Lars H. Hahn

CoLa - Codesign Language Referenz-Handbuch

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich „Technische Grundlagen der Informatik“

Stand: 19.12.2000

1 Einleitung

CoLa (**Co**design **La**nguage) ist eine Beschreibungssprache zur Spezifikation von Systemen mit Software- und Hardwareanteilen (HW-/SW-Systeme). *CoLa* enthält deshalb sowohl Anteile von klassischen Programmiersprachen als auch von Hardwarebeschreibungssprachen. Dennoch ist *CoLa* mehr als die Summe unterschiedlicher Sprachkonstrukte aus beiden Bereichen. Vielmehr war es Ziel bei der Konzeption von *CoLa*, die Umsetzung aller Konstrukte sowohl in Hardware als auch in Software zu ermöglichen. Die Auswahl eines Sprachmittels durch den Entwerfer legt noch nicht die spätere Partitionierung in Hardware und Software fest. Diese Entscheidung wird erst durch einen Compiler aufgrund des Kontextes und der Randbedingungen getroffen.

CoLa eignet sich besonders zur Spezifikation von eingebetteten Systemen (*embedded systems*) auf der Systemebene. Gerade hier geht es darum, Systeme „aus einem Guß“ zu entwerfen. Ziel ist es, zunächst das Verhalten des gesamten Systems zu spezifizieren, um dann möglichst automatisch eine konkrete Realisierung in Hardware und Software zu erhalten.

Dieser Bericht enthält die Definition der Syntax und der Semantik der Sprache *CoLa*. Für weitere Informationen der zugrunde liegenden Entwurfsmethodik wird auf die Dissertation des Autors verwiesen. Dieser Bericht erläutert somit nicht die Intentionen, die zur Aufnahme des einen oder anderen Konstruktes in den Sprachumfang von *CoLa* führten. Es findet an dieser Stelle auch kein Vergleich mit anderen Sprachen statt, sondern es wird ausschließlich die Bedeutung der Sprachkonstrukte erläutert. Auch wird nicht erklärt, wie eine Umsetzung in Hardware oder Software erfolgt oder wie eine mögliche Partitionierung aussehen könnte. Dieser Bericht beantwortet also nicht die Frage, **warum** sich *CoLa* besonders gut für den Entwurf von Codesign-Systemen eignet, sondern er gibt eine Anwort darauf, **wie** man eine korrekte Beschreibungen in *CoLa* erstellt.

1.1 Systembegriff

CoLa ist ein Sprache zur Spezifikation von HW/SW-Systemen. Es ist deshalb sinnvoll zunächst zu definieren, was ein HW/SW-System genau ist. In [SCHNEIDER] findet sich folgende Definition des Begriffs „System“:

> *„Ein System ist eine abgegrenzte Anordnung von aufeinander einwirkenden Gebilden. Solche Gebilde können sowohl Gegenstände als auch Denkmethoden und deren Ergebnisse sein. Diese Anordnung wird durch eine Hüllfläche von ihrer Umgebung abgegrenzt oder abgegrenzt gedacht. Durch die Hüllfläche werden Verbindungen des Systems mit seiner Umgebung geschnitten. Die mit diesen Verbindungen übertragenen Eigenschaften und Zustände sind die Größen, deren Beziehungen untereinander das dem System eigentümliche Verhalten beschreiben.“*

Diese Definition ist für ein HW/SW-System schon erstaunlich gut geeignet. Ersetzt man „Gegenstände“ durch „Hardware“ und „Denkmethoden“ durch „Software“ und formuliert man etwas knapper und moderner, so erhält man die folgende Definition für ein HW/SW-System:

> *„Ein HW-/SW-System ist eine abgegrenzte Anordnung von miteinander interagierenen Einheiten. Einheiten können sowohl Hardware als auch Software sein. Das System hat eine feste Grenze zu seiner Umgebung. Es gibt Verbindungen zwischen System und Umgebung. Die über die Verbindungen übertragenen Werte beschreiben das Verhalten des Systems.“*

Teilweise wird ein HW/SW-System auch als „heterogenes System“ bezeichnet. Dies ist allerdings mißverständlich, da nicht deutlich wird, welches die heterogene Teile des Systems sind. Es könnte z.B. auch ein System gemeint sein, das sowohl aus analogen als auch aus digitalen Komponenten besteht.

