stau kommt.

- **Die connection-Matrix**
  Diese Matrix ordnet je zwei Prozessoren einen Bus zu. 0 bedeutet, daß keine Verbindung besteht. Bei der Spezifikation ist zu beachten, daß ein zusammenhängendes Netzwerk von Prozessoren konfiguriert werden muß. Ein Bus kann beliebig viele Prozessoren verbinden; beispielweise kann ein einziger Bus alle Prozessoren direkt verbinden. Können alle Busse in beide Richtungen Nachrichten übertragen, so ist die connection-Matrix symmetrisch bezüglich ihrer Hauptdiagonalen.

Die Benutzung einer speziellen, eigenen Konfiguration kann verlangt werden, indem bei Aufruf des Entwicklungssystems der Parameter 'CONFIG=<Dateiname>' angegeben wird. Die angegebene Datei muß eine SAM-Datei mit einer maximalen Satzlänge von 80 sein. Die Standard-Konfiguration steht in der Datei $SPENKE.BMS.CONFIG und sieht wie folgt aus:

==

C S S A - S I M U L A T I O N - S Y S T E M



==

7 (= ANZAHL DER PROZESSOREN)
1.0 (= TIMESLICE)

ROUTING MATRIX: PROZESSOR X PROZESSOR ---> PROZESSOR

	1	2	3	4	5	6	7
1	1	2	3	2	2	3	3
2	1	2	1	4	5	1	1
3	1	1	3	1	1	6	7
4	2	2	2	4	2	2	2
5	2	2	2	2	5	2	2
6	3	3	3	3	3	6	3
7	3	3	3	3	3	3	7

6 (= ANZAHL DER BUSSE)
0.1 (=UEBERTRAGUNGSZEIT FUER EINE MESSAGE)
R (=BUS_QUEUEING_DISCIPLINE R : RANDOM F: FIFO)

CONNECTION MATRIX: PROZESSOR X PROZESSOR ---> BUS

	1	2	3	4	5	6	7
1	0	1	2	0	0	0	0
2	1	0	0	3	4	0	0
3	2	0	0	0	0	5	6
4	0	3	0	0	0	0	0
5	0	4	0	0	0	0	0
6	0	0	5	0	0	0	0
7	0	0	6	0	0	0	0

Die ersten 25 Zeilen der Datei dienen lediglich als Kommentar und werden am Anfang der CSSA-Sitzung ausgegeben. Das Format der Eingabe muß genau eingehalten werden (insbesondere die Anzahl der Leerzeilen zwischen den relevanten Daten). Die Eingabe wird nicht auf Konsistenz geprüft (z.B. muß zwischen je 2 Prozessoren eine Verbindung bestehen). Da das interface stets alleine auf Prozessor 1 läuft, muß es mindestens zwei Prozessoren geben. Neu gegründete Agenten werden zyklisch auf die restlichen Prozessoren verteilt.

Neben der hardware-Konfiguration werden auch die wesentlichen Aspekte eines **verteilten Betriebssystems** simuliert. Nachrichten werden über mehrere Busse und Prozessoren weitergeleitet. Auf jedem einzelnen Prozessor läuft ein **lokales unterbrechungsgesteuertes multiprogramming-Betriebssystem**, das die quasi-parallele Ausführung mehrerer Agenten auf einem Prozessor ermöglicht.

Alle Agenten, die nicht im idle-Zustand sind, bewerben sich um die Prozessorzuteilung. Der aktive Agent benötigt pro CSSA-Statement 0.1 Sekunden simulierte Zeit. Allerdings wird nicht nach jedem Statement die simulierte Zeit verbraucht (durch den HOLD-Befehl von SIMULA), sondern erst vor Ausführung des nächsten Statements, das eine Auswirkung über den lokalen Zustand des Agenten hinaus haben wird, wie z.B. 'send', 'reply' oder 'print'. Das hat zur Folge, daß beim tracing (——> 5.4) teilweise bereits Aktionen protokolliert werden, die logisch noch in der Zukunft liegen und umgekehrt ein Agent aktiviert werden kann, ohne daß irgendeine Aktion protokolliert wird, da er lediglich Simulationszeit verbraucht.

Im folgenden sollen die **Kommandos** erläutert werden, die zur **Überwachung** und **Steuerung** des Simulationssystems dienen. Diese Kommandos wären bei einer CSSA-Implementierung auf einer realen Anlage nicht oder nur mit sehr großem Aufwand zu realisieren: Eine verteilte Berechnung auf einem realen System könnte nicht ohne weiteres zu jedem beliebigen Zeitpunkt angehalten werden; systemglobale Zustandsinformationen wären nur sehr schwer zu beschaffen.

```
<SYSSTATUS-STATEMENT> ::= sysstatus
<RUN-STATEMENT> ::= run [<INT-CONST> | <REAL-CONST>]
<LIMIT-STATEMENT> ::= limit <INT-CONST>
<SYSTEM-STATEMENT> ::= system
<NOSYSTEM-STATEMNENT> ::= nosystem
<TIME-STATEMENT> ::= time
<STOP-STATEMENT> ::= stop <AGENT-DENOTATION> || ','
<START-STATEMENT> ::= start <AGENT-DENOTATION> || ','
<AGENT-DENOTATION> ::= <IDENTIFIER> [ '(' <INT-CONST> ')' ]
                   ::= self | interface
```

Das **sysstatus-Kommando** liefert eine Übersicht über alle Prozessoren und Busse.

Pro Prozessor:

- die Auslastung in %
- die Queue der Agenten, die sich um die Prozessorzuteilung bewerben (durch * markiert).
- die Queue der Agenten, die zur Zeit im idle-Zustand sind

Pro Bus:

- die Auslastung in %
- die Anzahl der bereits übertragenen Nachrichten
- die durchschnittliche Länge der Warteschlange vor dem Bus (genauer: die durchschnittliche Länge der Queue bei Eintreffen einer Nachricht. 0.00 heißt daher, daß jede Nachricht sofort bearbeitet wurde)
- Die Operationsbezeichner aller wartenden Nachrichten

Während der interface-Agent Kommandos verarbeitet, bleibt die simulierte Zeit stehen, so daß keiner der anderen Agenten rechnet. Erst bei bei Eingabe von **'run'** gibt der interface-Agent die Kontrolle ab (indem er selbst simulierte Zeit verbraucht), und die anderen Agenten können arbeiten. Sobald das Agentennetz terminiert, geht die Kontrolle wieder an den interface-Agenten.

Um die Simulation gezielt bis zu einem bestimmten Zeitpunkt oder einer interessanten Konstellation ablaufen lassen zu können, gibt es verschiedene Möglichkeiten, die Kontrolle schon früher an das interface zurückzugeben:

- Es kann hinter 'run' eine Maximalzeit in Sekunden angegeben werden, die simuliert werden soll (z.B. 'run 1' oder 'run 1.5'). Wenn nach der angegebenen Simulationszeit das Netz nicht terminiert ist, erhält das interface die Kontrolle.
- Es wurden mehr als 5 Sekunden Realzeit für 0.1 Sekunden Simulationszeit gebraucht. Das ist beispielsweise der Fall, wenn man nicht sofort weiterblättert, wenn ein Schirm vollgeschrieben ist. Man kann auch jederzeit mit der K2-Taste unterbrechen und auf die Frage 'DO YOU WANT TO TERMINATE PROC-MODE? (Y/N)' erst nach 5 Sekunden mit 'N' antworten. Man hat so die Möglichkeit, durch eine geeignete Form der Protokollierung den Fortgang der verteilten Berechnung zu beobachten und dann zu einem beliebigen Zeitpunkt zu unterbrechen, um den Systemzustand genauer zu untersuchen.

Die mit ';RUN' belegten Funktionstasten (——> 5.1) stellen daher quasi spezielle enter-Tasten dar, mit denen nicht nur ein Befehl an den Interpreter abgeschickt wird, sondern auch die Simulation in Gang gesetzt wird.

Mit dem **limit-Statement** kann die mit 5 Sekunden vorbesetzte Realzeitgrenze verändert werden. Vor allem, wenn die Kommandos aus einer Datei gelesen werden, ist es sinnvoll, die Grenze sehr hoch zu setzen und so einen vorzeitigen Abbruch zu verhindern (z.B. 'LIMIT 10000').

Durch Eingabe von **'system'** wird bewirkt, daß von nun an die wichtigsten Aktionen des verteilten Betriebssystems protokolliert werden (Aktivieren eines Agenten, Deaktivieren eines Agenten, Übermittlung von Nachrichten, ...). Die Meldungen werden durch '///' gekennzeichnet. Durch **'nosystem'** kann die Protokollierung wieder ausgeschaltet werden.

Mit dem **time-Statement** werden die bisher in der CSSA-Session verbrauchte Realzeit, Simulationszeit und CPU-Zeit angezeigt.

Mit Hilfe des **stop-Statements** kann die Ausführung eines Agenten, der z.B. nicht das gewünschte Verhalten zeigt, gewaltsam abgebrochen werden. Der angegebene Agent wird nach der nächsten Unterbrechung aus der Warteschlange vor seinem Prozessor entfernt und kann daher nicht mehr weiterrechnen. In der STATUS-Tabelle (——> 5.4) taucht er nicht mehr auf. Das stop-Kommando ist beispielsweise dafür geeignet, Agenten, die in einer Endlosschleife sind, zu stoppen. Allerdings können Endlosschleifen nur abgebrochen werden, wenn wenigstens ein Statement vorkommt, vor dem Simulationszeit verbraucht wird. Mit dem **start-Statement** können zuvor gestoppte Agenten wieder gestartet werden. Man beachte, daß durch Angabe einer Standardbezeichnung (——> 5.1) jeder beliebige Agent gestoppt werden kann..

5.4 Der interaktive debugger

Für **Test** und **Entwicklung** von verteilten CSSA-Anwendungen steht ein in das Laufzeitsystem integriertes, **interaktives debugging-System** zur Verfügung. Auf interaktiv eingegebene Kommandos hin werden detaillierte Informationen über den Zustand einzelner Agenten oder des gesamten Agentennetzes ausgegeben. Darüberhinaus können die Aktionen aller oder einzelner Agenten in vielen unterschiedlichen Detaillierungsgraden protokolliert werden, so daß bei der Entwicklung von CSSA-Anwendungen das Einbringen von Testausgaben in den script-code überflüssig ist. Das debugging wird ausschließlich **dynamisch** gesteuert, irgendwelche Vorkehrungen bei der Übersetzung von scripts (wie z.B. die Angabe bestimmter Compilerparameter) sind nicht notwendig.

Darüberhinaus bietet die Tatsache, daß die verteilte Berechnung auf einer **simulierten** Konfiguration durchgeführt wird, einen entscheidenden Vorteil beim Programmtest: Obwohl CSSA eine **nicht-deterministische** Programmiersprache ist, sind die Simulationsläufe (und vor allem die aufgetretenen Fehler) **reproduzierbar**. Der Nicht-Determinismus findet nämlich durch Zufallszahlen in das Simulationssystem Eingang. Bei einer wiederholten Ausführung einer CSSA-Sitzung (mit identischer Eingabe) werden aber dieselben Zahlen erneut ausgelost. In einer realen Implementierung hingegegen könnte bereits ein veränderter Zeitabstand zwischen dem Versenden zweier Nachrichten vom interface

aus zu einem gänzlich anderen Verlauf der verteilten Berechnung führen.

