TUM

INSTITUT FÜR INFORMATIK

INSEL Syntax-Bericht

Ralph Radermacher und Frank Weimer

TUM-I9617
März 1996

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-03-1996-I9617-350/1.-FI

Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

Inhaltsverzeichnis

Beispielverzeichnis

Kapitel 1

Einleitung

In diesem Bericht wird die Sprache INSEL[1] vorgestellt. Die Entwicklung von INSEL gründet sich auf Arbeiten, die an der Universität Oldenburg begonnen wurden und an der Technischen Universität München im Rahmen des MoDiS[2]–Projekts am Lehrstuhl Systemarchitektur fortgesetzt werden.

Das Projekt MoDiS befaßt sich mit der Entwicklung von Konzepten für die Konstruktion Verteilter Systeme nach einem Top–Down–Ansatz. Verteilte Systeme sind Rechensysteme, die ihren Benutzern eine homogene Schnittstelle für verteilte Problemlösungen anbieten und auf einer Hardware–Konfiguration aus vernetzten Stellenrechnern realisiert sind.

Die homogene Schnittstelle wird dem Benutzer durch die Sprache INSEL zur Verfügung gestellt. INSEL ist eine imperative Programmiersprache mit hohem Abstraktionsniveau. Ein Benutzer spezifiziert und programmiert ein verteiltes System als ein INSEL-Programm. Es wird also ein Ein–Programm–Ansatz verfolgt.

Die Sprache INSEL enthält Sprachkonstrukte, mit denen der Benutzer die abstrakte Parallelität seiner Problemlösung explizit angeben kann.

Der Schwerpunkt dieses Berichts liegt auf dem programmiersprachlichen Aspekt von INSEL. Die der Sprache zugrundeliegenden Konzepte werden detailliert in [Spi94] vorgestellt.

In Kapitel 2 werden die Sprachkonzepte und die Nomenklatur von INSEL eingeführt. Kapitel 3 enthält sowohl die Syntax von INSEL als auch eine informelle Beschreibung der Semantik. Darüberhinaus enthält dieses Kapitel Beispiele zu den wichtigsten Sprachkonstrukten.

[1] Integration– and Separation–supporting Experimental Language
[2] Model–oriented Distributed Systems

Kapitel 2

Die Sprache INSEL

Wie bereits in der Einleitung erwähnt wurde, soll die Sprache INSEL dazu dienen, verteilte Systeme zu spezifizieren und zu programmieren. Ein in Ausführung befindliches INSEL–Programm wird als INSEL–System bezeichnet. INSEL–Systeme sind aus Komponenten zusammengesetzt.

Bevor näher auf den Aufbau und die Entwicklungsmöglichkeiten von INSEL–Systemen eingegangen wird, werden zunächst die INSEL–Systemen zugrundeliegenden Prinzipien beschrieben. Diese sind das Objektprinzip, das Schachtelungsprinzip und das Prinzip der Klassenbildung.

Das Objektprinzip besagt, daß die Eigenschaften einer Komponente durch Operationen festgelegt sind, daß die inneren und äußeren Eigenschaften einer Komponente zu unterscheiden sind, und daß Komponenten von außen allein dadurch benutzbar sind, daß ihre äußeren Operationen ausgeführt werden.

Durch die Verwendung des Schachtelungsprinzips werden Komponenten während ihrer gesamten Existenz in andere Komponenten eingeordnet. Komponenten, die in andere Komponenten eingeordnet sind, sind ein Bestandteil dieser. Die konsequente Anwendung des Schachtelungsprinzips hat zur Folge, daß die Systeme aus einer ausgezeichneten Komponente bestehen, in die alle weiteren Komponenten eingeordnet sind. In INSEL–Systemen wird diese ausgezeichnete Komponente als Hauptkomponente bezeichnet.

Das Prinzip der Klassenbildung legt fest, daß die Komponenten eines Systems Elemente von Komponentenklassen sind. Diese Komponentenklassen werden in INSEL als Generatoren bezeichnet. Sie legen die Eigenschaften fest, die allen Elementen ihrer Klasse gemeinsam sind.

INSEL–Systeme sind aus Komponenten zusammengesetzt. Die Eigenschaften dieser Komponenten sowie deren Abhängigkeiten untereinander werden durch ein INSEL–Programm festgelegt.

Es gibt verschiedene Arten von Komponenten, zum einen die einfachen oder DE–Komponenten, für die alle Operationen vordefiniert sind, und zum anderen die wesentlichen oder DA–Komponenten, für die äußere und innere Operationen explizit definiert werden können. DA–Komponenten bestehen aus einem Deklarationsteil, der ihre lokalen Komponenten festlegt, und einem Anweisungsteil.

Eine weitere Klassifikation der Komponentenarten liefert die Art ihrer Identifikation. Es gibt benannte Komponenten, für die während ihrer gesamten Existenz ein in ihrem Kontext eindeutiger Name zugeordnet ist, und anonyme Komponenten, die mittels Zeigerwerten identifiziert werden. Benannte Komponenten werden durch die Erarbeitung einer Deklaration erzeugt, anonyme Komponenten durch die Auswertung eines Generierungsausdrucks[1].

Im folgenden werden zunächst die DE–Komponenten und dann die DA–Komponenten beschrieben.

2.1 DE–Komponenten

Wie schon beschrieben, sind für die DE–Komponenten alle Operationen vordefiniert. Die DE–Komponenten können gemäß Abbildung 2.1 klassifiziert werden, wobei die Aufteilung in Generatoren und wertorientierte DE–Komponenten wesentlich ist.

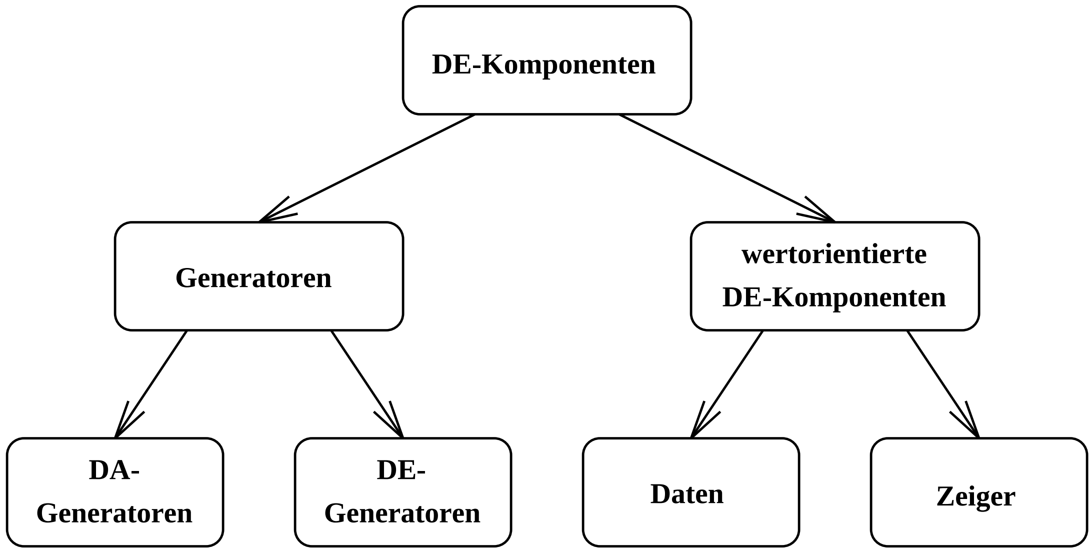

Abbildung 2.1: Klassifikation der DE–Komponenten

[1]Siehe dazu Regel 93 in Kapitel 3.

2.1.1 Generatoren

Die Generatoren sind die Komponenten, die Komponentenklassen definieren. Auf allen Generatoren ist eine Operation vordefiniert, die mit *Erzeuge* bezeichnet wird. Die Ausführung von *Erzeuge* bewirkt die Erzeugung einer Inkarnation, die ein Element der durch den Generator definierten Komponentenklasse ist. Es gibt Generatoren für DE–Komponenten und für DA–Komponenten.

Die Generatoren sind die einzigen Komponenten eines Systems, die durch die Erarbeitung ihrer Deklaration[2] erzeugt werden. Alle übrigen Komponenten werden erzeugt, indem die Operation *Erzeuge* auf einem Generator ausgeführt wird, der bereits als Komponente existiert.

Die Generatoren sind vergleichbar mit den Typen in anderen imperativen Programmiersprachen (vgl. Ada[3]).

2.1.2 Wertorientierte DE–Komponenten

Die wertorientierten DE–Komponenten lassen sich gemäß Abbildung 2.1 in Daten und Zeiger unterteilen.

Es gibt unterschiedliche Arten von Daten, deren Struktur und Zustandsmenge durch entsprechende DE–Generatoren festgelegt werden. Dazu stehen die folgenden Arten von DE–Generatoren zur Verfügung: vordefinierte Generatoren, Bereichsgeneratoren, Record–Generatoren und Feldgeneratoren.

Die vordefinierten Generatoren sind `boolean`, `character`, `integer`, `real` und `string`. Diese sind mit der für imperative Programmiersprachen üblichen Semantik definiert.

Mit den Bereichsgeneratoren können Wertebereiche von `character`, `integer` bzw. `real` definiert werden. Wertebereiche können entweder explizit durch Angabe ihres kleinsten und ihres größten Werts oder als Erweiterung eines bereits existierenden Bereichsgenerators von unten bzw. nach oben festgelegt werden.

Mit Record–Generatoren können wie üblich Mengen von Deklarationen von DE–Komponenten zu Einheiten zusammengefaßt werden.