1.2 Entwurfsprozeß

Zu Beginn eines jeden Entwurfsprozesses steht die Beschreibung des Systems auf der höchsten Abstraktionsebene. Am Ende des Entwurfsprozesses steht das fertige System auf der untersten Abstraktionsebene. Im Laufe des Entwurfsprozesses wird das System von einer Abstraktionsebene auf die nächst Tiefere transformiert. Auf jeder Abstraktionsebene gibt es eine Beschreibung des Systems und es kann die korrekte Funktionsweise des Systems überprüft werden.

Der vorstehende Absatz beschreibt umgangssprachlich den Entwurfsprozeß eines Systems. Im folgenden wird die umgangssprachliche Formulierung schrittweise präzisiert. Es werden zunächst Systeme im allgemeinen betrachtet. In den folgenden Abschnitten wird der Entwurf von Soft- und Hardware im speziellen dargestellt.

In [MARWEDEL] findet sich folgende Definition des Begriffs „Synthese“:

> *„Synthese ist das Zusammensetzen von Komponenten oder Teilen einer niedrigeren Ebene zu einem Ganzen mit dem Ziel, ein auf einer höheren Ebene beschriebenes Verhalten zu erzielen.“*
>
> Definition 1.1

Häufig wird in der Literatur zwischen „Synthese“ und „Konstruktion“ unterschieden. Wobei „Konstruktion“ der allgemeinere Begriff ist. Mit „Synthese“ werden dann ausschließlich automatische Verfahren bezeichnet.

Bisher wurde im allgemeinen von einer Beschreibung des Systems gesprochen. Es ist aber sinnvoll zwischen Beschreibungen auf unterschiedlichen Abstraktionsebenen zu unterscheiden. In diesem Zusammenhang läßt sich Synthese auch knapper als Übergang zwischen unterschiedlichen Beschreibungen auffassen:

> *„Synthese ist die Überführung einer Spezifikation in eine Implementierung. Wobei Spezifikation die Beschreibung eines Systems auf der höheren Abstraktionsebene und Implementation die Beschreibung auf der niedrigeren Ebene ist.“*
>
> Definition 1.2

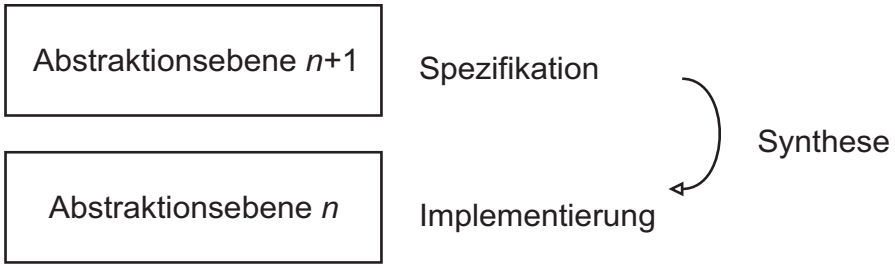


Abbildung 1.1

1.3 Ein einfaches Beispiel

Ein einfaches Programm könnte in *CoLa* wie folgt formuliert werden:

```
(num(32) out) function simple (num(32) in);

function simple (
  out := in * 3;
);
```

Im obigen Beispiel wird die Funktion **`simple`** zunächst deklariert, d.h. die Ein- und Ausgangsparameter werden festgelegt. In diesem Beispiel gibt es nur einen Eingangsparameter **`in`** und einen Ausgangsparameter **`out`**. Beide haben den Typ „**`num(32)`**“. Dieser wird intern als positive 32-Bit Ganzzahl repräsentiert. Es folgt die Definition der Funktion **`simple`**. Der Eingangsparameter **`in`** wird mit der Konstanten '**`3`**' multipliziert und dem Ausgangsparameter **`out`** zugewiesen. Wobei die '**`3`**' hier für eine abstrakte Zahl steht. Es ist Aufgabe eines Compilers, eine passende Repräsentation zu wählen.

Jedes *CoLa*-Programm muß mindestens eine Funktion enthalten. Die Funktion, deren Name mit dem Dateinamen identisch ist, ist der Startpunkt des Kontrollflusses ähnlich dem 'main' in 'C'. Diese Funktion wird Toplevel-Funktion genannt. Im Gegensatz zu 'C' wird die Toplevel-Funktion nicht nur einmal gestartet und abgearbeitet, sondern wiederholt angestoßen. Die Toplevel-Funktion wird also quasi in einer Endlosschleife immer wieder aufgerufen.