Es wurde strikt darauf geachtet, daß alle debugging-Informationen nur in Termen ausgegeben werden, die für den **Benutzer** verständlich sind. Es werdem beispielsweise stets symbolische Namen und Zeilennummern aus dem Quellprogramm verwendet und nicht Adressen im generierten code.

In allen nun folgenden Kommandos zur Steuerung des debuggers können nicht nur die dem interface durch eine AGENT-Variable bekannten Agenten genannt werden, sondern über die Standardbezeichnungen (——> 5.1) können **alle** Agenten angesprochen werden.

Zunächst wird erläutert, welche **Zustandsinformationen** der debugger liefern kann.

```
<AGENT-DENOTATION> ::= <IDENTIFIER> [ '(' <INT-CONST> ')' ]
                   ::= self | interface
<STATUS-STATEMENT> ::= status
<MAILBOX-STATEMENT> ::= mailbox [<AGENT-DENOTATION>]
<DUMP-STATEMENT> ::= dump <AGENT-DENOTATION>
<STACKS-STATEMENT> ::= stacks <AGENT-DENOTATION>
```

Mit **'status'** wird eine tabellarische Übersicht über alle existierenden Agenten ausgegeben. Pro Agent werden folgende Informationen angezeigt:

- auf welchem Prozessor der Agent läuft
- ob sich der Agent um die Prozessorzuteilung bewirbt (durch * markiert)
- die aktuelle Facette des Agenten
- evtl. die aktuelle Operation
- die Operationsbezeichner aller Nachrichten in der mailbox. Wenn der Platz reicht, wird auch der Inhalt der Nachrichten gezeigt.

Mit dem mailbox-Statement kann die mailbox eines beliebigen Agenten gezeigt werden. Zu jeder Nachricht werden Operationsbezeichner sowie zu jedem Feld Typ und Wert ausgegeben. Wird hinter 'mailbox' kein Agent benannt, so wird die mailbox des interface gezeigt.

Mit **'dump'** kann der gesamte **lokale Speicher** eines Agenten ausgegeben werden. Zu jedem aktiven Block wird der Inhalt des zugehörigen Aktivierungssatzes gezeigt. Zunächst erscheinen einige Informationen, die zu jedem Block gehören:

- Name des Blockes
- Verweis auf den statisch umgebenden Block
- Zeilennummer, bis zu der der Block abgearbeitet ist

Es folgen zu jeder lokalen Variablen des Blocks der symbolische Name und der Wert im jeweiligen CSSA-Standardformat. Auch der Wert von komplexen Datenstrukturen wie records, arrays, sets und Relationen wird ausgegeben. Das dump-Statement macht Kontrollausdrucke praktisch überflüssig, da man jederzeit jede beliebige Variable im gesamten Agentennetz inspizieren kann.

Das **stacks-Statement** ist in erster Linie für den Systementwickler selbst gedacht, seine Existenz soll aber hier nicht verschwiegen werden. Es gibt zu einem beliebigen Agenten den Inhalt der für jeden Datentyp vorhandenen stacks der virtuellen Zwischencodemaschine aus.

Weitere Zustandsinformationen liefern das display-Kommando (——> 5.2) und das sysstatus-Kommando (——> 5.3).

Es folgt nun die Beschreibung der Kommandos, mit denen der Detaillierungsgrad der **Protokollierung** gesteuert werden kann.

```
<PROT-STATEMENT> ::= prot
<NOPROT-STATEMENT> ::= noprot
<OBSERVE-STATEMENT> ::= observe
<NOOBSERVE-STATEMENT> ::= noobserve
<EVENT-STATEMENT> ::= event [ stop ]
<NOEVENT-STATEMENT> ::= noevent
<SNAPSHOT-STATEMENT> ::= snapshot <AGENT-DENOTATION> || ','
<NOSNAPSHOT-STATEMENT>
                    ::= nosnapshot <AGENT-DENOTATION> || ','
<TRACE-STATEMENT> ::= trace <AGENT-DENOTATION> || ','
<NOTRACE-STATEMENT> ::= notrace <AGENT-DENOTATION> || ','
```

Durch Angabe von **'prot'** wird bewirkt, daß alle Meldungen, die auf den Bildschirm gelangen, auch in einer Datei protokolliert werden, die auf Wunsch nach einer CSSA-Sitzung gedruckt wird. Durch **'noprot'** wird die Protokollierung wieder abgeschaltet. In einer Sitzung kann mehrfach die Protokollierung an- und abgeschaltet werden und so gezielt interessante Passagen der Sitzung festgehalten werden.

Alle CSSA-Meldungen enthalten die simulierte Zeit und sind chronologisch sortiert. Die eingegebenen Kommandos sind im Protokoll durch '==>' markiert, alle CSSA-Standardmeldungen durch '>>>'. Meldungen, die aufgrund des system-Kommandos ausgegeben werden, beginnen mit '///' und solche, die aufgrund des observe-Kommandos (s.u.) ausgegeben wurden, mit '+++'. Schließlich werden Zeilen, die aufgrund eines expliziten print-Statement (——> B.16) von einem Agenten ausgegeben werden, durch '***' markiert.

Das Einschalten der **observe-option** bewirkt, daß die wichtigsten Aktionen aller Agenten (Versenden / Empfangen von Nachrichten, Facettierung, Erzeugen neuer Agenten, ...) gemeldet werden. Man erhält so einen hervorragenden Überblick über den Fortgang der verteilten Berechnung. Durch 'noobserve' kann die option wieder ausgeschaltet werden. Man vergleiche auch die system-option, die detaillierte Informationen über das verteilte Betriebssystem liefert (——> 5.3).

Mit Hilfe der **event-option** kann erreicht werden, daß bei jedem wichtigen Ereignis (Versenden / Empfangen von Nachrichten, Erzeugen eines neuen Agenten) die status-Information (s.o.) 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 eine vorangestellten Pfeil ('===>') gekennzeichnet wird, erhält der Benutzer einen recht plastischen Eindruck von der Kommunikation im Agentennetz. Wird hinter 'event' noch 'stop' angegeben, so wartet das System nach jeder Tabelle auf eine Quittung des Benutzers durch Drücken der Eingabetaste. Durch 'noevent' wird die option wieder ausgeschaltet.

Die **snapshot-option** ermöglicht es, die Veränderungen am lokalen Speicher eines Agenten genauestens zu verfolgen. Jedesmal wenn ein neuer Block angelegt und initialisiert wurde - also beispielsweise beim Start einer Operation - wird der gesamte lokale Speicher des Agenten gezeigt (vgl. dump-Statement).

Die detaillierteste Form der Protokollierung ist das **tracing.** Jeder einzelne Befehl eines Agenten wird unter Angabe der Zeilennummer genauestens protokolliert. Unter anderem werden die folgenden Ereignisse gemeldet:

- Facettierung
- Auswahl einer Nachricht aus der mailbox
- Zuweisungen während eines pattern-match
- Start einer Operation
- Zuweisungen von Werten an einfache Variablen oder Komponenten von arrays und records
- Einfügen oder Entfernen von Mengenelementen
- Einfügen oder Löschen von records in Relationen
- Aufruf von Prozeduren und Funktionen
- Übergabe von Parametern
- Gründen eines neuen Agenten
- Versenden einer Nachricht

- Verlassen oder Fortsetzen von Schleifen
- Erreichen des idle-Zustandes
- Terminierung

Jedesmal wenn der Agent vom multiprogramming-Betriebssystem aktiviert wird, beginnt ein Abschnitt des tracings. Der Beginn eines solchen Abschnitts ist im Protokoll durch eine waagerechte Linie gekennzeichnet, auf der die Simulationszeit, die Bezeichnung des Agenten sowie evtl. die aktuelle Facette und die aktuelle Operation angegeben sind. Der tracing-Abschnitt ist beendet, wenn der Agent aufgrund einer Unterbrechung wieder deaktiviert wird. Das Ende des Abschnitts ist ebenfalls durch eine waagerechte Linie markiert. Innerhalb des auf diese Weise optisch hervorgehobenen Abschnitts sind die einzelnen Aktionen des Agenten während seiner Aktivierung protokolliert. Dabei ist zu beachten, daß der Agent nur simulierte Zeit verbraucht, bevor er ein Statement ausführt, welches nicht nur Auswirkungen auf den lokalen Speicher des Agenten hat, wie z.B 'send', 'reply' oder 'print'. Der Agent kann daher nur unmittelbar vor einem solchen Statement deaktiviert werden, und es wurden dann evtl. bereits Aktionen protokolliert, die logisch noch in der Zukunft liegen, da für sie noch keine Simulationszeit verbraucht wurde. Bei der nächsten Aktivierung des Agenten werden dann womöglich gar keine Aktionen protokolliert: Der Agent verbraucht lediglich Simulationszeit für bereits ausgeführte Aktionen.

Zu allen in diesem Kapitel erklärten Kommandos finden sich **Beispiele** in dem im Anhang abgedruckten Sitzungsprotokoll. Weitere Protokolle zu ausgewählten CSSA-Anwendungen sind im Teil D der CSSA-Dokumentation zu finden.

5.5 Fehlerbehandlung

Während einer interaktiven CSSA-Sitzung können verschiedenartige Fehlerbedingungen auftreten, die sich folgendermaßen klassifizieren lassen:

- a) interface-Fehler
- b) Laufzeitfehler
- c) Abbruchfehler

Die **interface-Fehlermeldungen** werden vom CSSA-Interpreter als Reaktion auf eine fehlerhafte interaktive Eingabe generiert. Dazu gehören die **lexikalischen Fehler** (falsche Zeichen wie bspw. Kleinbuchstaben oder nicht vorgesehene Sonderzeichen, fehlende rechte Kommentarklammer oder Stringbegrenzung und Fehler in Zahlenkonstanten), die **Syntaxfehler** (Verletzung der Syntax, die durch die im Anhang angegebene kontextfreie Grammatik bestimmt ist) und die **statischen Semantikfehler**.

Die Reaktion des Interpreters auf Syntaxfehler ist in Kap. 5.1 beschrieben. Zu den **Semantikfehlern** gehören:

- Falsche Nachrichtennummer bei 'reply', 'delete' oder 'receive'.

- Bei 'receive' hat eine Nachricht weniger Werte als das angegebene pattern Parameter besitzt.

- Bei einer <AGENT-DENOTATION> (im trace-Statement, dump-Statement, mailbox-Statement etc.) existiert der Agent nicht oder die angegebene Variable hat den Wert NOAGENT.

- falscher Typ (so muß bspw. bei pattern-Elementen Übereinstimmumg bezüglich des Typs mit den ankommenden Werten der Nachricht herrschen, oder bei "send A to B reply to C" muß es sich bei A um OPER-, bei B um AGENT- und bei C um PORT-Ausdrücke handeln etc.).

Die Sematikfehler werden ebenfalls am Bildschirm des interface (und ggf. im Protokoll) gemeldet. Wie bei Syntaxfehlern ist auch hier u.U. zu beachten, daß der Interpreter eingegebene Kommandoteile möglichst frühzeitig evaluiert, so daß bei fehlerhaften Befehlen (korrekte) Teile davon bereits ausgeführt sein können (——> 5.1).

Während des Verlaufs einer verteilten Berechnung können lokal in Agenten oder im Kommunikationssubsystem des Agentennetzes **Ausnahmebedingungen** in Form von Laufzeitfehlern und Abbruchfehlern entstehen. Ein exception-handling Konzept ist in der vorliegenden Implementierung nicht vorgesehen. Auf auftretende Ausnahmebedingungen kann daher nicht beliebig reagiert werden, statt dessen ist die Reaktion des Systems fest vorgegeben.