Mit Feldgeneratoren können ein– oder mehrdimensionale Felder definiert werden. Dazu sind der Generator der Feldelemente und die Bereichsgeneratoren der einzelnen Dimensionen anzugeben.

---

[2] Eine Ausnahme bilden die Generatoren, die Inkarnationen bzgl. Generatoren zweiter Ordnung sind (siehe dazu Abschnitt 2.3).

[3] Ada is a trademark of the Department of Defense (Ada Joint Program Office).

Die Zeiger sind die Komponenten, welche zur Identifikation anonymer Komponenten benutzt werden können. Jeder Zeiger ist über seinen Generator mit einer Komponentenklasse qualifiziert.

Um die für rekursive Datenstrukturen benötigten wechselseitigen Bezüge zwischen Generatoren ausdrücken zu können, darf der Name eines DE–Zeigergenerators vor seiner eigentlichen Definition mittels einer Forward–Deklaration bekanntgemacht werden. Die endgültige Definition eines solchen Zeigergenerators muß im selben Deklarationsteil folgen.

2.2 DA–Komponenten

Die Eigenschaften von DA–Komponenten werden durch DA–Generatoren definiert. DA–Generatoren bestehen aus zwei Teilen: einem Spezifikationsteil und einem Implementationsteil.

Der Spezifikationsteil ist optional und wird nur für Generatoren solcher Komponenten benötigt, die Operationen nach außen anbieten sollen. Ein Spezifikationsteil besteht i. w. aus einer Menge von Deklarationen, die die Schnittstelle der entsprechenden DA–Komponenten definieren. Genau die Komponenten, deren Deklaration in einem Spezifikationsteil enthalten ist, sind von außen durch andere Komponenten benutzbar.

Der Implementationsteil, der für alle DA–Generatoren angegeben werden muß, enthält die Deklarationen, die nicht zur Schnittstelle nach außen gehören sollen, und den Anweisungsteil.

Ist für einen DA–Generator ein Spezifikationsteil angegeben, so muß der zu ihm gehörende Implementationsteil im selben Deklarationsteil folgen.

Durch die Aufteilung von DA–Generatoren in Spezifikationsteil und Implementationsteil ist es möglich, wechselseitige Abhängigkeiten zwischen DA–Generatoren zu spezifizieren.

Für DA–Generatoren können formale Parameter definiert werden. Es stehen Eingabe–, Ein–/Ausgabe– und Ausgabe–Parameter zur Verfügung. Die für die einzelnen Arten von DA–Komponenten erlaubten Parameterarten und –übergabemodi sind in Kapitel 3 auf Seite 19 angegeben.

Um die Zugriffsmöglichkeiten zu Komponenten einzuschränken, die in der Schachtelungsstruktur innen bzw. außen zum Generator liegen, können Import–Export–Beschränkungen für DA–Generatoren definiert werden. Mit den Import–Beschränkungen kann die Nutzung von Komponenten eingeschränkt werden, die gemäß der Schachtelungsstruktur außen liegen. Die Export–Beschränkungen können dazu verwendet werden, die Nutzungen lokaler Komponenten durch von in der Schachtelungsstruktur innen liegenden Komponenten einzuschränken. Die Syntax der Import–Export–Beschränkungen ist in Kapitel 3 auf Seite 19 angegeben. Für eine detaillierte Beschreibung der Import–Export–Beschränkungen siehe [Spi94].

Wie in Abbildung 2.2 dargestellt ist, können folgende Arten von DA–Komponenten unterschieden werden: Order, Depots und Akteure.

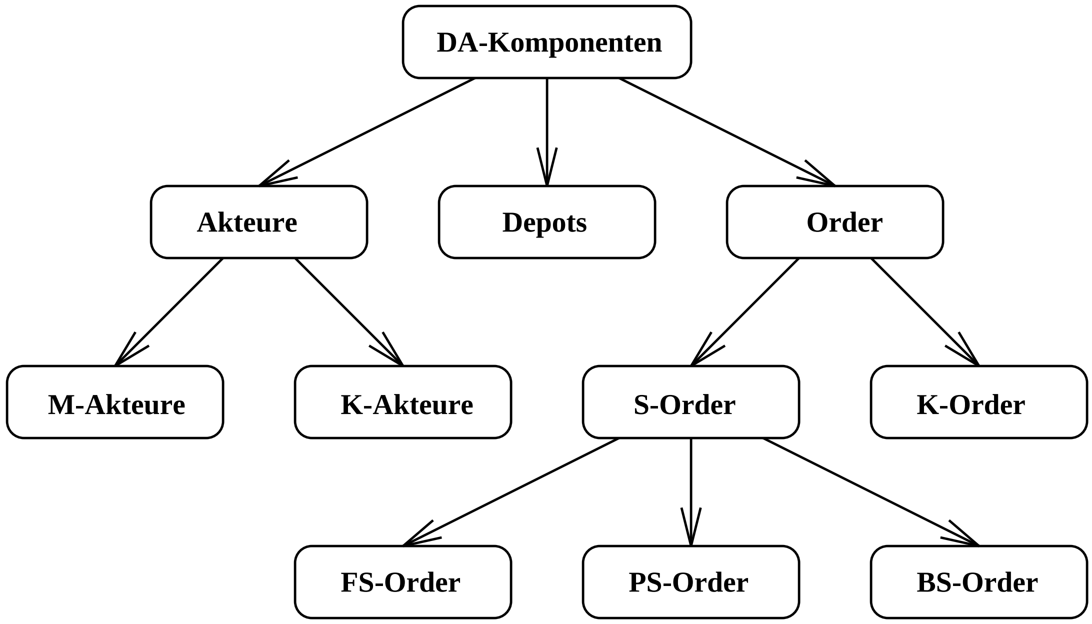

Abbildung 2.2: Klassifikation der DA–Komponenten

2.2.1 Order

Order sind operationendefinierende Komponenten. Sie legen i. w. ihre kanonische Operation fest. Order lassen sich weiter in S–Order und K–Order unterteilen. K–Order sind Kommunikationsoperationen und werden weiter unten im Zusammenhang mit den Akteuren behandelt. S–Order entsprechen den Unterprogrammen der üblichen imperativen Programmiersprachen. Es gibt drei Arten von S–Order: PS–Order, FS–Order und BS–Order. PS–Order haben die Eigenschaften von Prozeduren, FS–Order die von Funktionen und BS–Order die von Blöcken.

2.2.2 Depots

Depots sind speicherdefinierende Komponenten. Ein Depot definiert mit seinen lokalen Komponenten Speicher und Operationen für deren Benutzung. Während der Ausführung eines Depot–Anweisungsteils werden i. w. die lokalen Komponenten des Depots initialisiert. Nach der Beendigung des Anweisungsteils ist das Depot von außen benutzbar. Ein Depot bietet in erster Linie Zugriffoperationen in Form von FS– und PS–Order–Generatoren an. Depots sind vergleichbar mit den *packages* in Ada.

2.2.3 Akteure

Akteure sind diejenigen Komponenten eines INSEL–Systems, die abstrakte Rechenfähigkeit besitzen. Sie führen Berechnungen durch, indem sie ihren Anweisungsteil ausführen. Dabei können weitere DA–Komponenten erzeugt und aufgelöst werden. Ein Akteur, der während der Ausführung seines Anweisungsteils eine Order oder ein Depot erzeugt, führt dann deren bzw. dessen Anweisungsteil aus. Alle Akteure eines INSEL–Systems werden abstrakt parallel zueinander ausgeführt.

Die Akteure, für die Kommunikationsoperationen in Form von K–Order–Generatoren definiert werden können, werden als K–Akteure bezeichnet, und Akteure, für die dies konzeptionell nicht möglich ist, als M–Akteure. M–Akteure werden in erster Linie zur Parallelisierung von Berechnungen verwendet.

Die Kommunikationsoperationen, die K–Akteure nach außen anbieten, werden nach dem operationenorientierten Rendezvous–Konzept ausgeführt. Ein K–Akteur ist ein Auftragnehmer für die Kommunikationsoperationen, die er anbietet. Andere Akteure können Aufträge zur Ausführung von Kommunikationsoperationen erteilen, indem sie eine spezielle Anweisung, einen K–Order–Aufruf bzgl. eines K–Order–Generators, ausführen. Für jeden K–Order–Generator eines K–Akteurs ist implizit ein Warteraum für die zugehörenden K–Order definiert. Falls ein Akteur einen Auftrag bzgl. eines K–Order–Generators erteilt, wird eine entsprechende K–Order erzeugt und in den zugehörenden Warteraum eingefügt. Falls der K–Akteur, der den K–Order–Generator anbietet, auf die Erteilung eines Auftrags bzgl. dieses K–Order–Generators wartet, wird er aufgeweckt. Der Auftraggeber wird von der Ausführung suspendiert, und der Auftrag gilt damit als erteilt. Der Auftragnehmer führt dann die K–Order aus, die aus dem Warteraum entnommen wurde. Wenn die Ausführung der K–Order beendet ist, wird die Suspendierung des Auftraggebers aufgehoben, und der Auftrag gilt als erfüllt.

Falls ein K–Akteur eine K–Order–Annahmeanweisung bzgl. eines K–Order–Generators ausführt, wird eine K–Order aus dem zugehörenden Warteraum entnommen. Wenn dieser Warteraum leer ist, wartet der K–Akteur auf die Erteilung eines Auftrags bzgl. des in der Annahmeanweisung genannten K–Order–Generators.

Mit Hilfe der auswählenden Annahmeanweisung kann ein K–Akteur auf die Erteilung von Aufträgen bzgl. mehrerer seiner K–Order–Generatoren warten.