1.4 Lexikalische Struktur

Die lexikalische Struktur einer Sprache ist die Struktur ihrer Wörter, die Token genannt werden. Im folgenden werden vier Arten von Token unterschieden:

- ★ Reservierte Wörter, z.B. **`function`**,
- ★ Literale, z.B. **`13`**,
- ★ Sonderzeichen, z.B. **`>=`**,
- ★ Bezeichner, z.B. **`in7`**.

Bezeichner in *CoLa* bestehen aus Buchstaben, Zahlen und dem „Unterstrich“ ('_'). Am Anfang eines Bezeichners muß immer ein Buchstabe stehen. Umlaute und weitere Sonderzeichen sind nicht erlaubt. Reservierte Wörter heißen so, weil ein Bezeichner nicht aus

derselben Zeichenfolge bestehen kann wie ein reserviertes Wort. Zwischen Groß- und Kleinschreibung wird weder bei Bezeichnern noch bei reservierten Wörtern unterschieden.

In *CoLa* können Kommentare wie in C++ ('**//**') oder wie in VHDL ('**--**') eingefügt werden. Ein Kommentar endet mit dem Zeilenende.

```
// Dies ist ein Kommentar
-- dies auch
```

Blöcke werden in *CoLa* in „normale“ Klammern '**(**' und '**)**' eingeschlossen.

2 Datentypen

Ein Datentyp (kurz Typ) ist eine Menge von Werten. Variablen werden mit Hilfe der Datentypen deklariert, d.h. ihnen können im folgenden nur Werte aus der Menge, die dem Typ entspricht, zugewiesen werden.

CoLa kennt vordefinierte und benutzerdefinierte Datentypen. Alle vordefinierten Typen in *CoLa* (**num**, **snum**, **bool** und **char**) sind einfache Datentypen. Der Benutzer kann in *CoLa* eigene Datentypen definieren. Hierzu stehen unterschiedliche Typkonstruktoren (**type**, **vector** und **struct**) zur Verfügung. Da Typen als Mengen betrachtet werden können, sind Typkonstruktoren somit Mengenoperationen, die aus den vorhandenen Typen (Mengen) neue erstellen.

CoLa ist streng typisiert (*strongly typed*). Die gesamte Typüberprüfung (*type checking*) findet zur Übersetzungszeit statt. Während der Typüberprüfung stellt sich die Frage der Typäquivalenz. Die in *CoLa* verwendete Typäquivalenz ist die strukturelle Äquivalenz. Zwei Typen sind also gleich, wenn sie dieselbe Struktur besitzen. Dies bedeutet, daß sie mit dem gleichen Typkonstruktor aus den gleichen vordefinierten Typen aufgebaut wurden.

2.1 Vordefinierte Typen

CoLa enthält **num** als fundamentalen vordefinierten Typ. Er dient zur Beschreibung von positiven Festkommazahlen fester maximaler Größe. Bei jeder Deklaration einer Variablen vom Typ **num** muß die Anzahl der für die Variable zur Verfügung stehenden Bits explizit angegeben werden:

```
num(100) gamma;
num(20) delta1, delta2;
```

Für die Variable **gamma** werden 100 Bit reserviert. Dies entspricht einem Wertebereich von [0, 2^{100}-1] bzw. [0, 1267650600228229401496703205375]. Es können mehrere Variablen gleichen Typs hintereinander in einer Zeile deklariert werden. Die Variablen **delta1** und **delta2** sind beide vom Typ **num(20)**. Um vorzeichenbehaftete Zahlen verwenden zu können stellt *CoLa* den Typ **snum** (*signed number*) zur Verfügung. Für die folgende Variable **alpha** werden 8 Bit reserviert. Sie hat einen Wertebereich von [–128; 127].

```
snum(8) alpha;
```

Die interne Darstellung von vorzeichenbehafteten Zahlen erfolgt im Zweierkomplement. Es ist ein Zugriff auf einzelne Bits möglich:

```
num(8) alpha;
num(1) beta;
snum (3) gamma;
beta := alpha[7];
gamma := alpha[3..1];
```

Der Variablen **beta** wird das höchstwertigste Bit von **alpha** zugewiesen, wobei das höchstwertigste Bit immer ganz links steht. Das niederwertigste Bit steht somit immer rechts und ein Zugriff erfolgt über den Indexwert '0'. **gamma** werden drei Bits von **alpha** zugewiesen. Dabei wird keine Vorzeichenkonvertierung vorgenommen. Vielmehr ist es so das die Zugriffsoperatoren '**[**' und '**]**' immer den Typ **num** liefern. Hat **alpha** z.B. den Wert 66_{10} (1000010_2) , so wird **gamma** der Wert 1_{10} (001_2) zugewiesen.