Laufzeitfehler führen i.a. nicht zu einer Terminierung des verursachenden Agenten oder gar des gesamten Netzes, sondern werden nach der Fehlermeldung durch eine naheliegende Korrektur übergangen. Es hat sich gezeigt, daß so Testläufe meist auch nach Laufzeitfehlern sinnvoll fortgesetzt werden können, wobei evtl. weitere Fehler entdeckt werden.

Eine Laufzeitfehlermeldung hat folgendes Format, das den übrigen Systemmeldungen (——> 5.4) entspricht:

XXX <Simzeit> <Agent>: <Fehlertext> AT LINE <nnnn>

Nach dieser Meldung wird der gesamte lokale Zustand des verursachenden Agenten ausgegeben (vgl. dump-Statement Kap. 5.4).

Folgende Laufzeitfehlermeldungen sind vorgesehen:

- ZERODIVIDE. DIVISION IGNORED. (Die Division wird nicht ausgeführt, d.h. das Ergebnis entspricht einer Division durch 1.)

- SEND TO <NOAGENT> ISSUED. NO MESSAGE SENT. (Das Statement ist wirkungslos; diese Meldung wird auch generiert, wenn 'reply' ohne vorliegende Antwortverpflichtung ausgeführt

wird.)

- SEND TO TERMINATED AGENT ISSUED. (Das send-Statement ist wirkungslos.)

- TARGET AGENT TERMINATED. (Während die Nachricht unterwegs war, ist der Zielagent terminiert. Die Nachricht geht verloren, der verursachende Agent wird nicht gedumpt.)

- NEW <NOSCRIPT> ISSUED. NOAGENT RETURNED. (Die SCRIPT-Expression nach 'new' wurde zu NOSCRIPT evaluiert. Als Ergebnis wird der AGENT-Wert NOAGENT geliefert.)

- LOWER-BOUND > UPPER-BOUND AT ARRAY GENERATION. (Bei der dynamischen Deklaration eines arrays ist die obere Grenze kleiner als die untere. Als Korrekturmaßnahme wird sie gleich der unteren gesetzt.)

- ARRAY BOUNDS ERROR. (Bei einem array-Zugriff liegt der Indexwert außerhalb des bei der Deklaration angegebenen Bereiches. Als Korrekturmaßnahme wird der Index gleich der unteren Grenze gesetzt.)

- RETURN MISSING. (Bei einer ARRAY- RECORD- oder SET-liefernden Funktion wurde kein 'return' ausgeführt. Nach der Fehlermeldung wird die CSSA-Sitzung abgebrochen.)

- ZERODIVIDE IN BUILT-IN FUNCTION MOD. (Es wurde mod(...,0) ausgeführt. Als Korrekturmaßnahme wird 0 zurückgeliefert.)

- ILLEGAL ARGUMENT TO FUNCTION LN. (Die Funktion ln wurde mit einem Argument <= 0.0 aufgerufen. Als Korrekturmaßnahme wird 0.0 zurückgeliefert.)

- ILLEGAL ARGUMENT TO FUNCTION SQRT. (Die Funktion sqrt wurde mit einem Argument < 0.0 aufgerufen. Als Korrekturmaßnahme wird 0.0 zurückgeliefert.)

- OVERFLOW IN BUILT-IN FUNCTION ENTIER. (Der Wert der als Parameter übergebenen REAL-Zahl lag außerhalb des Bereichs für INT-Zahlen. Als Korrekturmaßnahme wird 0 zurückgeliefert.)

- OVERFLOW IN BUILT-IN FUNCTION ROUND. (Der Wert der als Parameter übergebenen REAL-Zahl lag außerhalb des Bereichs für INT-Zahlen. Als Korrekturmaßnahme wird 0 zurückgeliefert.)

Einige (seltene) Fehler führen zum Abbruch der gesamten CSSA-Sitzung. Diese **Abbruchfehler** äußern sich dadurch, daß auf dem Bildschirm des interface vor Abbruch der Sitzung eine Laufzeitfehlermeldung des SIMULA-Systems erscheint. Zu diesen Fehlern gehören:

- **Bereichsüberschreitung bei INT, REAL oder STRING** (——> B 6).

- Beim Potenzieren: Basis Null und nicht positiver Exponent, oder negative Basis bei REAL-Exponentiation (——> B 6.1 und B 6.2).
- Argumente der built-in Funktionen sin, cos oder tan sind größer als PI*2**50 (——> B 6.2)
- 'read' mit fehlerhaftem file-Namen (——> 5.1).
- Falsche Organisationsform externer Dateien (read, init-Kommandodatei etc.) (——> 5.1, 5.3)
- Laufzeitfehler in eigenen externen SIMULA-Routinen (——> 4.8).
- Zu wenig dynamischer Speicherplatz (——> 3.).
- Interne Fehler des SIMULA-Laufzeitsystems (vermutlich im garbage-collector). Diese treten manchmal "nicht-deterministisch" auf; sie äußern sich in der Meldung "ZYQ077 OBJECT NONE AT LINE 0000". Unter Umständen schafft die Angabe von mehr Speicherplatz (——> 3.) hier Abhilfe.

Anhang A: Sitzungsprotokolle

Dieser Anhang ist im wesentlichen eine Ergänzung zu Kapitel 5. Die dort beschriebenen Interpreter-Kommandos werden hier im Rahmen vollständiger interaktiver CSSA-Sitzungen demonstriert.

Im ersten Protokoll werden die Kommandos der CSSA-Teilsprache verwendet. Es wird ein zweiter interface-Agent gegründet, der eine Nachricht vom ersten interface empfängt und beantwortet. Darüberhinaus werden je ein Syntax- und ein Semantikfehler hervorgerufen.

Im zweiten Protokoll wird demonstriert, wie vom interface-Agenten aus ein Agentennetz aufgebaut und überwacht und eine verteilte Berechnung initiert und kontrolliert wird. Besonderer Wert wird dabei auf die möglichen Zustandsinformationen und die verschiedenen Formen der Systemprotokollierung gelegt.

In der Sitzung wird zunächst ein gerichteter Graph aus gleichartigen Agenten (siehe script NODE in diesem Anhang) aufgebaut, indem jedem Agenten die Liste seiner Nachbarn zugeschickt wird (operation KNOW). Die Gesamtheit der Agenten führt einen (verteilten) Algorithmus (Echo-Algorithmus) aus, dessen Ziel es ist, alle Knoten des Graphen zu markieren und anschließend eine Erfolgsmeldung (Echo) zurückzusenden.

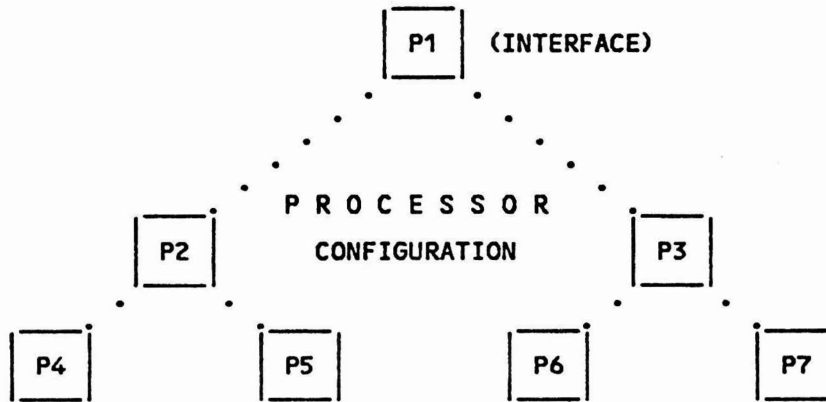
Im nicht markierten Zustand (facet NOT_MARKED) stellen alle Agenten die Operation MARK zur Verfügung. Je nachdem, ob von dem Knoten noch weitere Kanten ausgehen, sendet er sofort das Echo zurück oder schickt MARK-Nachrichten entlang aller ausgehenden Kanten. In jedem Falle wechselt der Agent in die Facette MARKED.

Der markierte Agent wartet nun auf alle ausstehenden Echos, um dann seinerseits das Echo zurückzusenden. Weitere eintreffende MARK-Nachrichten können sofort mit einem Echo beantwortet werden.

Die Berechnung wird vom interface aus durch eine MARK-Nachricht an die Wurzel (des spannenden Baums) des Graphen initiert. Sie ist beendet, wenn beim interface-Agenten das Echo eintrifft.

==

C S S A - S I M U L A T I O N - S Y S T E M



==

```
         PROGRAM GENERATED ON 1982/05/23 AT 20:32:59.00
         BY BMS-CSSA-COMPILER (VERS. 30 SEP 1981)

         PROTOCOL OF CSSA SESSION ON 1982/05/23 AT 21:28:43.00
         =====================================================

>>>      0.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> OBSERVE                                                         <==
>>>      0.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> VAR AGENT : IF2:=NEW OWNMODE                                    <==
+++      0.000  P1: INTERFACE(1) CREATES INTERFACE(2)()
>>>      0.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> RUN 1                                                           <==
>>>      0.000  INTERFACE(2) : ENTER CSSA COMMAND -
==> ;DISPLAY                                                        <==
```

IDENTIFIER	TYPE	VALUE
NODE	SCRIPT	NODE

```
>>>      0.000  INTERFACE(2) : ENTER CSSA COMMAND -
==> ;STATUS                                                         <==
+++      0.000  ALL EXISTING AGENTS:
```