2.3 Generatoren zweiter Ordnung

Generatoren zweiter Ordnung sind, wie alle Generatoren, DE–Komponenten. Auf ihnen ist die Operation *Erzeuge* vordefiniert. Die Ausführung von *Erzeuge* auf einem Generator zweiter Ordnung bewirkt die Erzeugung eines Generators erster Ordnung. Es gibt Generatoren zweiter Ordnung für Depot–Generatoren und für K–Akteur–Generatoren.

Für Generatoren zweiter Ordnung können formale Parameter definiert werden. Der Parameterübergabemodus ist *Name*. Die Parameterarten sind

- formale DE–Generatoren,
- formale PS–Order–Generatoren und
- formale FS–Order–Generatoren.

Die Generatoren zweiter Ordnung sind mit den generischen Einheiten von Ada oder den Templates von C++ vergleichbar.

Die Syntax für Generatoren zweiter Ordnung und für die Bildung entsprechender Inkarnationen ist in Kapitel 3 auf Seite 21ff. angegeben. Für eine detaillierte Beschreibung siehe [Spi94].

Kapitel 3

INSEL-Sprachbeschreibung

.... **Hauptkomponente**

Das Nichtterminalzeichen <*system*> ist das Startsymbol für die Grammatik.

Ein INSEL-Programm wird durch einen M-Akteur-Generator, den Generator der Hauptkomponente, definiert. Dieser Generator besteht, wie alle DA-Generatoren, aus einem Deklarations- und einem Anweisungsteil. Für ihn sind, im Gegensatz zu den übrigen DA-Generatoren, keine formalen Parameter und keine Import-Export-Beschränkungen zugelassen.

Regel 1 <*system*> *::=*
    PROCESS <*identifier*> **IS**
        <*declarative-part*>
    BEGIN
        <*statement-part*>
    END <*opt-identifier*> ';'

```
PROCESS Hauptkomponente IS
    -- Deklarationsteil
BEGIN
    -- Anweisungsteil
END Hauptkomponente;
```

Deklarationsteil

Ein Deklarationsteil ist eine evtl. leere Folge von durch ';' abgeschlossenen Deklarationen.

Eine Deklaration definiert entweder einen Generator erster oder zweiter Ordnung oder ist eine Objekt-Deklaration.

Regel 2 *<declarative-part> ::=*
<empty>
| *<declarative-part>* *<declaration>* ';'

Regel 3 *<declaration> ::=*
<de-generator>
| *<da-generator>*
| *<generic-generator>*
| *<generic-generator-incarnation>*
| *<object-declaration>*

DA–Generatoren

Ein DA-Generator besteht aus einem Implementationsteil oder einem Spezifikations- und einem Implementationsteil. Falls ein Spezifikationsteil angegeben wird, muß der zu ihm gehörende Implementationsteil im selben Deklarationsteil folgen.

Regel 4 *<da-generator> ::=*
<specification-part>
| *<implementation-part>*

Der Spezifikationsteil eines DA–Generators legt als Klasseneigenschaft die Schnittstelle nach außen der bzgl. ihm erzeugten Inkarnationen fest. Die im Spezifikationsteil deklarierten Komponenten, und nur diese, sind von außen benutzbar. Falls ein DA–Generator nur aus einem Implementationsteil besteht, kann er keine Komponenten nach außen zur Benutzung anbieten.

Regel 5 *<specification-part> ::=*
<generator-type> **TYPE SPEC** *<identifier>*
<formal-parameter-part>
<limitation-part>
<return-part>
<interface-part>

```
DEPOT TYPE SPEC StackTyp (MaxTiefe : IN integer) IS
    FUNCTION  TYPE SPEC Pop RETURN integer;
    PROCEDURE TYPE SPEC Push (Wert : IN integer);
END StackTyp;
```

Die vollständige Definition eines DA–Generators erfolgt in seinem Implementationsteil. Falls ein Spezifikationsteil für ihn angegeben wurde, so kann auf eine Wiederholung von *<formal-parameter-part>*, *<limitation-part>* oder *<return-part>* verzichtet werden. Falls einer der oben genannten Teile wiederholt wird, muß er mit seiner Entsprechung im Spezifikationsteil übereinstimmen.

Regel 6 *<implementation-part> ::=*
<generator-type> **TYPE** *<identifier>*
<formal-parameter-part> *<limitation-part>*
<return-part>
IS *<declarative-part>*
BEGIN *<statement-part>* **END** *<opt-identifier>*

```
DEPOT TYPE StackTyp (MaxTiefe : IN integer) IS

             TYPE IndexTyp  IS 1 .. MaxTiefe;
    EXTENDED TYPE Index0Typ IS IndexTyp FROM 0;
             TYPE FeldTyp   IS ARRAY [IndexTyp] OF integer;

    Feld     : FeldTyp;
    Position : Index0Typ;

    FUNCTION TYPE Pop RETURN integer IS
    BEGIN
        Position := Position - 1;
        RETURN Feld [Position + 1];
    END Pop;

    PROCEDURE TYPE Push (Wert : IN integer) IS
    BEGIN
        Position := Position + 1;
        Feld [Position] := Wert;
    END Push;

BEGIN
    Position := 0;
END StackTyp;
```

DA–Generatorarten

Das Nichtterminalsymbol <*generator-type*> legt entsprechend der folgenden Tabelle die Generator-Art fest.

Schlüsselwort	Generator–Art
PROCESS	M–Akteur–Generator
TASK	K–Akteur–Generator
DEPOT	Depot–Generator
PROCEDURE	PS–Order–Generator
FUNCTION	FS–Order–Generator
ENTRY	K–Order–Generator

Regel 7 <*generator-type*> *::=*
PROCESS
| **TASK**
| **DEPOT**
| **PROCEDURE**
| **FUNCTION**
| **ENTRY**

Schnittstelle nach außen

Ein nichtleerer <*interface-part*> ist nur für K–Akteur– und Depot–Generatoren zulässig. Für einen K–Akteur–Generator sind hier genau seine K–Order–Generatoren anzugeben. Für Depot–Generatoren sind alle Arten von Deklarationen erlaubt.

Regel 8 <*interface-part*> *::=*
<*empty*>
| **IS** <*declarative-part*> **END** <*opt-identifier*>

Rückgabewertgenerator für FS–Order–Generatoren

Nur für FS–Order–Generatoren ist ein <*return-part*> anzugeben. In diesem Fall legt der <*return-part*> den Generatornamen für den Rückgabewert fest.

Regel 9 <*return-part*> *::=*
<*empty*>
| **RETURN** <*name*>

Import–Export–Beschränkungen

Der <*limitation-part*> gibt die Import–Export–Beschränkungen für einen DA–Generator an. Es darf höchstens eine Import– und höchstens eine Export–Beschränkung in einem <*limitation-part*> angegeben werden.

Regel 10 <*limitation-part*> *::=*
<*empty*>
| <*limitation-part*> <*limitation*> ';'

Regel 11 <*limitation*> *::=*
IMPORT <*name-list*>
| **IMPORT NONE**
| **EXPORT** <*identifier-list*>
| **EXPORT NONE**

formale Parameter

Der <*formal-parameter-part*> legt die formalen Parameter erster Ordnung eines DA–Generators fest. Ein nichtleerer <*formal-parameter-part*> besteht aus einer in runde Klammern eingeschlossenen Liste von formalen Parametern. Die formalen Parameter werden durch ';' voneinander getrennt. Für einen formalen Parameter sind sein Übergabemodus und sein Generatorname anzugeben. Mehrere formale Parameter, deren Übergabemodi und Generatornamen übereinstimmen, können gemäß Regel 14 zusammengefaßt werden. Für Parameter erster Ordnung stehen die in der folgenden Tabelle angegebenen Übergabemodi zur Verfügung.

Schlüsselwort	Übergabemodus
IN	`value`
IN OUT	`value-result`
OUT	`result`

Nicht alle Generatorarten und Übergabemodi sind als formale Parameter erster Ordnung von DA–Generatoren zugelassen. Die diesbezüglichen Restriktionen sind in der folgenden Tabelle zusammengefaßt.

DA-Generator-Art	**Parameter erster Ordnung**	
	Generator-Arten	**Übergabemodi**

M–Akteur	Daten
	Zeiger
		`value`
		`value-result`
		`result`

K–Akteur	Daten DA-Zeiger	`value`
FS–Order	Daten Zeiger	`value`

PS–Order	Daten
	Zeiger
		`value`
		`value-result`
		`result`

BS–Order	keine	—

K–Order	Daten
	Zeiger
		`value`
		`value-result`
		`result`

Depot	Daten DA-Zeiger	`value`

Regel 12 *<formal-parameter-part> ::=*
<empty>
| '(' *<formal-parameter-list>* ')'

Regel 13 *<formal-parameter-list> ::=*
<formal-parameter>
| *<formal-parameter-list>* ';' *<formal-parameter>*

Regel 14 *<formal-parameter> ::=*
<identifier-list> ':' *<parameter-mode>* *<name>*

Regel 15 *<parameter-mode> ::=*
IN
| **OUT**
| **IN OUT**

Bezeichner– und Namenslisten

Regel 16 *<identifier-list> ::=*
  <identifier>
  | *<identifier-list>* **','** *<identifier>*

Regel 17 *<name-list> ::=*
  <name>
  | *<name-list>* **','** *<name>*

Regel 18 *<opt-identifier> ::=*
  <empty>
  | *<identifier>*

Generatoren zweiter Ordnung

Die Definition eines Generators zweiter Ordnung wird durch das Schlüsselwort **GENERIC** eingeleitet. Darauf folgt eine evtl. leere Liste von Parametern zweiter Ordnung, die jeweils durch **';'** abgeschlossen sind. Die Definition wird durch die Angabe eines DA–Generators gemäß Regel 4 beendet.
Als DA–Generatoren sind hier ausschließlich K–Akteur– und Depot–Generatoren erlaubt.