Genaugenommen ist **num(8)** die Kurzschreibweise für **num(8,0)**, einer Variablen ohne Nachkommastellen. Beispiele für Variablendeklarationen von Festkommazahlen mit Nachkommastellen sind:

```
num(4,4) delta1;
snum(4,4) delta2;
```

Die Variable **delta1** wurde mit vier Bits vor dem Komma und mit vier Bits nach dem Komma deklariert. Die Variable **delta2** wurde ebenso deklariert, sie ist im Unterschied zu **delta1** vorzeichenbehaftet. Der Wertebereich von **delta1** ist $\left[0; 15\frac{15}{16}\right]$. Die kleinste von Null verschiedene Zahl ist $\frac{1}{16} = 0{,}0625$. Analog hierzu ist der Wertebereich von **delta2** $\left[-8; 7\frac{15}{16}\right]$.

Mit den Typen **bool** und **char** stellt *CoLa* zusätzlich zum Typ **num** zwei weitere vordefinierte Typen zur Verfügung. *CoLa* unterstützt explizit Boolsche Datentypen. Eine Variable vom Typ **bool** kann somit die Werte 'true' und 'false' annehmen.

```
bool wahroderfalsch1;
num(1) wahroderfalsch2;
```

Der Typ **bool** entspricht gemäß seiner internen Darstellung einem **num(1)**. Der Wert 'true' wird durch eine '1' repräsentiert, der Wert 'false' durch eine '0'. Die Typen der Variablen **wahroderfalsch1** und **wahroderfalsch2** sind in diesem Beispiel also austauschbar. Ein Boolscher Wert wird verwendet, um das Ergebnis einer logischen Operation darzustellen.

Einer Variablen vom Typ **char** kann ein ASCII-Zeichen zugewiesen werden:

```
char zeichen1;
num(8) zeichen2;
zeichen := 'a';
```

Der Typ **char** entspricht einem **num(8)**. Auch hier sind die Typen der Variablen **zeichen1** und **zeichen2** austauschbar. Der Wertebereich einer Variablen vom Typ **char** ist somit [0, 255].

2.2 Benutzerdefinierte Typen

CoLa erlaubt dem Benutzer, eigene Typen, sogenannte benutzerdefinierte Typen, zu definieren. Hierzu steht der Typkonstruktor **`type`** zur Verfügung. Zwei weitere Typkonstruktoren werden mit **`vector`** und **`struct`** in den folgenden Abschnitten beschrieben. Beispiele für benutzerdefinierte Typen sind:

```
type n32 num(32);
n32 alpha;
num(32) beta;
```

Die Variable **`alpha`** ist vom Typ **`num(32)`**. **`n32`** wurde lediglich als Alias für diesen Typ definiert. Somit sind die Variablen **`alpha`** und **`beta`** hier vom gleichen Typ. Sie wurden lediglich auf zwei unterschiedliche Arten definiert. Die Typen beider Variablen sind strukturell äquivalent.

2.3 Arraytyp

Ein Array ist eine Funktion *f*, die einen Indextyp *U* auf einen Komponententyp *V* abbildet:

$$f: U \rightarrow V$$

Der Indextyp muß in *CoLa* vom Typ **`num(?,0)`** sein, d.h. positiv und ganzzahlig sein. Der Komponententyp ist beliebig. Arrays sind sogenannte homogene Typen, d.h. alle Elemente (= Komponenten) sind vom gleichen Typ. Dies ermöglicht einen dynamischen Zugriff auf die Elemente, da alle Elemente einen gleich großen Platz im Speicher benötigen. Der Indexwert kann somit auch erst zur Laufzeit berechnet werden. Der in *CoLa* enthaltene Typkonstruktor für einen benutzerdefinierten Arraytyp ist **`vector`**:

```
type v10 vector(10) num(16);
v10 alpha;
vector(10) num(16) beta;
```

Es wird zunächst der Typ **`v10`** definiert. Mit dessen Hilfe wird ein Array **`alpha`** mit zehn Elementen deklariert. Jedes Element des Arrays ist vom Typ **`num(16)`**. Die typäquivalente Variable **`beta`** wird direkt definiert, d.h. ihr Typ erhält keinen eigenen Namen.