AGENT	FACET	OPERATION	MAILBOX
P1: INTERFACE(1) *			
P2: INTERFACE(2) *			

```
>>>      0.000  INTERFACE(2) : ENTER CSSA COMMAND -
==> RUN 1                                                           <==
>>>      1.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> OPER: HALLO;PORT : P                                            <==
>>>      1.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> SEND HALLO (5,TRUE,SELF) TO IF2 REPLY TO P                      <==
+++      1.000  P1: INTERFACE(1) SENDS HALLO(INT:5,BOOL:TRUE,AGENT:
                           INTERFACE(1)),REPLY TO: P TO INTERFACE(2)
>>>      1.000  INTERFACE(2) : ENTER CSSA COMMAND -
==> MAILBOX                                                         <==
MAILBOX OF INTERFACE(2) :
____________________________________________________________
____________________________________________________________

>>>      1.000  INTERFACE(2) : ENTER CSSA COMMAND -
==> RUN 0.8                                                         <==
>>>      1.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> RUN 1                                                           <==
>>>      1.100  REAL-TIME LIMIT EXCEEDED
>>>      1.100  INTERFACE(2) : ENTER CSSA COMMAND -
==> RUN 1                                                           <==
>>>      2.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> RUN 1                                                           <==
+++      2.000  P2: INTERFACE(2) RECEIVES HALLO(INT:5,BOOL:TRUE,AGENT:
                           INTERFACE(1)),REPLY TO: P FROM INTERFACE(1)
>>>      2.100  INTERFACE(2) : ENTER CSSA COMMAND -
==> MAILBOX                                                         <==
MAILBOX OF INTERFACE(2) :
____________________________________________________________
```

```
(1) HALLO(INT:5,BOOL:TRUE,AGENT:INTERFACE(1)),REPLY TO: P
```

```
>>>      2.100  INTERFACE(2) : ENTER CSSA COMMAND -
==> VAR AGENT : X                                                        <==
>>>      2.100  INTERFACE(2) : ENTER CSSA COMMAND -
==> RECEIVE 1 (,,X)                                                      <==
PLEASE REPLY TO INTERFACE(1)
USING THE <REPLY_STATEMENT>

>>>      2.100  INTERFACE(2) : ENTER CSSA COMMAND -
==> ?X                                                                   <==
INTERFACE(1)
>>>      2.100  INTERFACE(2) : ENTER CSSA COMMAND -
==> REPLY(99)                                                            <==
+++      2.100  P2: INTERFACE(2) SENDS *REPLY(INT:99) TO INTERFACE(1)
>>>      2.100  INTERFACE(2) : ENTER CSSA COMMAND -
==> RUN 2                                                                <==
>>>      3.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> RUN 1.2                                                              <==
+++      3.000  P1: INTERFACE(1) RECEIVES *REPLY(INT:99) FROM INTERFACE(2)
>>>      3.100  REAL-TIME LIMIT EXCEEDED
>>>      3.100  INTERFACE(2) : ENTER CSSA COMMAND -
==> RUN 1                                                                <==
>>>      4.100  INTERFACE(2) : ENTER CSSA COMMAND -
==> RUN 1                                                                <==
>>>      4.200  INTERFACE(1) : ENTER CSSA COMMAND -
==> MAILBOX                                                              <==
MAILBOX OF INTERFACE(1) :
```

```
(1) *REPLY(INT:99)
```

```
>>>      4.200  INTERFACE(1) : ENTER CSSA COMMAND -
==> DELETE 1                                                             <==
>>>      4.200  INTERFACE(1) : ENTER CSSA COMMAND -
==> MAILBOX                                                              <==
MAILBOX OF INTERFACE(1) :
```

```
>>>      4.200  INTERFACE(1) : ENTER CSSA COMMAND -
==> (* SYNTAX-FEHLER *)#                                                 <==
==> VAR INT I                                                            <==

==> SYNTAX-ERROR : ILLEGAL SYMBOL 'I'
    EXPECTED SYMBOLS: ':'
>>>      4.200  INTERFACE(1) : ENTER CSSA COMMAND -
==> VAR INT: I                                                           <==
>>>      4.200  INTERFACE(1) : ENTER CSSA COMMAND -
==> (* SEMANTIK-FEHLER *)#                                               <==
==> DUMP I                                                               <==

SEMANTIC ERROR :
I IS NOT OF TYPE AGENT
>>>      4.200  INTERFACE(1) : ENTER CSSA COMMAND -
==> TERMINATE                                                            <==
+++      4.200  ALL EXISTING AGENTS:
```

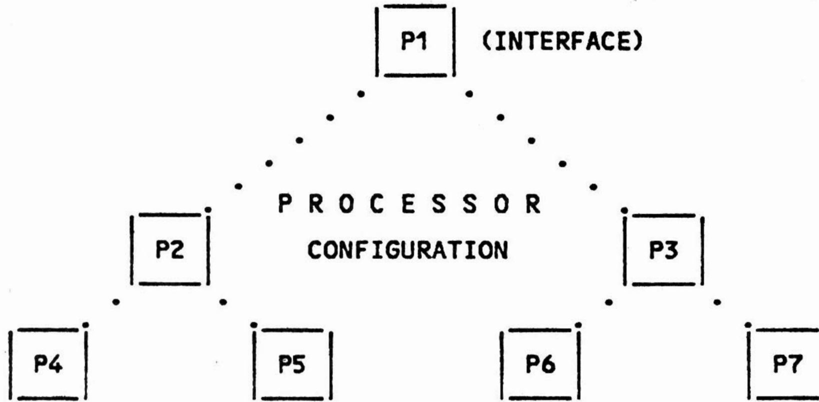
AGENT	FACET	OPERATION	MAILBOX
P1: INTERFACE(1) *			
P2: INTERFACE(2) *			HALLO

```
                 CSSA-SESSION-STATISTICS
                 =======================

SESSION  STARTED  AT  21:28:26.00
SESSION TERMINATED AT 21:36:00.00  ON 1982/05/23
REAL-TIME USED :  454.00 SEC.
CPU-TIME  USED :    1.31 SEC.
SIMULATION TIME USED :       4.2000 SEC.
NUMBER OF AGENTS CREATED :     1
NUMBER OF MESSAGES SENT :     2
```

```
   |----+----1----+----2----+----3----+----4----+----5----+----6----+----7--| B L O C K
 2 |type NODE is
 3 |script                                                                  |+1
 4 |    type AGENT_LIST is set of agent;
 5 |    functionhead EMPTY(AGENT_LIST:L) returns bool external;
 6 |    var AGENT_LIST: NEIGHBOURS;
 7 |    var agent: SENDER;
 8 |
 9 |    facethead MARKED
10 |
11 |    facet NOT_MARKED is                                                 |+2
12 |       public: MARK,KNOW;
13 |
14 |       operation KNOW (int:I;array (1..I) of agent:NGHBRS)              |+3
15 |                 assert I>=1 is
16 |          loop for J in 1..I do put NGHBRS(J) into NEIGHBOURS;endloop;  |+4-4
17 |       endoperation                                                      |-3
18 |                                                                         |*2
19 |       operation MARK(--> SENDER) assert not EMPTY(NEIGHBOURS) is       |+5
20 |          loop for N of NEIGHBOURS do                                   |+6
21 |             send MARK(self) to N;
22 |          endloop;                                                      |-6
23 |          replace by MARKED;                                            |*5
24 |       endoperation                                                      |-5
25 |                                                                         |*2
26 |       operation MARK(agent:SENDER) assert EMPTY(NEIGHBOURS) is         |+7
27 |          oper:ECHO;
28 |          ----------
29 |          send ECHO(self) to SENDER;
30 |          replace by MARKED;
31 |       endoperation;                                                     |-7
32 |                                                                         |*2
33 |    endfacet;                                                            |-2
34 |                                                                         |*1
35 |    facet MARKED is                                                      |+8
36 |       public: MARK,ECHO;
37 |
38 |       operation MARK(agent:SENDER) is                                  |+9
39 |          send ECHO(self) to SENDER;
40 |       endoperation;                                                     |-9
41 |                                                                         |*8
42 |       operation ECHO(agent:X) is                                       |+10
43 |          remove X from NEIGHBOURS;
44 |          if EMPTY(NEIGHBOURS) then send ECHO(self) to SENDER; endif;   |+11-11
45 |       endoperation                                                      |-10
46 |                                                                         |*8
47 |    endfacet                                                             |-8
48 |                                                                         |*1
49 |    initial NOT_MARKED
50 |endscript                                                                |-1
```

```
BMS-CSSA-COMPILER - DATE OF RELEASE: 30 SEP 1981  NO ERROR DETECTED
END OF COMPILING ON 1982/05/21 AT 13:25:07.00     RETURNCODE = 0
COMPILE-TIME (CPU) =    3.06 SEC.    EXECUTION-TIME =  9.00 SEC.
NUMBER OF SOURCE-LINES READ =   50     NUMBER OF TOKENS =   215
NUMBER OF OBJECT-RECORDS GENERATED =    714
```

C S S A - S I M U L A T I O N - S Y S T E M




```
PROGRAM GENERATED ON 1982/05/23 AT 20:32:59.00
BY BMS-CSSA-COMPILER (VERS. 30 SEP 1981)

PROTOCOL OF CSSA SESSION ON 1982/05/23 AT 21:25:40.00

>>>      0.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> LIMIT 10000                                                      <==
>>>      0.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> DISPLAY                                                          <==
```

IDENTIFIER	TYPE	VALUE
NODE	SCRIPT	NODE

```
>>>      0.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> VAR AGENT: ROOT:=NEW NODE                                        <==
>>>      0.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> VAR AGENT: N2,N3,N4,N5,N6,N7; #                                  <==
==> N2 := NEW NODE; #                        (* CREATE NODES *)      <==
==> N3 := NEW NODE; #                                                <==
==> N4 := NEW NODE; #                                                <==
==> N5 := NEW NODE; #                                                <==
==> N6 := NEW NODE; #                                                <==
==> N7 := NEW NODE; #                                                <==
==> DISPLAY                                                          <==
```

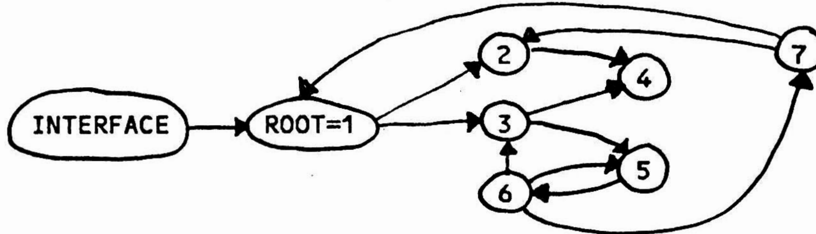
IDENTIFIER	TYPE	VALUE
NODE	SCRIPT	NODE
N2	AGENT	NODE(2)
N3	AGENT	NODE(3)
N4	AGENT	NODE(4)
N5	AGENT	NODE(5)
N6	AGENT	NODE(6)
N7	AGENT	NODE(7)
ROOT	AGENT	NODE(1)

```
>>>      0.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> RUN    (* CREATION MATCH *)                                      <==
>>>      0.100  SYSTEM TERMINATED
>>>      0.100  INTERFACE(1) : ENTER CSSA COMMAND -
==> DUMP ROOT                                                        <==

+++      0.100  RUNTIME STACK OF NODE(1)

        | FACET  NOT_MARKED              |
        | ENV: SCRIPT  NODE              |
        | LINE:    11                    |
        |................................|
        | SCRIPT  NODE                   |
        | ENV:                           |
        | LINE:    50                    |
        |................................|
        | NEIGHBOURS:                    |
        | SENDER = <NOAGENT>             |
```

```
>>>      0.100   INTERFACE(1) : ENTER CSSA COMMAND -
==> STATUS                                                                          <==
+++      0.100   ALL EXISTING AGENTS:
```

AGENT	FACET	OPERATION	MAILBOX
P1: INTERFACE(1) *			
P2: NODE(1)	NOT_MARKE		
P3: NODE(2)	NOT_MARKE		
P4: NODE(3)	NOT_MARKE		
P5: NODE(4)	NOT_MARKE		
P6: NODE(5)	NOT_MARKE		
P7: NODE(6)	NOT_MARKE		
P2: NODE(7)	NOT_MARKE		

```
>>>      0.100   INTERFACE(1) : ENTER CSSA COMMAND -
==> (*                DER ZUGRUNDELIEGENDE GRAPH                              *) #       <==
```




```
==> (*                                                                        *) #       <==
==> (*                                                                        *) #       <==
==> (*                                                                        *) #       <==
==> (*                                                                        *) #       <==
==> (*                                                                        *) #       <==
==> (*                                                                        *) #       <==
==> (*                                                                        *) #       <==
==> (*                                                                        *) #       <==
==> (*                                                                        *) #       <==
==> (*                                                                        *) #       <==
==> (*                                                                        *) #       <==
==> (*                                                                        *) #       <==
==> (*                                                                        *) #       <==
==> OPER: KNOW,MARK; #                                                                   <==
==> SEND KNOW(2,N2,N3) TO ROOT; #                         (* BUILD GRAPH *)              <==
==> SEND KNOW(1,N4) TO N2; #                                                             <==
==> SEND KNOW(2,N4,N5) TO N3; #                                                          <==
==> SEND KNOW(1,N6) TO N5; #                                                             <==
==> SEND KNOW(3,N3,N5,N7) TO N6; #                                                       <==
==> SEND KNOW(2,N2,ROOT) TO N7; #                                                        <==
==> OBSERVE;RUN 0.2                                                                      <==
+++      0.200   P2: NODE(1) RECEIVES KNOW(INT:2,AGENT:NODE(2),AGENT:
                              NODE(3)) FROM INTERFACE(1)
+++      0.200   P3: NODE(2) RECEIVES KNOW(INT:1,AGENT:NODE(4))
                              FROM INTERFACE(1)
+++      0.200   P2: NODE(1) STARTING OPERATION KNOW(INT:2,AGENT:NODE(2)
                             ,AGENT:NODE(3))
+++      0.200   P3: NODE(2) STARTING OPERATION KNOW(INT:1,AGENT:NODE(4))
>>>      0.300   INTERFACE(1) : ENTER CSSA COMMAND -
==> STATUS                                                                               <==
+++      0.300   ALL EXISTING AGENTS:
```