Regel 19 *<generic-generator> ::=*
  GENERIC *<formal-generic-parameter-part>* *<da-generator>*

Regel 20 *<formal-generic-parameter-part> ::=*
  <empty>
  | *<formal-generic-parameter-part>* *<formal-generic-parameter>* **';'**

Es gibt drei Arten von formalen Parametern zweiter Ordnung. Diese sind

- DE–Generatoren,
- FS–Order–Generatoren und
- PS–Order–Generatoren.

Für K–Akteur–Generatoren zweiter Ordnung sind nur DE–Generatoren als Parameter zweiter Ordnung zugelassen. Für Depot–Generatoren zweiter Ordnung hingegen sind alle der oben genannten Parameterarten möglich.

Regel 21 <*formal-generic-parameter*> *::=*
WITH TYPE <*identifier*>
| **WITH FUNCTION TYPE** <*identifier*>
<*formal-parameter-part*> **RETURN** <*name*>
| **WITH PROCEDURE TYPE** <*identifier*>
<*formal-parameter-part*>

```
GENERIC
    WITH TYPE ElementTyp;
    WITH FUNCTION TYPE Kleiner (A, B : IN ElementTyp) RETURN boolean;
DEPOT TYPE SPEC AllgemeinerSortierer (Groesse : IN integer) IS
    TYPE FeldIndex IS 1..Groesse;
    TYPE FeldTyp   IS ARRAY [FeldIndex] OF ElementTyp;
    PROCEDURE TYPE SPEC Sortiere (Feld : IN FeldTyp);
END AllgemeinerSortierer;

GENERIC DEPOT TYPE AllgemeinerSortierer IS
    PROCEDURE TYPE Sortiere (Feld : IN FeldTyp) IS
    BEGIN
        -- ...
    END Sortiere;
BEGIN
END AllgemeinerSortierer;
```

Inkarnationen bzgl. Generatoren zweiter Ordnung

Inkarnationen bzgl. Generatoren zweiter Ordnung sind Generatoren erster Ordnung. Solche Inkarnationen werden gemäß Regel 22 gebildet.
Durch <*generator-type*> wird die Art des erzeugten DA-Generators festgelegt. Hier sind, wie oben bereits gesagt wurde, nur **DEPOT** und **TASK** erlaubt. Der auf das Schlüsselwort **TYPE** folgende <*identifier*> gibt den Namen des neuen DA–Generators an. Der Name des Generators zweiter Ordnung, bzgl. dessen die Inkarnation erzeugt werden soll, wird durch <*name*> spezifiziert. Die aktuellen Parameter zweiter Ordnung werden durch <*actual-generic-parameter-part*> festgelegt.

Regel 22 <*generic-generator-incarnation*> *::=*
NEW <*generator-type*> **TYPE** <*identifier*> **IS**
<*name*> <*actual-generic-parameter-part*>

Ein nichtleerer <*actual-generic-parameter-part*> ist eine Liste von Generatornamen, die in runde Klammern eingeschlossen ist.

Regel 23 *<actual-generic-parameter-part> ::=*
<empty>
| '(' *<name-list>* ')'

```
PROCESS SortiererBeispiel IS  -- Definition des Generators zweiter
                              -- Ordnung AllgemeinerSortierer (s.o.)

    FUNCTION TYPE IntegerKleiner (A, B : IN integer) RETURN boolean IS
    BEGIN
        RETURN A < B;
    END IntegerKleiner;

    NEW DEPOT TYPE IntegerSortierer IS
                       AllgemeinerSortierer (integer, IntegerKleiner);

    DEPOT IS10 : IntegerSortierer (10);
    Feld       : IS10.FeldTyp;

BEGIN
    IS10.Sortiere (Feld);
END SortiererBeispiel;
```

DE–Generatoren

Die Definition eines DE–Generators erfolgt gemäß Regel 24. Dabei sind nicht alle syntaktisch ableitbaren Kombinationen zulässig. Die folgende Tabelle faßt die erlaubten Kombinationen zusammen.

<type-definition-part>	*<type-prefix>*		
	<empty>	**EXTENDED**	**POINTER**
<empty>	—	—	Forward-Deklaration
<range-specification>	Bereichs-Generator	Bereichserweiterungsgenerator	—
<array-type-definition>	Feld-Generator	—	—
<record-type-definition>	Record-Generator	—	—
<pointer-type-definition>	—	—	Zeiger-Generator

Hierbei sind die Forward–Deklarationen und die Bereichserweiterungsgeneratoren hervorzuheben.

Eine Forward–Deklaration wird dazu verwendet, den Namen eines Generators vor seiner eigentlichen Deklaration, die im selben Deklarationsteil folgen muß, bekanntzumachen. Damit werden wechselseitige Referenzen zwischen Generatoren möglich. Forward–Deklarationen dienen insbesondere der Konstruktion rekursiver Datenstrukturen. Aus diesem Grunde sind für sie nur Zeiger–Generatoren zugelassen.
Bereichsgeneratoren und Bereichserweiterungsgeneratoren definieren Wertebereiche. Mit einem Bereichserweiterungsgenerator wird ein Wertebereich definiert, der den Wertebereich eines bereits existierenden Bereichs– oder Bereichserweiterungsgenerators von unten oder nach oben erweitern kann.

Regel 24 *<de-generator> ::=*
<type-prefix> **TYPE** *<identifier>* *<type-definition-part>*

Regel 25 *<type-prefix> ::=*
<empty>
| **EXTENDED**
| **POINTER**

Regel 26 *<type-definition-part> ::=*
<empty>
| **IS** *<type-definition>*

Regel 27 *<type-definition> ::=*
<range-specification>
| *<array-type-definition>*
| *<record-type-definition>*
| *<pointer-type-definition>*

Bereichsgeneratoren

Mit *<range-specification>* können Wertebereiche definiert werden. Dazu stehen zwei Alternativen zur Verfügung. Zum einen die einfachen Wertebereiche, die durch die Angabe ihres kleinsten und ihres größten Wertes festgelegt werden. Zum anderen können Erweiterungen bzgl. bereits existierender Wertebereiche vorgenommen werden. Dabei sind der Name des zugrundeliegenden Bereichs– oder Bereichserweiterungsgenerators und optional die Erweiterungen anzugeben. Die Erweiterung von unten erfolgt mit Regel 29. Der dort angegebene Wert darf nicht größer sein, als der kleinste Wert des zugrundeliegenden Wertebereichs. Analoges gilt für die Erweiterung nach oben mit Regel 30.

Regel 28 <*range-specification*> *::=*
<*simple-expression*> **'..'** <*simple-expression*>
| <*name*> <*from-part*> <*to-part*>

Regel 29 <*from-part*> *::=*
<*empty*>
| **FROM** <*simple-expression*>

Regel 30 <*to-part*> *::=*
<*empty*>
| **TO** <*simple-expression*>

```
TYPE NaturalBereich   IS  0  ..  2 ** 16 - 1;
TYPE RealBereich      IS 0.0 .. 1.0;
TYPE CharacterBereich IS 'a' .. 'z';


EXTENDED TYPE IntegerBereich IS NaturalBereich FROM -(2 ** 16);
EXTENDED TYPE ProzentBereich IS RealBereich    TO   100.0;


         TYPE KleinerBereich        IS 1 .. 10;
EXTENDED TYPE GroessererBereich     IS KleinerBereich    FROM -10;
EXTENDED TYPE NochGroessererBereich IS GroessererBereich TO   100;
```

Feldgeneratoren

Feldgeneratoren werden gemäß Regel 31 definiert. Dazu sind sowohl die Generatornamen für die Wertebereiche der einzelnen Dimensionen als auch der Name des Generators für die Feldelemente anzugeben. Als Generator der Feldelemente ist jede Art von DE–Generator zulässig.

Regel 31 <*array-type-definition*> *::=*
ARRAY '[' <*name-list*> **']' OF** <*name*>

```
TYPE Dimension1 IS 1 .. 10;
TYPE Dimension2 IS 'A' .. 'Z';

TYPE FeldTyp IS ARRAY [Dimension1, Dimension2] OF real;

Summe : real;
Feld  : FeldTyp;

BEGIN
    -- ...

    Summe := 0.0;

    FOR I IN Dimension1
    LOOP
        FOR J IN Dimension2
        LOOP
            Summe := Summe + Feld [I, J];
        END LOOP;
    END LOOP;

    -- ...
END ...;
```

.... **Record–Generatoren**

Mit Regel 32 werden Record–Generatoren definiert. Ihre Komponenten werden gemäß Regel 33 als Liste von DE–Objektdeklarationen festgelegt. Nicht alle Möglichkeiten, die <*de-object-declaration*> (siehe Regel 36) zuläßt, sind hier erlaubt. Die Teile <*constant-part*> und <*init-part*> von Regel 36 müssen leer sein. Für <*name*> ist jede Art von DE–Generator zulässig.