AGENT	FACET	OPERATION	MAILBOX
P1: INTERFACE(1) *			
P2: NODE(1) *	NOT_MARKE	KNOW	
P3: NODE(2) *	NOT_MARKE	KNOW	
P4: NODE(3)	NOT_MARKE		
P5: NODE(4)	NOT_MARKE		
P6: NODE(5)	NOT_MARKE		
P7: NODE(6)	NOT_MARKE		
P2: NODE(7)	NOT_MARKE		

```
>>>      0.300   INTERFACE(1) : ENTER CSSA COMMAND -
==> SYSSTATUS                                                                            <==
+++      0.300   SYSTEM STATUS:
```

PROC.	UTIL.	AGENTS
P1	100%	INTERFACE(1)*
P2	33%	NODE(1)* NODE(7)
P3	33%	NODE(2)*
P4	0%	NODE(3)
P5	0%	NODE(4)
P6	0%	NODE(5)
P7	0%	NODE(6)

BUS	UTIL.	#MSGS	AVG-QLEN	MESSAGES
B1	67%	3	1.00	KNOW
B2	67%	3	1.00	KNOW
B3	0%	0	0.00	
B4	0%	0	0.00	
B5	0%	0	0.00	
B6	0%	0	0.00	

```
|_______________________________________________________________________________|

>>>      0.300  INTERFACE(1) : ENTER CSSA COMMAND -
==> DUMP NODE(2)                                                                  <==

+++      0.300  RUNTIME STACK OF NODE(2)

                |----------------------------------|
                |  OPERATION  KNOW                 |
                |  ENV: FACET  NOT_MARKED          |
                |  LINE:   16                      |
                |..................................|
                |  I = 1                           |
                |  ANONYM = (1..1)                 |
                |  NGHBRS(1) = NODE(4)             |
                |----------------------------------|
                |  FACET  NOT_MARKED               |
                |  ENV: SCRIPT  NODE               |
                |  LINE:   12                      |
                |..................................|
                |----------------------------------|
                |  SCRIPT  NODE                    |
                |  ENV:                            |
                |  LINE:   50                      |
                |..................................|
                |  NEIGHBOURS:                     |
                |  ..... NODE(4)                   |
                |  SENDER = <NOAGENT>              |
                |__________________________________|

>>>      0.300  INTERFACE(1) : ENTER CSSA COMMAND -
==> RUN                                                                           <==
+++      0.400  P3: NODE(2) IS IDLE
+++      0.500  P7: NODE(6) RECEIVES KNOW(INT:3,AGENT:NODE(3),AGENT:
                           NODE(5),AGENT:NODE(7)) FROM INTERFACE(1)
+++      0.500  P6: NODE(5) RECEIVES KNOW(INT:1,AGENT:NODE(6))
                            FROM INTERFACE(1)
+++      0.500  P7: NODE(6) STARTING OPERATION KNOW(INT:3,AGENT:NODE(3)
                            ,AGENT:NODE(5),AGENT:NODE(7))
+++      0.500  P6: NODE(5) STARTING OPERATION KNOW(INT:1,AGENT:NODE(6))
+++      0.500  P2: NODE(1) IS IDLE
+++      0.500  P2: NODE(7) RECEIVES KNOW(INT:2,AGENT:NODE(2),AGENT:
                           NODE(1)) FROM INTERFACE(1)
+++      0.500  P2: NODE(7) STARTING OPERATION KNOW(INT:2,AGENT:NODE(2)
                            ,AGENT:NODE(1))
+++      0.600  P4: NODE(3) RECEIVES KNOW(INT:2,AGENT:NODE(4),AGENT:
                           NODE(5)) FROM INTERFACE(1)
+++      0.600  P4: NODE(3) STARTING OPERATION KNOW(INT:2,AGENT:NODE(4)
                            ,AGENT:NODE(5))
+++      0.700  P6: NODE(5) IS IDLE
+++      0.800  P2: NODE(7) IS IDLE
+++      0.900  P7: NODE(6) IS IDLE
+++      0.900  P4: NODE(3) IS IDLE
>>>      0.900  SYSTEM TERMINATED
>>>      0.900  INTERFACE(1) : ENTER CSSA COMMAND -
==> DUMP ROOT; DUMP NODE(2)                                                       <==

+++      0.900  RUNTIME STACK OF NODE(1)

                |----------------------------------|
                |  FACET  NOT_MARKED               |
                |  ENV: SCRIPT  NODE               |
                |  LINE:   12                      |
                |..................................|
                |----------------------------------|
                |  SCRIPT  NODE                    |
                |  ENV:                            |
                |  LINE:   50                      |
                |..................................|
                |  NEIGHBOURS:                     |
                |  ..... NODE(2)                   |
                |  ..... NODE(3)                   |
                |  SENDER = <NOAGENT>              |
                |__________________________________|


+++      0.900  RUNTIME STACK OF NODE(2)

                |----------------------------------|
                |  FACET  NOT_MARKED               |
                |  ENV: SCRIPT  NODE               |
                |  LINE:   12                      |
                |..................................|
                |----------------------------------|
                |  SCRIPT  NODE                    |
                |  ENV:                            |
                |  LINE:   50                      |
                |..................................|
                |  NEIGHBOURS:                     |
                |  ..... NODE(4)                   |
                |  SENDER = <NOAGENT>              |
```

```
>>>     0.900  INTERFACE(1) : ENTER CSSA COMMAND -
==> SEND MARK(SELF) TO ROOT;                                                    <==
+++     0.900  P1: INTERFACE(1) SENDS MARK(AGENT:INTERFACE(1))
                   TO NODE(1)
>>>     0.900  INTERFACE(1) : ENTER CSSA COMMAND -
==> SYSTEM; RUN 0.5                                                             <==
///     1.000  B1: BUS IS IDLE
///     1.000  P2: PROCESSOR IS ACTIVATED BY EXTERNAL INTERRUPT
+++     1.000  P2: NODE(1) RECEIVES MARK(AGENT:INTERFACE(1))
                   FROM INTERFACE(1)
///     1.000  P2: NODE(1) ACTIVATED
+++     1.000  P2: NODE(1) STARTING OPERATION MARK(AGENT:INTERFACE(1))
+++     1.200  P2: NODE(1) SENDS MARK(AGENT:NODE(1)) TO NODE(2)
///     1.200  P2: NODE(1) SUSPENDED
///     1.200  P2: NODE(1) ACTIVATED
///     1.200  B1: STARTING TO TRANSMIT MARK(AGENT:NODE(1))
///     1.300  B1: BUS IS IDLE
+++     1.300  P2: NODE(1) SENDS MARK(AGENT:NODE(1)) TO NODE(3)
///     1.300  P2: NODE(1) SUSPENDED
///     1.300  P2: NODE(1) ACTIVATED
///     1.300  B3: STARTING TO TRANSMIT MARK(AGENT:NODE(1))
///     1.400  B3: BUS IS IDLE
///     1.400  P4: PROCESSOR IS ACTIVATED BY EXTERNAL INTERRUPT
+++     1.400  P4: NODE(3) RECEIVES MARK(AGENT:NODE(1)) FROM NODE(1)
///     1.400  P4: NODE(3) ACTIVATED
+++     1.400  P2: NODE(1) PERFORMS FACETTING : NOT_MARKED --> MARKED
+++     1.400  P4: NODE(3) STARTING OPERATION MARK(AGENT:NODE(1))
+++     1.400  P2: NODE(1) IS IDLE
///     1.400  P2: PROCESSOR IS IDLE
>>>     1.400  INTERFACE(1) : ENTER CSSA COMMAND -
==> NOSYSTEM; EVENT; RUN 1                                                      <==
+++     1.600  ALL EXISTING AGENTS:
```

AGENT	FACET	OPERATION	MAILBOX
P1: INTERFACE(1) *			
P2: NODE(1)	MARKED		
P3: NODE(2)	NOT_MARKE		
P4: NODE(3) *	NOT_MARKE	MARK	===> MARK(AGENT:NODE(3))
P5: NODE(4)	NOT_MARKE		
P6: NODE(5)	NOT_MARKE		
P7: NODE(6)	NOT_MARKE		
P2: NODE(7)	NOT_MARKE		

```
+++     1.600  P4: NODE(3) SENDS MARK(AGENT:NODE(3)) TO NODE(4)
+++     1.700  ALL EXISTING AGENTS:
```

AGENT	FACET	OPERATION	MAILBOX
P1: INTERFACE(1) *			
P2: NODE(1)	MARKED		
P3: NODE(2)	NOT_MARKE		
P4: NODE(3) *	NOT_MARKE	MARK	===> MARK(AGENT:NODE(3))
P5: NODE(4)	NOT_MARKE		
P6: NODE(5)	NOT_MARKE		
P7: NODE(6)	NOT_MARKE		
P2: NODE(7)	NOT_MARKE		

```
+++     1.700  P4: NODE(3) SENDS MARK(AGENT:NODE(3)) TO NODE(5)
+++     1.800  ALL EXISTING AGENTS:
```

AGENT	FACET	OPERATION	MAILBOX
P1: INTERFACE(1) *			
P2: NODE(1)	MARKED		
P3: NODE(2)	NOT_MARKE		
P4: NODE(3) *	NOT_MARKE	MARK	
P5: NODE(4) *	NOT_MARKE		MARK(AGENT:NODE(3))
P6: NODE(5)	NOT_MARKE		
P7: NODE(6)	NOT_MARKE		
P2: NODE(7)	NOT_MARKE		

```
+++     1.800  P5: NODE(4) RECEIVES MARK(AGENT:NODE(3)) FROM NODE(3)
+++     1.800  P4: NODE(3) PERFORMS FACETTING : NOT_MARKED --> MARKED
+++     1.800  P4: NODE(3) IS IDLE
+++     1.800  P5: NODE(4) STARTING OPERATION MARK(AGENT:NODE(3))
+++     1.900  ALL EXISTING AGENTS:
```