Regel 32 <*record-type-definition*> *::=*
RECORD <*de-object-declaration-list*> **END RECORD**

Regel 33 <*de-object-declaration-list*> *::=*
<*de-object-declaration*> ';'
| <*de-object-declaration-list*> <*de-object-declaration*> ';'

```
TYPE TagTyp   IS 1 .. 31;
TYPE MonatTyp IS 1 .. 12;
TYPE JahrTyp  IS 0 .. 4711;

TYPE DatumTyp IS
    RECORD
        Tag   : TagTyp;
        Monat : MonatTyp;
        Jahr  : JahrTyp;
    END RECORD;

TYPE ZeitraumTyp IS
    RECORD
        Anfang : DatumTyp;
        Ende   : DatumTyp;
    END RECORD;

TYPE KomplexeZahlTyp IS
    RECORD
        Re, Im : real;
    END RECORD;
```

Zeigergeneratoren

Sowohl DE– als auch DA–Zeigergeneratoren werden mit <*pointer-type-definition*> definiert. Die Qualifikation des Zeigergenerators wird durch <*name*> bestimmt. Als Qualifikation sind alle Arten von DE–Generatoren sowie K–Akteur– und Depot–Generatoren zulässig.

Regel 34 <*pointer-type-definition*> *::=*
ACCESS <*name*>

```
POINTER TYPE ZeigerTyp;   -- Forward-Deklaration von ZeigerTyp

TYPE KnotenTyp IS
    RECORD
        Wert         : integer;
        LinkeTochter : ZeigerTyp;
        RechterSohn  : ZeigerTyp;
    END RECORD;

POINTER TYPE ZeigerTyp IS ACCESS KnotenTyp;
```

Objektdeklarationen

Eine Objektdeklaration ist entweder eine DE– oder eine DA–Objektdeklaration (siehe Regel 36 und 39).

Regel 35 *<object-declaration> ::=*
<de-object-declaration>
| *<da-object-declaration>*

DE–Objektdeklarationen

Eine DE–Objektdeklaration besteht aus vier Teilen:

- einer nichtleeren Liste von Bezeichnern, die die Namen der deklarierten Objekte festlegt,
- dem Schlüsselwort **CONSTANT**, falls es sich um Konstanten handelt,
- dem Namen des zugrundeliegenden DE–Generators und
- einem Initialisierungsteil, der für Konstanten nicht leer sein darf.

Regel 36 *<de-object-declaration> ::=*
<identifier-list> ':' *<constant-part>* *<name>* *<init-part>*

Regel 37 *<constant-part> ::=*
<empty>
| **CONSTANT**

Regel 38 *<init-part> ::=*
<empty>
| ':=' *<expression>*

```
Bereich : BereichsTyp;
A, B, C : integer       := 42;
Flag    : boolean       := true;
Text    : string        := "INSEL ist toll! ;-)";
Char    : character     := '*';
Pi      : CONSTANT real := 3.1415927;
```

DA-Objektdeklarationen

Eine DA–Objektdeklaration dient zur Deklaration von benannten DA–Komponenten. Sie besteht, wie eine DE–Objektdeklaration, aus vier Teilen:

- der Komponentenart, wobei nur die Schlüsselworte **DEPOT** und **TASK** möglich sind,
- einer nichtleeren Liste von Bezeichnern, die die Namen der deklarierten Objekte festlegt,
- dem Namen des zugrundeliegenden DA–Generators und
- einer Liste von aktuellen Parametern.

Regel 39 *<da-object-declaration> ::=*
<generator-type> <identifier-list> ':' *<name>*
<actual-parameter-part>

Regel 40 *<actual-parameter-part> ::=*
<empty>
| '(' *<expression-list>* ')'

Regel 41 *<expression-list> ::=*
<expression>
| *<expression-list>* ',' *<expression>*

```
DEPOT Stack  : StackTyp  (13);
TASK  Puffer : PufferTyp (200);
```

Anweisungsteil

Ein Anweisungsteil ist eine evtl. leere Folge von durch ';' abgeschlossenen Anweisungen. Die Menge der Anweisungsarten zerfällt in drei Klassen:

- DA–Komponenten–orientierte Anweisungen (siehe Regel 44),
- zusammengesetzte Anweisungen (siehe Regel 45) und
- einfache Anweisungen (siehe Regel 46).

Der Aufbau und die Wirkung der einzelnen Anweisungsarten werden bei den jeweiligen Regeln erläutert.

Regel 42 *<statement-part> ::=*
<empty>
| *<statement-part> <statement>* ';'

Regel 43 *<statement> ::=*
<da-related-statement>
| *<compound-statement>*
| *<simple-statement>*

.... **DA–Komponenten–orientierte Anweisungen**

Regel 44 *<da-related-statement> ::=*
<procedure-or-entry-call>
| *<actor-call>*
| *<accept-statement>*
| *<select-statement>*
| *<block-statement>*
| *<for-statement>*
| *<forall-statement>*
| *<exit-statement>*

.... **zusammengesetzte Anweisungen**

Regel 45 *<compound-statement> ::=*
<loop-statement>
| *<while-statement>*
| *<if-statement>*
| *<case-statement>*

.... **einfache Anweisungen**

Regel 46 *<simple-statement> ::=*
<assignment>
| *<return-statement>*
| *<input-statement>*
| *<output-statement>*
| *<incomplete-statement>*
| *<empty-statement>*

PS– oder K–Order–Aufrufe

Eine PS– oder K–Order wird erzeugt, indem ein PS– oder K–Order–Aufruf ausgeführt wird. Hierfür sind der Name des entsprechenden Order–Generators und die aktuellen Parameter anzugeben.

Regel 47 *<procedure-or-entry-call> ::=*
<name> <actual-parameter-part>

```
TYPE FeldIndex IS 1 .. N;
TYPE FeldTyp   IS ARRAY [FeldIndex] OF integer;

PROCEDURE TYPE Sortiere (Feld : IN OUT FeldTyp) IS
BEGIN
    -- ...
END Sortiere;

Feld : FeldTyp;

BEGIN
    -- ...

    Sortiere (Feld);  -- PS-Order-Aufruf

    -- ...
END ...;
```

M–Akteur–Aufruf

Durch die Ausführung eines M–Akteur–Aufrufs wird ein M–Akteur erzeugt. Hier sind, analog zum PS– oder K–Order–Aufruf, der Name des M–Akteur–Generators und die aktuellen Parameter anzugeben.

Regel 48 *<actor-call> ::=*
FORK *<name> <actual-parameter-part>*

```
PROCESS TYPE Faktultaet (N : IN integer; Ergebnis : OUT integer) IS
BEGIN
    Ergebnis := 1;
    FOR I IN 1..N LOOP
        Ergebnis := Ergebnis * I;
    END LOOP;
END Fakultaet;

Erg : integer;

BEGIN
    FORK Fakultaet (7, Erg);  -- M-Akteur-Aufruf
END ...;
```

Annahmeanweisung

Eine Annahmeanweisung darf nur in Anweisungsteilen von K–Akteur–Generatoren enthalten sein. Insbesondere darf sie nicht in Blöcken (Regel 56) oder For–Schleifen (Regel 57) auftreten, auch wenn diese Bestandteil des Anweisungsteils eines K–Akteur–Generators sind, da die Ausführung dieser Anweisungen als BS–Order erfolgt.
Der auf das Schlüsselwort **ACCEPT** folgende Bezeichner gibt den Namen eines lokalen K–Order–Generators an, bzgl. dessen eine Annahme erfolgen soll.

Regel 49 *<accept-statement> ::=*
ACCEPT *<identifier>*

```
TASK TYPE SPEC WechselseitigerAusschluss IS
    ENTRY TYPE SPEC P;
    ENTRY TYPE SPEC V;
END WechselseitigerAusschluss;

TASK TYPE WechselseitigerAusschluss IS
    ENTRY TYPE P IS BEGIN END P;
    ENTRY TYPE V IS BEGIN END V;
BEGIN
    LOOP
        ACCEPT P;
        ACCEPT V;
    END LOOP;
END WechselseitigerAusschluss;
```

.... $\boxed{\textbf{auswählende Annahmeanweisung}}$

Wie schon die Annahmeanweisung, darf auch die auswählende Annahmeanweisung nur im Anweisungsteil von K–Akteur–Generatoren enthalten sein. Mit ihr kann flexibel auf die Erteilung von Aufträgen bzgl. mehrerer K–Order–Generatoren gewartet und reagiert werden.

Eine auswählende Annahmeanweisung gliedert sich in eine Menge von Auswahlalternativen und einen optionalen **ELSE**–Teil auf. Eine Auswahlalternative ist entweder eine Annahmealternative oder eine **TERMINATE**–Alternative, wobei beide auch bedingt sein können. Es darf höchstens eine **TERMINATE**–Alternative angegeben werden, und diese muß dann die letzte Auswahlalternative sein.

Eine Annahmealternative besteht aus einer Annahmeanweisung und einem eventuell leeren Anweisungsteil. Eine auswählende Annahmeanweisung wird in mehreren Phasen ausgeführt.

In der ersten Phase werden die wählbaren Auswahlalternativen bestimmt. Eine Auswahlalternative ist wählbar, wenn sie entweder unbedingt ist oder ihre Bedingung erfüllt ist. Falls es keine wählbare Alternative gibt und der **ELSE**–Teil nicht leer ist, werden die Anweisungen des **ELSE**–Teils ausgeführt. Der Fall, daß es keine wählbaren Alternativen gibt und der **ELSE**–Teil leer ist, ist unzulässig.

In der zweiten Phase wird aus der Menge der wählbaren die Menge der ausführbaren Auswahlalternativen bestimmt. Eine ausführbare Auswahlalternative ist entweder eine ausführbare Annahmealternative oder eine ausführbare **TERMINATE**–Alternative. Eine Annahmealternative ist ausführbar, wenn der zu ihr gehörende Warteraum nicht leer ist. Dieser Warteraum ist der Warteraum des K–Order–Generators, der in der entsprechenden Annahmeanweisung genannt wird.