AGENT	FACET	OPERATION	MAILBOX
P1: INTERFACE(1) *			
P2: NODE(1)	MARKED		
P3: NODE(2)	NOT_MARKE		

```
| P4: NODE(3)          | MARKED     |           |                                     |
| P5: NODE(4)         *| NOT_MARKE  | MARK      | ===> ECHO(AGENT:NODE(4))            |
| P6: NODE(5)          | NOT_MARKE  |           |                                     |
| P7: NODE(6)          | NOT_MARKE  |           |                                     |
| P2: NODE(7)          | NOT_MARKE  |           |                                     |
|-----------------------------------------------------------------------------------|

+++      1.900  P5: NODE(4) SENDS ECHO(AGENT:NODE(4)) TO NODE(3)
+++      2.000  P5: NODE(4) PERFORMS FACETTING : NOT_MARKED --> MARKED
+++      2.000  P5: NODE(4) IS IDLE
+++      2.000  ALL EXISTING AGENTS:

| AGENT                | FACET      | OPERATION |              MAILBOX                |
|----------------------|------------|-----------|-------------------------------------|
| P1: INTERFACE(1)    *|            |           |                                     |
| P2: NODE(1)          | MARKED     |           |                                     |
| P3: NODE(2)         *| NOT_MARKE  |           | MARK(AGENT:NODE(1))                 |
| P4: NODE(3)          | MARKED     |           |                                     |
| P5: NODE(4)          | MARKED     |           |                                     |
| P6: NODE(5)          | NOT_MARKE  |           |                                     |
| P7: NODE(6)          | NOT_MARKE  |           |                                     |
| P2: NODE(7)          | NOT_MARKE  |           |                                     |
|-----------------------------------------------------------------------------------|

+++      2.000  P3: NODE(2) RECEIVES MARK(AGENT:NODE(1)) FROM NODE(1)
+++      2.000  P3: NODE(2) STARTING OPERATION MARK(AGENT:NODE(1))
+++      2.100  ALL EXISTING AGENTS:

| AGENT                | FACET      | OPERATION |              MAILBOX                |
|----------------------|------------|-----------|-------------------------------------|
| P1: INTERFACE(1)    *|            |           |                                     |
| P2: NODE(1)          | MARKED     |           |                                     |
| P3: NODE(2)         *| NOT_MARKE  | MARK      |                                     |
| P4: NODE(3)         *| MARKED     |           | ECHO(AGENT:NODE(4))                 |
| P5: NODE(4)          | MARKED     |           |                                     |
| P6: NODE(5)          | NOT_MARKE  |           |                                     |
| P7: NODE(6)          | NOT_MARKE  |           |                                     |
| P2: NODE(7)          | NOT_MARKE  |           |                                     |
|-----------------------------------------------------------------------------------|

+++      2.100  P4: NODE(3) RECEIVES ECHO(AGENT:NODE(4)) FROM NODE(4)
+++      2.100  P4: NODE(3) STARTING OPERATION ECHO(AGENT:NODE(4))
+++      2.200  ALL EXISTING AGENTS:

| AGENT                | FACET      | OPERATION |              MAILBOX                |
|----------------------|------------|-----------|-------------------------------------|
| P1: INTERFACE(1)    *|            |           |                                     |
| P2: NODE(1)          | MARKED     |           |                                     |
| P3: NODE(2)         *| NOT_MARKE  | MARK      | ===> MARK(AGENT:NODE(2))            |
| P4: NODE(3)         *| MARKED     | ECHO      |                                     |
| P5: NODE(4)          | MARKED     |           |                                     |
| P6: NODE(5)          | NOT_MARKE  |           |                                     |
| P7: NODE(6)          | NOT_MARKE  |           |                                     |
| P2: NODE(7)          | NOT_MARKE  |           |                                     |
|-----------------------------------------------------------------------------------|

+++      2.200  P3: NODE(2) SENDS MARK(AGENT:NODE(2)) TO NODE(4)
+++      2.300  P4: NODE(3) IS IDLE
+++      2.300  ALL EXISTING AGENTS:

| AGENT                | FACET      | OPERATION |              MAILBOX                |
|----------------------|------------|-----------|-------------------------------------|
| P1: INTERFACE(1)    *|            |           |                                     |
| P2: NODE(1)          | MARKED     |           |                                     |
| P3: NODE(2)         *| NOT_MARKE  | MARK      |                                     |
| P4: NODE(3)          | MARKED     |           |                                     |
| P5: NODE(4)          | MARKED     |           |                                     |
| P6: NODE(5)         *| NOT_MARKE  |           | MARK(AGENT:NODE(3))                 |
| P7: NODE(6)          | NOT_MARKE  |           |                                     |
| P2: NODE(7)          | NOT_MARKE  |           |                                     |
|-----------------------------------------------------------------------------------|

+++      2.300  P6: NODE(5) RECEIVES MARK(AGENT:NODE(3)) FROM NODE(3)
+++      2.300  P3: NODE(2) PERFORMS FACETTING : NOT_MARKED --> MARKED
+++      2.300  P6: NODE(5) STARTING OPERATION MARK(AGENT:NODE(3))
+++      2.300  P3: NODE(2) IS IDLE
>>>      2.400  INTERFACE(1) : ENTER CSSA COMMAND -
==> NOEVENT; TRACE N2; DUMP N2; RUN                                                        <==

+++      2.400  RUNTIME STACK OF NODE(2)

                    |-------------------------------|
                    |  FACET  MARKED                |
                    |  ENV: SCRIPT  NODE            |
                    |  LINE:    35                  |
                    |...............................|
                    |-------------------------------|
                    |  SCRIPT  NODE                 |
                    |  ENV:                         |
```

```
                 | LINE:   50                          |
                 ...................................
                 | NEIGHBOURS:                         |
                 | ..... NODE(4)                       |
                 | SENDER = NODE(1)                    |
                 |_____________________________________|

+++     2.500  P6: NODE(5) SENDS MARK(AGENT:NODE(5)) TO NODE(6)
+++     2.600  P6: NODE(5) PERFORMS FACETTING : NOT_MARKED --> MARKED
+++     2.600  P6: NODE(5) IS IDLE
+++     2.700  P7: NODE(6) RECEIVES MARK(AGENT:NODE(5)) FROM NODE(5)
+++     2.700  P7: NODE(6) STARTING OPERATION MARK(AGENT:NODE(5))
+++     2.900  P7: NODE(6) SENDS MARK(AGENT:NODE(6)) TO NODE(3)
+++     3.000  P7: NODE(6) SENDS MARK(AGENT:NODE(6)) TO NODE(5)
+++     3.100  P7: NODE(6) SENDS MARK(AGENT:NODE(6)) TO NODE(7)
+++     3.100  P5: NODE(4) RECEIVES MARK(AGENT:NODE(2)) FROM NODE(2)
+++     3.100  P5: NODE(4) STARTING OPERATION MARK(AGENT:NODE(2))
+++     3.200  P6: NODE(5) RECEIVES MARK(AGENT:NODE(6)) FROM NODE(6)
+++     3.200  P7: NODE(6) PERFORMS FACETTING : NOT_MARKED --> MARKED
+++     3.200  P6: NODE(5) STARTING OPERATION MARK(AGENT:NODE(6))
+++     3.200  P7: NODE(6) IS IDLE
+++     3.200  P5: NODE(4) SENDS ECHO(AGENT:NODE(4)) TO NODE(2)
+++     3.200  P5: NODE(4) IS IDLE
+++     3.300  P6: NODE(5) SENDS ECHO(AGENT:NODE(5)) TO NODE(6)
+++     3.300  P6: NODE(5) IS IDLE
+++     3.500  P7: NODE(6) RECEIVES ECHO(AGENT:NODE(5)) FROM NODE(5)
+++     3.500  P7: NODE(6) STARTING OPERATION ECHO(AGENT:NODE(5))
+++     3.700  P7: NODE(6) IS IDLE
+++     4.000  P3: NODE(2) RECEIVES ECHO(AGENT:NODE(4)) FROM NODE(4)

___     4.000  TRACING NODE(2) / MARKED ______________________________
               SEARCHING NEXT_MESSAGE
               FOUND : ECHO(AGENT:NODE(4))
               LINE  42: X := NODE(4)
               STARTING OPERATION ECHO(AGENT:NODE(4))
               LINE  43: NODE(4) REMOVED FROM A SET
               LINE  44: CALLING FUNCTION EMPTY
               LINE   5: L
_____________________________________________________________________

+++     4.100  P2: NODE(7) RECEIVES MARK(AGENT:NODE(6)) FROM NODE(6)
+++     4.100  P4: NODE(3) RECEIVES MARK(AGENT:NODE(6)) FROM NODE(6)
+++     4.100  P2: NODE(7) STARTING OPERATION MARK(AGENT:NODE(6))
+++     4.100  P4: NODE(3) STARTING OPERATION MARK(AGENT:NODE(6))
+++     4.200  P4: NODE(3) SENDS ECHO(AGENT:NODE(3)) TO NODE(6)
+++     4.200  P4: NODE(3) IS IDLE

___     4.300  TRACING NODE(2) / MARKED / ECHO _______________________
               LINE  44: SEND ECHO(AGENT:NODE(2)) TO NODE(1)
_____________________________________________________________________

+++     4.300  P2: NODE(7) SENDS MARK(AGENT:NODE(7)) TO NODE(2)

___     4.300  TRACING NODE(2) / MARKED / ECHO _______________________
_____________________________________________________________________

___     4.300  TRACING NODE(2) / MARKED / ECHO _______________________
_____________________________________________________________________

___     4.300  TRACING NODE(2) / MARKED ______________________________
               SEARCHING NEXT_MESSAGE - NO MESSAGE FOUND
_____________________________________________________________________

+++     4.300  P3: NODE(2) IS IDLE
+++     4.400  P2: NODE(7) SENDS MARK(AGENT:NODE(7)) TO NODE(1)
+++     4.400  P2: NODE(1) RECEIVES MARK(AGENT:NODE(7)) FROM NODE(7)
+++     4.400  P2: NODE(1) STARTING OPERATION MARK(AGENT:NODE(7))
+++     4.500  P2: NODE(1) SENDS ECHO(AGENT:NODE(1)) TO NODE(7)
+++     4.500  P2: NODE(7) RECEIVES ECHO(AGENT:NODE(1)) FROM NODE(1)
+++     4.600  P2: NODE(7) PERFORMS FACETTING : NOT_MARKED --> MARKED
+++     4.600  P2: NODE(7) STARTING OPERATION ECHO(AGENT:NODE(1))
+++     4.800  P2: NODE(7) IS IDLE
+++     4.800  P2: NODE(1) IS IDLE
+++     5.000  P2: NODE(1) RECEIVES ECHO(AGENT:NODE(2)) FROM NODE(2)
+++     5.000  P2: NODE(1) STARTING OPERATION ECHO(AGENT:NODE(2))
+++     5.100  P3: NODE(2) RECEIVES MARK(AGENT:NODE(7)) FROM NODE(7)
+++     5.100  P7: NODE(6) RECEIVES ECHO(AGENT:NODE(3)) FROM NODE(3)

___     5.100  TRACING NODE(2) / MARKED ______________________________
               SEARCHING NEXT_MESSAGE
               FOUND : MARK(AGENT:NODE(7))
               LINE  38: SENDER := NODE(7)
               STARTING OPERATION MARK(AGENT:NODE(7))
_____________________________________________________________________

+++     5.100  P7: NODE(6) STARTING OPERATION ECHO(AGENT:NODE(3))
```

```
___      5.200   TRACING NODE(2) / MARKED / MARK ______________________________
                 LINE  39: SEND ECHO(AGENT:NODE(2)) TO NODE(7)
______________________________________________________________________________

___      5.200   TRACING NODE(2) / MARKED / MARK ______________________________
______________________________________________________________________________

+++      5.200   P2: NODE(1) IS IDLE

___      5.200   TRACING NODE(2) / MARKED / MARK ______________________________
______________________________________________________________________________

___      5.200   TRACING NODE(2) / MARKED ____________________________________
                 SEARCHING NEXT_MESSAGE - NO MESSAGE FOUND
______________________________________________________________________________

+++      5.200   P3: NODE(2) IS IDLE
+++      5.300   P7: NODE(6) IS IDLE
+++      6.000   P2: NODE(7) RECEIVES ECHO(AGENT:NODE(2)) FROM NODE(2)
+++      6.000   P2: NODE(7) STARTING OPERATION ECHO(AGENT:NODE(2))
+++      6.300   P2: NODE(7) SENDS ECHO(AGENT:NODE(7)) TO NODE(6)
+++      6.300   P2: NODE(7) IS IDLE
+++      7.100   P7: NODE(6) RECEIVES ECHO(AGENT:NODE(7)) FROM NODE(7)
+++      7.100   P7: NODE(6) STARTING OPERATION ECHO(AGENT:NODE(7))
+++      7.400   P7: NODE(6) SENDS ECHO(AGENT:NODE(6)) TO NODE(5)
+++      7.400   P7: NODE(6) IS IDLE
+++      7.600   P6: NODE(5) RECEIVES ECHO(AGENT:NODE(6)) FROM NODE(6)
+++      7.600   P6: NODE(5) STARTING OPERATION ECHO(AGENT:NODE(6))
+++      7.900   P6: NODE(5) SENDS ECHO(AGENT:NODE(5)) TO NODE(3)
+++      7.900   P6: NODE(5) IS IDLE
+++      9.100   P4: NODE(3) RECEIVES ECHO(AGENT:NODE(5)) FROM NODE(5)
+++      9.100   P4: NODE(3) STARTING OPERATION ECHO(AGENT:NODE(5))
+++      9.400   P4: NODE(3) SENDS ECHO(AGENT:NODE(3)) TO NODE(1)
+++      9.400   P4: NODE(3) IS IDLE
+++      9.500   P2: NODE(1) RECEIVES ECHO(AGENT:NODE(3)) FROM NODE(3)
+++      9.500   P2: NODE(1) STARTING OPERATION ECHO(AGENT:NODE(3))
+++      9.800   P2: NODE(1) SENDS ECHO(AGENT:NODE(1)) TO INTERFACE(1)
+++      9.800   P2: NODE(1) IS IDLE
+++      9.900   P1: INTERFACE(1) RECEIVES ECHO(AGENT:NODE(1))
                              FROM NODE(1)
>>>     10.000   SYSTEM TERMINATED
>>>     10.000   INTERFACE(1) : ENTER CSSA COMMAND -
==> MAILBOX                                                              <==
MAILBOX OF INTERFACE(1) :
______________________________________________________________________________
(1) ECHO(AGENT:NODE(1))
______________________________________________________________________________

>>>     10.000   INTERFACE(1) : ENTER CSSA COMMAND -
==> STATUS                                                               <==
+++     10.000   ALL EXISTING AGENTS:
```

AGENT	FACET	OPERATION	MAILBOX
P1: INTERFACE(1) *			ECHO(AGENT:NODE(1))
P2: NODE(1)	MARKED		
P3: NODE(2)	MARKED		
P4: NODE(3)	MARKED		
P5: NODE(4)	MARKED		
P6: NODE(5)	MARKED		
P7: NODE(6)	MARKED		
P2: NODE(7)	MARKED		

```
>>>     10.000   INTERFACE(1) : ENTER CSSA COMMAND -
==> SYSSTATUS                                                            <==
+++     10.000   SYSTEM STATUS:
```

PROC.	UTIL.	AGENTS
P1	100%	INTERFACE(1)*
P2	25%	NODE(7) NODE(1)
P3	9%	NODE(2)
P4	13%	NODE(3)
P5	3%	NODE(4)
P6	9%	NODE(5)
P7	16%	NODE(6)

BUS	UTIL.	#MSGS	AVG-QLEN	MESSAGES
B1	17%	17	0.24	
B2	15%	15	0.33	
B3	9%	9	0.00	

```
| B4        | 4%      | 4      |         0.00 |                                        |
| B5        | 7%      | 7      |         0.00 |                                        |
| B6        | 9%      | 9      |         0.00 |                                        |
|________________________________________________________________________________________|

>>>    10.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> TIME                                                                                      <==
1982/05/23   21:25:56.00     CPU =     2.56 SEC.    CSSA-SESSION =   17.00 SEC.
>>>    10.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> STOP ROOT,N2,N3,N4,N5,N6,N7                                                               <==
>>>    10.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> STATUS                                                                                    <==
+++    10.000  ALL EXISTING AGENTS:
```

AGENT	FACET	OPERATION	MAILBOX
P1: INTERFACE(1) *			ECHO(AGENT:NODE(1))

```
>>>    10.000  INTERFACE(1) : ENTER CSSA COMMAND -
==> TERMINATE                                                                                 <==
+++    10.000  ALL EXISTING AGENTS:
```

AGENT	FACET	OPERATION	MAILBOX
P1: INTERFACE(1) *			ECHO(AGENT:NODE(1))

```
                        CSSA-SESSION-STATISTICS
                        =======================


SESSION  STARTED  AT  21:25:39.00
SESSION TERMINATED AT 21:26:14.00  ON 1982/05/23
REAL-TIME USED :   36.00 SEC.
CPU-TIME  USED :    2.71 SEC.
SIMULATION TIME USED :      10.0000 SEC.
NUMBER OF AGENTS CREATED :    7
NUMBER OF MESSAGES SENT :   30
```

METALANGUAGE (EXTENDED BNF)

```
<...> : NONTERMINALS (METALINGUISTIC VARIABLES)
UNDERLINED CHARACTERSTRINGS : KEYWORDS (RESERVED WORDS)
'...' : TERMINAL SYMBOLS OF THE DEFINED LANGUAGE ARE
        ENCLOSED IN SINGLE QUOTES (EXCEPT KEYWORDS)
::=   : THE LHS METAL. VAR. IS DEFINED BY THE RHS METAL. EXPR.
( )   : METALINGUISTIC PARENTHESES
|     : SEPERATES ALTERNATIVE METAL. EXPRESSIONS
[   ] : THE METAL. EXPRESSIONS ENCLOSED IN [ AND ] ARE OPTIONAL
*     : THE METAL. EXPR. BEFORE * MAY OCCUR 0 OR MORE TIMES
+     : THE METAL. EXPR. BEFORE + MAY OCCUR 1 OR MORE TIMES
||    : THE METAL. EXPRESSION IMMEDIATLY BEFORE || MAY BE REPEATED
        SEVERAL TIMES SEPERATED BY THE METAL. EXPR. AFTER ||

PRIORITY OF META-OPERATORS IN DESCENDING ORDER :
*  +  ||  CONCAT  |
```

```
<STATEMENT>
            ::= <DISPLAY-STATEMENT>
            ::= <STATUS-STATEMENT>
            ::= <SYSSTATUS-STATEMENT>
            ::= <EVENT-STATEMENT>
            ::= <NOEVENT-STATEMENT>
            ::= <SYSTEM-STATEMENT>
            ::= <NOSYSTEM-STATEMENT>
            ::= <OBSERVE-STATEMENT>
            ::= <PROT-STATEMENT>
            ::= <NOPROT-STATEMENT>
            ::= <NOOBSERVE-STATEMENT>
            ::= <TRACE-STATEMENT>
            ::= <NOTRACE-STATEMENT>
            ::= <SNAPSHOT-STATEMENT>
            ::= <NOSNAPSHOT-STATEMENT>
            ::= <CONST-VAR-DECL>
            ::= <ASSIGNMENT>
            ::= <OPER-DECL>
            ::= <PORT-DECL>
            ::= <SEND-STATEMENT>
            ::= <RECEIVE-STATEMENT>
            ::= <REPLY-STATEMENT>
            ::= <MAILBOX-STATEMENT>
            ::= <DELETE-STATEMENT>
            ::= <LIMIT-STATEMENT>
            ::= <RUN-STATEMENT>
            ::= <DUMP-STATEMENT>
            ::= <STACKS-STATEMENT>
            ::= <STOP-STATEMENT>
            ::= <START-STATEMENT>
            ::= <READ-STATEMENT>
            ::= <HELP-STATEMENT>
            ::= <TIME-STATEMENT>
            ::= <TERMINATE-STATEMENT>
            ::= <EMPTY>
```

```
<DISPLAY-STATEMENT> ::= (DISPLAY | '?') [ <FACTOR> ]                              2

<READ-STATEMENT> ::= READ <IDENTIFIER>                                            3

<TERMINATE-STATEMENT> ::= TERMINATE                                               4

<AGENT-DENOTATION> ::= <IDENTIFIER> [ '(' <INT-CONST> ')' ]                       5
                   ::= SELF | INTERFACE

<NOTRACE-STATEMENT> ::= NOTRACE <AGENT-DENOTATION> II ','                         6

<TRACE-STATEMENT> ::= TRACE <AGENT-DENOTATION> II ','                             7

<STOP-STATEMENT> ::= STOP <AGENT-DENOTATION> II ','                               8

<START-STATEMENT> ::= START <AGENT-DENOTATION> II ','                             9

<NOEVENT-STATEMENT> ::= NOEVENT                                                   10

<EVENT-STATEMENT> ::= EVENT [ STOP ]                                              11

<NOSNAPSHOT-STATEMENT> ::= NOSNAPSHOT <AGENT-DENOTATION> II ','                   12

<SNAPSHOT-STATEMENT> ::= SNAPSHOT <AGENT-DENOTATION> II ','                       13

<STATUS-STATEMENT> ::= STATUS                                                     14

<SYSSTATUS-STATEMENT> ::= SYSSTATUS                                               15

<NOPROT-STATEMENT> ::= NOPROT                                                     16

<PROT-STATEMENT> ::= PROT                                                         17

<NOOBSERVE-STATEMENT> ::= NOOBSERVE                                               18

<OBSERVE-STATEMENT> ::= OBSERVE                                                   19

<NOSYSTEM-STATEMENT> ::= NOSYSTEM                                                 20

<SYSTEM-STATEMENT> ::= SYSTEM                                                     21

<STACKS-STATEMENT> ::= STACKS <AGENT-DENOTATION>                                  22

<DUMP-STATEMENT> ::= DUMP <AGENT-DENOTATION>                                      23

<TIME-STATEMENT> ::= TIME                                                         24

<RUN-STATEMENT> ::= RUN [ <INT-CONST> | <REAL-CONST> ]                            25

<LIMIT-STATEMENT> ::= LIMIT <INT-CONST>                                           26

<MAILBOX-STATEMENT> ::= MAILBOX [ <AGENT-DENOTATION> ]                            27

<HELP-STATEMENT> ::= HELP                                                         28
```

```
<PATTERN> ::= '(' [ <IDENTIFIER> ] || ',' ')'                                         29
-----------------------------------------------------------------------------------------
<PORT-DECL> ::= PORT ':' <IDENTIFIER> || ','                                            30
-----------------------------------------------------------------------------------------
<OPER-DECL> ::= OPER ':' <IDENTIFIER> || ','                                            31
-----------------------------------------------------------------------------------------
<CONST-VAR-DECL> ::= (CONST|VAR) <TYPE> ':' <IDENTIFIER> ||','                          32
                     [ (':-'|':=') <EXPRESSION> ]
-----------------------------------------------------------------------------------------
<TYPE> ::= BOOL | INT | STRING | AGENT | OPER | SCRIPT | REAL                            33
-----------------------------------------------------------------------------------------
<SEND-STATEMENT> ::= SEND <OPER-EXPRESSION> <MESSAGE>                                   34
                       [ TO <AGENT-EXPRESSION>
                         [ REPLY TO <IDENTIFIER> | INHERIT ] ]
-----------------------------------------------------------------------------------------
<REPLY-STATEMENT> ::= REPLY [ <INT-CONST> ] <MESSAGE>                                   35
-----------------------------------------------------------------------------------------
<DELETE-STATEMENT> ::= DELETE <INT-CONST>                                               36
-----------------------------------------------------------------------------------------
<RECEIVE-STATEMENT> ::= RECEIVE <INT-CONST> [ <PATTERN> ]                               37
-----------------------------------------------------------------------------------------
<MESSAGE> ::= '(' <EXPRESSION> || ',' ')'                                               38
          ::= <EMPTY>
-----------------------------------------------------------------------------------------
<ASSIGNMENT> ::= <IDENTIFIER> (':='|':-') <EXPRESSION>                                  39
-----------------------------------------------------------------------------------------
<INT-EXPRESSION> ::= <EXPRESSION>                                                       40
-----------------------------------------------------------------------------------------
<BOOL-EXPRESSION> ::= <EXPRESSION>                                                      41
-----------------------------------------------------------------------------------------
<STRING-EXPRESSION> ::= <EXPRESSION>                                                    42
-----------------------------------------------------------------------------------------
<SCRIPT-EXPRESSION> ::= <EXPRESSION>                                                    43
-----------------------------------------------------------------------------------------
<AGENT-EXPRESSION> ::= <EXPRESSION>                                                     44
-----------------------------------------------------------------------------------------
<OPER-EXPRESSION> ::= <EXPRESSION>                                                      45
-----------------------------------------------------------------------------------------
<REAL-EXPRESSION> ::= <EXPRESSION>                                                      46
-----------------------------------------------------------------------------------------
<EXPRESSION> ::= <FACTOR>                                                               47
-----------------------------------------------------------------------------------------
<FACTOR>       ::= [ - ]                                                                48
                  ( <INT-CONST>
                  | <REAL-CONST>
                  | <STRING-CONST>
                  |  OWNMODE
                  |  NOSCRIPT
                  |  NEW <SCRIPT-EXPRESSION>    <MESSAGE>
                  |  NOAGENT
                  |  NOOPER
                  |  SELF
                  |  INTERFACE
                  | <IDENTIFIER>
                  )
```