Falls die Menge der ausführbaren Annahmealternativen nicht leer ist, wird eine dieser Alternativen ausgewählt[1] und ausgeführt. Andernfalls wird die Ausführbarkeit der **TERMINATE**–Alternative überprüft. Dies ist der Fall, wenn die Terminierungsbedingung erfüllt ist. Diese Bedingung wird hier nur informell beschrieben. Sie ist erfüllt, wenn der K–Akteur, der die auswählende Annahmeanweisung ausführt, von keinem seiner potentiellen Auftraggeber mehr einen Auftrag erhalten kann. Falls die Terminierungsbedingung erfüllt ist, wird der K–Akteur in der Zustand *terminiert* überführt. Im anderen Fall wartet er bzgl. seiner ausführbaren Auswahlalternativen.

Die auswählende Annahmeanweisung mit **TERMINATE**–Alternative ist insbesondere bei der Konstruktion von Server–Akteuren nützlich. Derartige Server–Akteure lassen sich dadurch charakterisieren, daß sie eine Menge von Diensten in Form von K–Order–Generatoren anbieten und daß sie ihre Dienste während eines wesentlichen Anteils ihrer

[1] Die Festlegung des Auswahlmechanismus ist nicht Bestandteil der Sprache, d. h. es können keine a priori Annahmen über Ausführungsreihenfolgen gemacht werden.

Existenz bereitstellen. Die Erteilung von Aufträgen bzgl. der angebotenen Dienste erfolgt asynchron, so daß keine a priori Annahmen über ihre Reihenfolge und Anzahl gemacht werden können.

Der Anweisungsteil eines Server–Akteurs besteht in der Regel nur aus einer auswählenden Annahmeanweisung, die in einer Endlosschleife enthalten ist. Mit einer **TERMINATE**–Alternative kann ein so konstruierter Server–Akteur beendet werden, wenn seine Dienste von keinem anderen Akteur mehr genutzt werden können.

Regel 50 *<select-statement> ::=*
SELECT
<select-alternatives>
<select-else-part>
END SELECT

Regel 51 *<select-alternatives> ::=*
<select-alternative>
| *<select-alternatives>* **OR** *<select-alternative>*

Regel 52 *<select-alternative> ::=*
<when-part> *<accept-alternative>*
| *<when-part>* **TERMINATE ';'**

Regel 53 *<when-part> ::=*
<empty>
| **WHEN** *<condition>* **DO**

Regel 54 *<accept-alternative> ::=*
<accept-statement> **';'**
<statement-part>

Regel 55 *<select-else-part> ::=*
<empty>
| **ELSE** *<statement-part>*

```
TASK TYPE SPEC PufferTyp (Kapazitaet : IN integer) IS
    ENTRY TYPE SPEC Lies     (Wert : OUT integer);
    ENTRY TYPE SPEC Schreibe (Wert : IN  integer);
END PufferTyp;

TASK TYPE PufferTyp IS

    TYPE FeldIndex IS 0 .. Kapazitaet-1;
    TYPE FeldTyp   IS ARRAY [FeldIndex] OF integer;

    Feld                          : FeldTyp;
    LesePosition, SchreibPosition : FeldIndex := 0;
    Elemente                      : integer   := 0;

    ENTRY TYPE Lies (Wert : OUT integer) IS
    BEGIN
        Wert := Feld [LesePosition];
        LesePosition := (LesePosition + 1) MOD Kapazitaet;
    END Lies;

    ENTRY TYPE Schreibe (Wert : IN integer) IS
    BEGIN
        Feld [SchreibPosition] := Wert;
        SchreibPosition := (SchreibPosition + 1) MOD Kapazitaet;
    END Schreibe;

BEGIN
    LOOP
        SELECT
               WHEN Elemente < Kapazitaet DO
                   ACCEPT Schreibe; Elemente := Elemente + 1;
            OR WHEN Elemente > 0 DO
                   ACCEPT Lies;     Elemente := Elemente - 1;
            OR
               TERMINATE;
        END SELECT;
    END LOOP;

END PufferTyp;
```

Block–Anweisungen

Die Block–, For– und For All–Anweisungen nehmen eine Sonderstellung ein. Mit ihnen werden BS–Order–Generatoren definiert. Eine BS–Order–Inkarnation wird durch die Ausführung einer entsprechenden Anweisung erzeugt. Im Gegensatz zu der Block–

Anweisung, für die ein Deklarationsteil explizit angegeben werden kann, haben die For– und For All–Anweisungen einen implizit definierten Deklarationsteil.

Eine Block–Anweisung ist gemäß Regel 56 aufgebaut. Der auf das Schlüsselwort **BLOCK** folgende Bezeichner legt den Namen des mit ihr definierten BS–Order–Generators fest. Der Name ist insbesondere für die Exit–Anweisung relevant (siehe Regel 61). Desweiteren besteht eine Block–Anweisung aus einem beliebigen Deklarations– und Anweisungsteil. Sie wird hauptsächlich für lokale Deklarationen und zur Synchronisation mit M–Akteuren benutzt. Eine Block–Inkarnation wird erst dann beendet, wenn alle von ihr abhängen-den Akteure terminiert sind, d. h. nach der Ausführung der Block–Anweisung stehen die Resultate der in ihrem Anweisungsteil gestarteten Akteure zur Verfügung.

Regel 56 *<block-statement> ::=*
BLOCK *<identifier>* **IS**
<declarative-part>
BEGIN
<statement-part>
END *<opt-identifier>*

```
A, B, C : integer;

BEGIN
    BLOCK Binom IS
        X, Y : integer;
    BEGIN
        INPUT X;
        INPUT Y;

        FORK Fakultaet (X,   A);
        FORK Fakultaet (Y,   B);
        FORK Fakultaet (X-Y, C);
    END Binom;

    OUTPUT A / (B * C);
END ...;
```

Eine For–Anweisung definiert einen speziellen BS–Order–Generator. Wie oben bereits gesagt wurde, besitzt sie einen implizit festgelegten Deklarationsteil, der die Deklaration des Schleifenzählers und ggf. die Definition des zu ihm gehörenden Bereichsgenerators[2]

[2] Die implizite Definition eines Bereichsgenerators ist notwendig, falls der angegebene Wertebereich explizit oder durch eine Erweiterung eines bereits existierenden Bereichsgenerators festgelegt ist (vgl. Regel 28).

enthält. Der Schleifenzähler ist für die Anweisungen des Schleifenkörpers eine Konstante, Wertzuweisungen an ihn sind dort also unzulässig.

Die For–Anweisung dient zur wiederholten Ausführung ihres Anweisungsteils. Die Anzahl der Wiederholungen wird durch den Wertebereich bestimmt. Der Schleifenzähler wird zu Beginn mit dem kleinsten Wert des Wertebereichs initialisiert und nach jeder Iteration inkrementiert. For–Anweisungen können, wie auch die Blöcke, benannt sein (siehe Exit-Anweisung, Regel 61).

Regel 57 *<for-statement>* ::=
<label-part> **FOR** *<identifier>* **IN** *<range-specification>*
LOOP
<statement-part>
END LOOP *<opt-identifier>*

Regel 58 *<label-part>* ::=
<empty>
| *<identifier>* ':'

```
TYPE FeldIndex IS 1 .. N;
TYPE FeldTyp   IS ARRAY [FeldIndex] OF integer;

Feld  : FeldTyp;
Summe : integer := 0;

BEGIN
    -- ...

    FOR Index IN FeldIndex LOOP
        Summe := Summe + Feld [Index];
    END LOOP;

    -- ...
END ...;
```

Eine For All–Anweisung definiert ebenfalls einen speziellen BS–Order–Generator. Für den Namen, den Schleifenzähler und den Wertebereich gilt das für die For–Anweisung (siehe Regel 57) Gesagte. Der Anweisungsteil besteht aus einer nichtleeren Folge von M–Akteur–Aufrufen.

Die For All–Anweisung dient zum nichtdeterministischen Starten[3] von M–Akteuren.

[3] Es können also keine a priori Aussagen über die Ausführungsreihenfolgen der Aufruf-Anweisungen gemacht werden.

Regel 59 *<forall-statement> ::=*
<*label-part*> **FOR ALL** <*identifier*> **IN** <*range-specification*>
DO
<*actor-calls*>
END FOR ALL <*opt-identifier*>

Regel 60 *<actor-calls> ::=*
<*actor-call*> ';'
| <*actor-calls*> <*actor-call*> ';'

```
TYPE FeldIndex IS 1 .. N;
TYPE FeldTyp   IS ARRAY [FeldIndex] OF integer;


Feld  : FeldTyp;


BEGIN
    -- ...

    FOR ALL Index IN FeldIndex
    DO
        FORK Fakultaet (Index, Feld [Index]);
    END FOR ALL;

    -- ...
END ...;
```

Exit–Anweisung

Die Exit–Anweisung dient zum vorzeitigen Beenden von Blöcken oder Prozeduren. Es gibt zwei Arten dieser Anweisung, mit oder ohne Bezeichner. Ohne Bezeichner ist sie nur in PS–Order–Generatoren erlaubt. Mit Bezeichner darf sie nur in BS–Order–Generatoren auftreten. Der Bezeichner gibt dann den Generatornamen des zu beendenden Blocks an.

Regel 61 *<exit-statement> ::=*
EXIT <*opt-identifier*>

```
TYPE FeldIndex IS 1 .. N;
TYPE FeldType  IS ARRAY [FeldIndex] OF integer;

Feld : FeldTyp;

PROCEDURE TYPE Suche (Element : IN integer; Resultat : OUT boolean) IS
BEGIN
    Resultat := False;

    Schleife : FOR Index IN FeldIndex LOOP
        IF Feld [Index] = Element THEN
            Resultat := True;
            EXIT Schleife;
        END IF;
    END LOOP;

END Suche;
```

zusammengesetzte und einfache Anweisungen

Die Menge der Anweisungsarten zerfällt gemäß Regel 43 in drei Klassen:

- DA–Komponenten–orientierte Anweisungen (Regel 47 bis 61),
- zusammengesetzte Anweisungen (Regel 62 bis 73) und
- einfache Anweisungen (Regel 74 bis 81).

Die im folgenden angegebenen zusammengesetzten und einfachen Anweisungen folgen im Aufbau und ihrer Wirkung den Konzepten imperativer Programmiersprachen und werden hier nicht näher erläutert.

Loop–Anweisung

Regel 62 *<loop-statement>* ::=
LOOP *<statement-part>* **END LOOP**

While–Anweisung

Regel 63 *<while-statement>* ::=
WHILE *<condition>* **LOOP** *<statement-part>* **END LOOP**

If–Anweisung

Regel 64 *<if-statement> ::=*
  IF *<condition>* **THEN** *<statement-part>*
    <elsif-part>
    <else-part>
  END IF

Regel 65 *<elsif-part> ::=*
  <empty>
  | *<elsif-part>* *<elsif>*

Regel 66 *<elsif> ::=*
  ELSIF
    <condition>
  THEN
    <statement-part>

Regel 67 *<else-part> ::=*
  <empty>
  | **ELSE** *<statement-part>*

```
FUNCTION TYPE Vorzeichen (Wert : IN integer) RETURN character IS
BEGIN
    IF Wert > 0 THEN
        RETURN '+';
    ELSIF Wert < 0 THEN
        RETURN '-';
    ELSE
        RETURN ' ';
    END IF;
END Vorzeichen;
```

Case–Anweisung

Regel 68 *<case-statement> ::=*
  CASE *<expression>* **IS**
    <case-alternatives>
  END CASE

Regel 69 *<case-alternatives> ::=*
<case-alternative>
| *<case-alternatives>* *<case-alternative>*

Regel 70 *<case-alternative> ::=*
WHEN
<choices-or-others>
DO
<statement-part>

Regel 71 *<choices-or-others> ::=*
<choices>
| **OTHERS**[4]

Regel 72 *<choices> ::=*
<choice>
| *<choices>* **'|'** *<choice>*

Regel 73 *<choice> ::=*
<sign-part> *<literal>*
| *<sign-part>* *<literal>* **'..'** *<sign-part>* *<literal>*

```
PROCEDURE TYPE VerarbeiteZeichen (Zeichen : IN character) IS
BEGIN
    CASE Zeichen IS
        WHEN 'a' .. 'z'       DO VerarbeiteKlein  (Zeichen);
        WHEN 'A' .. 'Z' | '_' DO VerarbeiteGross  (Zeichen);
        WHEN '0' .. '9'       DO VerarbeiteZiffer (Zeichen);
        WHEN OTHERS           DO VerarbeiteRest   (Zeichen);
    END CASE;
END VerarbeiteZeichen;
```

.... **Zuweisung**

Regel 74 *<assignment> ::=*
<name> **':='** *<expression>*

[4]In jeder Case–Anweisung darf höchstens ein **OTHERS**–Zweig angegeben werden, der dann an letzter Stelle stehen muß.

Return–Anweisung

Regel 75 *<return-statement> ::=*
RETURN *<expression>*[5]

```
FUNCTION TYPE Fakultaet (N : IN integer) RETURN integer IS
BEGIN
    IF N <= 0 THEN
        RETURN 1;
    ELSE
        RETURN N * Fakultaet (N - 1);
    END IF;
END Fakultaet;
```

Eingabeanweisung

Regel 76 *<input-statement> ::=*
INPUT
<name>
<input-from-part>

Regel 77 *<input-from-part> ::=*
<empty>
| **FROM** *<identifier>*

Ausgabeanweisung

Regel 78 *<output-statement> ::=*
OUTPUT
<expression>
<output-to-part>

Regel 79 *<output-to-part> ::=*
<empty>
| **TO** *<identifier>*

[5] Return–Anweisungen sind nur im Anweisungsteil von FS–Order–Generatoren zulässig.

Unvollständigkeitsanweisung

Regel 80 <*incomplete-statement*> *::=*
INCOMPLETE[6]

leere Anweisung

Regel 81 <*empty-statement*> *::=*
NULL

Ausdrücke

Der Aufbau von Ausdrücken in INSEL ist weitestgehend analog zu dem Aufbau von Ausdrücken in vergleichbaren imperativen Programmiersprachen. In INSEL gibt es sechs Klassen von Operatoren. Diese werden in der folgenden Tabelle nach aufsteigender Vorrangrelation aufgelistet.

Klasse	Operatoren-Bezeichner
binäre logische Operatoren	AND OR XOR
relationale Operatoren	'=' '/=' '<' '>' '<=' '>='
Additions-Operatoren	'+' '-' '&'
Multiplikations-Operatoren	'*' '/' MOD
Exponentiations-Operator	'**'
unäre Operatoren	'+' '-' NOT

Die für die einzelnen Operatoren erlaubten Operanden–Generatoren und die Ergebniswerte und –Generatoren sind den folgenden Tabellen zu entnehmen.

Binäre logische Operatoren				
Operator	**Generator**			**Berechnung**
	Operand 1 (Op_1)	**Operand 2** (Op_2)	**Ergebnis**	
AND	*Boolean*	*Boolean*	*Boolean*	$Op_1 \wedge Op_2$
OR	*Boolean*	*Boolean*	*Boolean*	$Op_1 \vee Op_2$
XOR	*Boolean*	*Boolean*	*Boolean*	$Op_1 \oplus Op_2$

[6] Mit der Unvollständigkeitsanweisung lassen sich noch nicht vollständig spezifizierte Anweisungsteile markieren.

Relationale Operatoren				
Operator	**Generator**			**Berechnung**
	Operand 1 (Op_1)	**Operand 2 (Op_2)**	**Ergebnis**	
'='	*Boolean*	*Boolean*	*Boolean*	$Op_1 = Op_2$
	Character	*Character*	*Boolean*	$Op_1 = Op_2$
	Integer	*Integer*	*Boolean*	$Op_1 = Op_2$
	Real	*Real*	*Boolean*	$Op_1 = Op_2$
	String	*String*	*Boolean*	$Op_1 = Op_2$
	Zeiger	*Zeiger*	*Boolean*	$Op_1 = Op_2$
	Zeiger	**NULL**	*Boolean*	$Op_1 =$ **NULL**
	NULL	*Zeiger*	*Boolean*	$Op_2 =$ **NULL**
	NULL	**NULL**	*Boolean*	**TRUE**
'/='	*Boolean*	*Boolean*	*Boolean*	$Op_1 \neq Op_2$
	Character	*Character*	*Boolean*	$Op_1 \neq Op_2$
	Integer	*Integer*	*Boolean*	$Op_1 \neq Op_2$
	Real	*Real*	*Boolean*	$Op_1 \neq Op_2$
	String	*String*	*Boolean*	$Op_1 \neq Op_2$
	Zeiger	*Zeiger*	*Boolean*	$Op_1 \neq Op_2$
	Zeiger	**NULL**	*Boolean*	$Op_1 \neq$ **NULL**
	NULL	*Zeiger*	*Boolean*	$Op_2 \neq$ **NULL**
	NULL	**NULL**	*Boolean*	**FALSE**
'<'	*Character*	*Character*	*Boolean*	$Op_1 < Op_2$
	Integer	*Integer*	*Boolean*	$Op_1 < Op_2$
	Real	*Real*	*Boolean*	$Op_1 < Op_2$
	String	*String*	*Boolean*	$Op_1 < Op_2$
'>'	*Character*	*Character*	*Boolean*	$Op_1 > Op_2$
	Integer	*Integer*	*Boolean*	$Op_1 > Op_2$
	Real	*Real*	*Boolean*	$Op_1 > Op_2$
	String	*String*	*Boolean*	$Op_1 > Op_2$
'<='	*Character*	*Character*	*Boolean*	$Op_1 \leq Op_2$
	Integer	*Integer*	*Boolean*	$Op_1 \leq Op_2$
	Real	*Real*	*Boolean*	$Op_1 \leq Op_2$
	String	*String*	*Boolean*	$Op_1 \leq Op_2$
'>='	*Character*	*Character*	*Boolean*	$Op_1 \geq Op_2$
	Integer	*Integer*	*Boolean*	$Op_1 \geq Op_2$
	Real	*Real*	*Boolean*	$Op_1 \geq Op_2$
	String	*String*	*Boolean*	$Op_1 \geq Op_2$

Additions–Operatoren				
Operator	**Generator**			**Berechnung**
	Operand 1 (Op_1)	**Operand 2 (Op_2)**	**Ergebnis**	
'+'	*Integer*	*Integer*	*Integer*	$Op_1 + Op_2$
	Real	*Real*	*Real*	$Op_1 + Op_2$
'-'	*Integer*	*Integer*	*Integer*	$Op_1 - Op_2$
	Real	*Real*	*Real*	$Op_1 - Op_2$
'&'	*String*	*String*	*String*	$Op_1 \circ Op_2$

Multiplikations–Operatoren				
Operator	**Generator**			**Berechnung**
	Operand 1 (Op_1)	**Operand 2 (Op_2)**	**Ergebnis**	
’*’	*Integer*	*Integer*	*Integer*	$Op_1 * Op_2$
	Real	*Real*	*Real*	$Op_1 * Op_2$
’/’	*Integer*	*Integer*	*Integer*	Op_1/Op_2
	Real	*Real*	*Real*	Op_1/Op_2
MOD	*Integer*	*Integer*	*Integer*	$Op_1 \bmod Op_2$
	Real	*Real*	*Real*	$Op_1 \bmod Op_2$