LIST OF KEYWORS :
=================

AGENT
BOOL
CONST
DELETE
DISPLAY
DUMP
EVENT
HELP
INHERIT
INT
INTERFACE
LIMIT
MAILBOX
NEW
NOAGENT
NOEVENT
NOOBSERVE
NOOPER
NOPROT
NOSCRIPT
NOSNAPSHOT
NOSYSTEM
NOTRACE
OBSERVE
OPER
OWNMODE
PORT
PROT
READ
REAL
RECEIVE
REPLY
RUN
SCRIPT
SELF
SEND
SNAPSHOT
STACKS
START
STATUS
STOP
STRING
SYSSTATUS
SYSTEM
TERMINATE
TIME
TO
TRACE
VAR

SYNTAX CROSS REFERENCE LISTING

THE LISTING TELLS WHERE EACH SYNTACTIC TERM IS USED IN THE PRODUCTIONS. IT ALSO SERVES AS AN INDEX SHOWING WHERE EACH TERM IS DEFINED. AN ELLIPSIS (...) INDICATES THAT A SYNTACTIC TERM IS NOT DEFINED BY A SYNTAX PRODUCTION. RESERVED WORDS APPEAR AT THE END OF THE LISTING.

```
<AGENT_DENOTATION>   5
        <NOTRACE_STATEMENT>  6 , <TRACE_STATEMENT>  7 ,
       <STOP_STATEMENT>  8 , <START_STATEMENT>  9 ,
       <NOSNAPSHOT_STATEMENT> 12 , <SNAPSHOT_STATEMENT> 13 ,
       <STACKS_STATEMENT> 22 , <DUMP_STATEMENT> 23 ,
       <MAILBOX_STATEMENT> 27
<AGENT_EXPRESSION>  44
        <SEND_STATEMENT> 34
<ASSIGNMENT>  39
        <STATEMENT>  1
<BOOL_EXPRESSION>  41

<CONST_VAR_DECL>  32
        <STATEMENT>  1
<DELETE_STATEMENT>  36
        <STATEMENT>  1
<DISPLAY_STATEMENT>   2
        <STATEMENT>  1
<DUMP_STATEMENT>  23
        <STATEMENT>  1
<EMPTY> ...
        <STATEMENT>  1 , <MESSAGE> 38
<EVENT_STATEMENT>  11
        <STATEMENT>  1
<EXPRESSION>  47
        <CONST_VAR_DECL> 32 , <MESSAGE> 38 , <ASSIGNMENT> 39 ,
       <INT_EXPRESSION> 40 , <BOOL_EXPRESSION> 41 ,
       <STRING_EXPRESSION> 42 , <SCRIPT_EXPRESSION> 43 ,
       <AGENT_EXPRESSION> 44 , <OPER_EXPRESSION> 45 ,
       <REAL_EXPRESSION> 46
<FACTOR>  48
        <DISPLAY_STATEMENT>  2 , <EXPRESSION> 47
<HELP_STATEMENT>  28
        <STATEMENT>  1
<IDENTIFIER> ...
        <READ_STATEMENT>  3 , <AGENT_DENOTATION>  5 , <PATTERN> 29 ,
       <PORT_DECL> 30 , <OPER_DECL> 31 , <CONST_VAR_DECL> 32 ,
       <SEND_STATEMENT> 34 , <ASSIGNMENT> 39 , <FACTOR> 48
<INT-CONST> ...
        <FACTOR> 48
<INT_CONST> ...
        <AGENT_DENOTATION>  5 , <RUN_STATEMENT> 25 ,
       <LIMIT_STATEMENT> 26 , <REPLY_STATEMENT> 35 ,
       <DELETE_STATEMENT> 36 , <RECEIVE_STATEMENT> 37
<INT_EXPRESSION>  40

<LIMIT_STATEMENT>  26
        <STATEMENT>  1
<MAILBOX_STATEMENT>  27
        <STATEMENT>  1
```

```
<MESSAGE>  38
        <SEND_STATEMENT> 34 , <REPLY_STATEMENT> 35 , <FACTOR> 48
<NOEVENT_STATEMENT>  10
        <STATEMENT>  1
<NOOBSERVE_STATEMENT>  18
        <STATEMENT>  1
<NOPROT_STATEMENT>  16
        <STATEMENT>  1
<NOSNAPSHOT_STATEMENT>  12
        <STATEMENT>  1
<NOSYSTEM_STATEMENT>  20
        <STATEMENT>  1
<NOTRACE_STATEMENT>   6
        <STATEMENT>  1
<OBSERVE_STATEMENT>  19
        <STATEMENT>  1
<OPER_DECL>  31
        <STATEMENT>  1
<OPER_EXPRESSION>  45
        <SEND_STATEMENT> 34
<PATTERN>  29
        <RECEIVE_STATEMENT> 37
<PORT_DECL>  30
        <STATEMENT>  1
<PROT_STATEMENT>  17
        <STATEMENT>  1
<READ_STATEMENT>   3
        <STATEMENT>  1
<REAL_CONST> ...
        <RUN_STATEMENT> 25 , <FACTOR> 48
<REAL_EXPRESSION>  46

<RECEIVE_STATEMENT>  37
        <STATEMENT>  1
<REPLY_STATEMENT>  35
        <STATEMENT>  1
<RUN_STATEMENT>  25
        <STATEMENT>  1
<SCRIPT_EXPRESSION>  43
        <FACTOR> 48
<SEND_STATEMENT>  34
        <STATEMENT>  1
<SNAPSHOT_STATEMENT>  13
        <STATEMENT>  1
<STACKS_STATEMENT>  22
        <STATEMENT>  1
<START_STATEMENT>   9
        <STATEMENT>  1
<STATEMENT>   1

<STATUS_STATEMENT>  14
        <STATEMENT>  1
<STOP_STATEMENT>   8
        <STATEMENT>  1
<STRING_CONST> ...
        <FACTOR> 48
<STRING_EXPRESSION>  42
```

```
<SYSSTATUS_STATEMENT>  15
        <STATEMENT>  1
<SYSTEM_STATEMENT>  21
        <STATEMENT>  1
<TERMINATE_STATEMENT>   4
        <STATEMENT>  1
<TIME_STATEMENT>  24
        <STATEMENT>  1
<TRACE_STATEMENT>   7
        <STATEMENT>  1
<TYPE>  33
        <CONST_VAR_DECL> 32
AGENT ...
        <TYPE> 33
BOOL ...
        <TYPE> 33
CONST ...
        <CONST_VAR_DECL> 32
DELETE ...
        <DELETE_STATEMENT> 36
DISPLAY ...
        <DISPLAY_STATEMENT>  2
DUMP ...
        <DUMP_STATEMENT> 23
EVENT ...
        <EVENT_STATEMENT> 11
HELP ...
        <HELP_STATEMENT> 28
INHERIT ...
        <SEND_STATEMENT> 34
INT ...
        <TYPE> 33
INTERFACE ...
        <AGENT_DENOTATION>  5 , <FACTOR> 48
LIMIT ...
        <LIMIT_STATEMENT> 26
MAILBOX ...
        <MAILBOX_STATEMENT> 27
NEW ...
        <FACTOR> 48
NOAGENT ...
        <FACTOR> 48
NOEVENT ...
        <NOEVENT_STATEMENT> 10
NOOBSERVE ...
        <NOOBSERVE_STATEMENT> 18
NOOPER ...
        <FACTOR> 48
NOPROT ...
        <NOPROT_STATEMENT> 16
NOSCRIPT ...
        <FACTOR> 48
NOSNAPSHOT ...
        <NOSNAPSHOT_STATEMENT> 12
NOSYSTEM ...
        <NOSYSTEM_STATEMENT> 20
NOTRACE ...
        <NOTRACE_STATEMENT>  6
```

OBSERVE ...
<OBSERVE_STATEMENT> 19
OPER ...
<OPER_DECL> 31 , <TYPE> 33
OWNMODE ...
<FACTOR> 48
PORT ...
<PORT_DECL> 30
PROT ...
<PROT_STATEMENT> 17
READ ...
<READ_STATEMENT> 3
REAL ...
<TYPE> 33
RECEIVE ...
<RECEIVE_STATEMENT> 37
REPLY ...
<SEND_STATEMENT> 34 , <REPLY_STATEMENT> 35
RUN ...
<RUN_STATEMENT> 25
SCRIPT ...
<TYPE> 33
SELF ...
<AGENT_DENOTATION> 5 , <FACTOR> 48
SEND ...
<SEND_STATEMENT> 34
SNAPSHOT ...
<SNAPSHOT_STATEMENT> 13
STACKS ...
<STACKS_STATEMENT> 22
START ...
<START_STATEMENT> 9
STATUS ...
<STATUS_STATEMENT> 14
STOP ...
<STOP_STATEMENT> 8 , <EVENT_STATEMENT> 11
STRING ...
<TYPE> 33
SYSSTATUS ...
<SYSSTATUS_STATEMENT> 15
SYSTEM ...
<SYSTEM_STATEMENT> 21
TERMINATE ...
<TERMINATE_STATEMENT> 4
TIME ...
<TIME_STATEMENT> 24
TO ...
<SEND_STATEMENT> 34
TRACE ...
<TRACE_STATEMENT> 7
VAR ...
<CONST_VAR_DECL> 32