Exponentiations–Operator				
Operator	**Generator**			**Berechnung**
	Operand 1 (Op_1)	**Operand 2 (Op_2)**	**Ergebnis**	
’**’	*Integer*	*Integer*	*Integer*	${Op_1}^{Op_2}$
	Real	*Real*	*Real*	${Op_1}^{Op_2}$

Unäre Operatoren			
Operator	**Generator**		**Berechnung**
	Operand (Op)	**Ergebnis**	
’+’	*Integer*	*Integer*	Op
	Real	*Real*	Op
’–’	*Integer*	*Integer*	$-Op$
	Real	*Real*	$-Op$
NOT	*Boolean*	*Boolean*	$\neg Op$

Regel 82 *<condition> ::=*
<expression>

Regel 83 *<expression> ::=*
<relation>
| *<expression> <logical-operator> <relation>*

Regel 84 *<relation> ::=*
<simple-expression>
| *<simple-expression> <relational-operator> <simple-expression>*

Regel 85 *<simple-expression> ::=*
<term>
| *<simple-expression> <adding-operator> <term>*

Regel 86 <*term*> ::=
<*factor*>
| <*term*> <*multiplying-operator*> <*factor*>

Regel 87 <*factor*> ::=
<*operand*>
| <*factor*> <*exponentiating-operator*> <*operand*>

Regel 88 <*operand*> ::=
<*primary*>
| <*unary-operator*> <*primary*>

Regel 89 <*primary*> ::=
<*literal*>
| <*name*>
| <*function-call*>
| <*generating-expression*>
| <*type-conversion*>
| '(' <*expression*> ')'

Regel 90 <*literal*> ::=
<*character-literal*>
| <*string-literal*>
| <*integer-literal*>
| <*real-literal*>
| <*boolean-literal*>
| **NULL**

.... **Komponentennamen**

Regel 91 <*name*> ::=
<*identifier*>
| <*name*> '.' <*identifier*>
| <*name*> '.' **ALL**
| <*name*> '[' <*expression-list*> ']'

.... **FS–Order–Aufruf**

Eine FS–Order wird erzeugt, indem ein FS–Order–Aufruf ausgeführt wird. Hierfür sind der Name des entsprechenden FS–Order–Generators und die aktuellen Parameter anzugeben. Als Ergebnis der Ausführung einer FS–Order wird ein Resultatwert zurückgeliefert.

Parameterlose FS–Order–Generatoren lassen sich unter Verwendung von Regel 89, Alternative <*name*> aufrufen.

Regel 92 <*function-call*> *::=*
<*name*> '(' <*expression-list*> ')'

Generierungsausdruck

Ein Generierungsausdruck dient zur Erzeugung anonymer Komponenten. Nach dem Schlüsselwort **NEW** sind zuerst der Zeigergenerator[7] und dann der Komponentengenerator[8] zu nennen. Der Zeigergenerator muß mit dem Komponentengenerator qualifiziert sein. Bei der Erzeugung anonymer DA–Komponenten[9] sind dem DA–Generator entsprechende aktuelle Parameter anzugeben.

Regel 93 <*generating-expression*> *::=*
NEW '(' <*name*> ',' <*name*> ')' <*actual-parameter-part*>

```
TASK TYPE SPEC BriefkastenTyp (Kapazitaet : IN integer) IS
    PROCEDURE TYPE SPEC Einwerfen (Brief : IN  BriefTyp);
    PROCEDURE TYPE SPEC Leeren;
END BriefkastenTyp;

TASK TYPE BriefkastenTyp IS
    -- ...
END BriefkastenTyp;

POINTER TYPE BriefkastenZeigerTyp IS ACCESS BriefkastenTyp;

BriefkastenZeiger : BriefkastenZeigerTyp :=
               NEW (BriefkastenZeigerTyp, BriefkastenTyp) (815);
```

Typumwandlung

Zwischen den vordefinierten Generatoren (und entsprechenden Bereichsgeneratoren) sind eine Reihe von Typumwandlungen erlaubt. Die zulässigen Kombinationen sind in der folgenden Tabelle zusammengefaßt.

[7] Der Zeigergenerator ist maßgeblich für die Lebenszeiteinordnung der neuen Komponente.

[8] Die Nennung des Komponentengenerators ist insofern redundant, als daß er bereits über die Qualifikation des Zeigergenerators festgelegt ist. Sie erfolgt hier nur zur Klarstellung, bzgl. welches Generators eine Komponente erzeugt werden soll.

[9] Hier sind nur anonyme K–Akteure oder anonyme Depots möglich.

Konversion	**nach**				
von	*Boolean*	*Character*	*Integer*	*Real*	*String*
Boolean	–	–	–	–	–
Character	–	$\surd$	$\surd$	–	–
Integer	–	$\surd$	$\surd$	$\surd$	–
Real	–	–	$\surd$	$\surd$	–
String	–	–	–	–	–

Regel 94 *<type-conversion> ::=*
<name> '(' *<expression>* ')'[10]

```
-- Umwandlung von Gross- in Kleinbuchstaben

FUNCTION TYPE GiK (Zeichen : IN character) RETURN character IS
    Abstand : CONSTANT integer := integer ('a') - integer ('A');
BEGIN
    CASE Zeichen IS
        WHEN 'A' .. 'Z' DO
            RETURN character (integer (Zeichen) + Abstand);
        WHEN OTHERS DO
            RETURN Zeichen;
    END CASE;
END GiK;
```

.... **Operatoren**

Regel 95 *<logical-operator> ::=*
AND
| **OR**
| **XOR**

[10] Diese Regel ist ein Spezialfall der Regel *<function-call>* und wird hier nur aus Gründen der Übersichtlichkeit gesondert aufgeführt.

Regel 96 <*relational-operator*> *::=*
'`=`'
| '`<`'
| '`>`'
| '`/=`'
| '`<=`'
| '`>=`'

Regel 97 <*adding-operator*> *::=*
'`+`'
| '`-`'
| '`&`'

Regel 98 <*multiplying-operator*> *::=*
'`*`'
| '`/`'
| **MOD**

Regel 99 <*exponentiating-operator*> *::=*
'`**`'

Regel 100 <*unary-operator*> *::=*
'`+`'
| '`-`'
| **NOT**

.... **Literale**

Regel 101 <*character-literal*> *::=*
''' <*any-for-character*> '''

Regel 102 <*string-literal*> *::=*
'`"`' <*string*> '`"`'

Regel 103 <*string*> *::=*
<*empty*>
| <*string*> <*any-for-string*>

Regel 104 *<integer-literal> ::=*
<digits>

Regel 105 *<real-literal> ::=*
<digits> `'.'` *<digits> <exponent-part>*

Regel 106 *<exponent-part> ::=*
<empty>
| `'E'` *<sign-part> <digits>*

Regel 107 *<boolean-literal> ::=*
TRUE
| **FALSE**

.... **Allgemeines und Bezeichner**

Regel 108 *<empty> ::=*
ε

Regel 109 *<identifier> ::=*
<letter>
| *<identifier> <x-letter>*

Regel 110 *<letter> ::=*
`'A'` | ... | `'Z'` | `'a'` | ... | `'z'`

Regel 111 *<x-letter> ::=*
<letter>
| *<digit>*
| `'_'`

Regel 112 *<digit> ::=*
`'0'` | ... | `'9'`

Regel 113 *<digits> ::=*
<digit>
| *<digits> <digit>*

Regel 114 *<any> ::=*

<x-letter> | `'''` | `'"'` | `'!'` | `'@'` | `'#'` | ...

Regel 115 *<any-for-character> ::=*

<x-letter> | `'"'` | `'!'` | `'@'` | `'#'` | ...

Regel 116 *<any-for-string> ::=*

<x-letter> | `'''` | `'!'` | `'@'` | `'#'` | ...

Regel 117 *<sign-part> ::=*

<empty>
| `'+'`
| `'-'`

Kapitel 4

Liste der INSEL-Schlüsselworte

ACCEPT
ACCESS
ALL
AND
ARRAY
BEGIN
BLOCK
CASE
CONSTANT
DEPOT
DO
ELSE
ELSIF
END
ENTRY
EXIT
EXPORT
EXTENDED
FALSE
FOR
FORK
FROM
FUNCTION
GENERIC
IF
IMPORT
IN
INCOMPLETE
INPUT
IS
LOOP
MOD
NEW
NONE
NOT
NULL
OF
OR
OTHERS
OUT
OUTPUT
POINTER
PROCEDURE
PROCESS
RECORD
RETURN
SELECT
SPEC
TASK
TERMINATE
THEN
TO
TRUE
TYPE
WHEN
WHILE
WITH
XOR

Literaturverzeichnis

[Bau91] Uwe Baumgarten. *OlDiLa*–Syntax. Interner Bericht SA/91/2, Universität Oldenburg, Abteilung Systemarchitektur, Oldenburg, März 1991.

[Spi88] Peter Paul Spies. Sprachkonzepte für die Konstruktion Verteilter Systeme. Interner Bericht SA/88/2, Universität Oldenburg, Abteilung Systemarchitektur, Oldenburg, August 1988.

[Spi94] Peter Paul Spies et al. Sprachkonzepte für die Konstruktion Verteilter Systeme. Technischer Bericht, Technische Universität München, München, November 1994.

Index

1

1

1

1

1

1

1

1

1

1