Mit Hilfe der eckigen Klammern '**`[`**' und '**`]`**' kann auf einzelne Elemente eines Arrays zugegriffen werden.

```
num(10) beta;
beta := alpha[3];
```

Einzelne Variablen können zu einem Array konkateniert werden. Ebenso kann ein Array zerteilt werden.

```
vector(3) num(10) alpha;
vector(2) num(10) beta;
num(10) a, b, c;
alpha := (a, b, c);
```

```
beta := alpha[0..1];
```

Soll ein Array zerteilt werden, so müssen die Grenzen der Teilung bereits zur Übersetzungszeit bekannt sein. Der Zugriff auf ein einzelnes Element des Arrays kann über einen erst zur Laufzeit berechneten Wert erfolgen. In *CoLa* können auch mehrdimensionale Arrays definiert werden:

```
vector(6) vector(5) vector(4) num(16) gamma;
gamma[1][2][3] := 17;
gamma[1][2] := (6, 7, 8, 9);
```

Es wird zunächst ein dreidimensionales Feld deklariert. Es können sowohl einzelnen Werte als auch ganze Vektoren zugewiesen werden.

2.4 Kartesisches Produkt

Das Kartesische Produkt besteht aus allen geordneten Elementpaaren aus zwei gegebenen Mengen *U* und *V*:

$$U \times V = \{ (u , v) \mid u \in U \wedge v \in V \}$$

Kartesische Produkte lassen sich auch für mehr als zwei Mengen bilden. Der Typkonstruktor für das Kartesische Produkt in *CoLa* ist **`struct`**. Ein Typ, der mit Hilfe von **`struct`** definiert wurde, ist einem mit **`vector`** definierten Typ sehr ähnlich. Die einzelnen Elemente dürfen jedoch unterschiedliche Typen haben. Man bezeichnet einen mit **`struct`** erzeugten Typ deshalb auch als heterogenen Typ, im Gegensatz zu einem homogenen Typ (**`vector`**). Die einzelnen Elemente eines mit **`struct`** erzeugten Typs haben Bezeichner. Mit ihnen wird der Zugriff auf die einzelnen Elemente ermöglicht. Die Elemente benötigen in der Regel unterschiedlich große Speicherbereiche, daher ist nur ein statischer und kein dynamischer Zugriff auf die Elemente erlaubt. Statischer Zugriff bedeutet, daß das Element, auf das zugegriffen werden soll, bereits zur Übersetzungszeit bekannt sein muß, es also nicht erst zur Laufzeit berechnet werden kann.

Anders als bei **`vector`** können die Variablen nicht direkt deklariert werden, sondern der neue Typ erhält immer einen eigenen Namen. Dieser dient dann zur Deklaration der Variablen.

```
struct struktur1(bool alpha, char beta, num(20) gamma);
struktur1 delta;
```

Es wird zunächst ein Typ **`struktur1`** mit Hilfe von **`struct`** mit den Feldern **`alpha`**, **`beta`** und **`gamma`** definiert. Dann wird eine Variable **`delta`** mit diesem Typ deklariert.

2.5 Konstanten und Literale

Analog zu Variablen können in *CoLa* Konstanten definiert werden:

```
const bool wahr := true;
const num(2, 4) pi := 3.14;
```

Als Wert einer Konstanten in einer Konstantendefinition in *CoLa* dürfen nur Literale verwendet werden. Die Rundung von Konstanten und Literalen erfolgt nach dem Verfahren „round-to-nearest“.

Teilweise problematisch ist die Ermittlung des Typs eines Literals. Innerhalb einer Konstantendefinition stellt dies kein Problem dar, weil der Typ durch den Entwerfer eindeutig spezifiziert wurde. Unklar ist der gewünschte Typ eines Literals mit Nachkommaanteil innerhalb einer Berechnung, wie z.B.:

```
num(4) a;
num(4, 4) b;
a := 7;
b := 0.1 * a;
```

Der Typ eines ganzzahligen Literals läßt sich eindeutig bestimmen. Es werden soviel Bits verwendet wie minimal zur Darstellung des Wertes benötigt werden. In unserem Beispiel hat das Literal **`7`** somit den Typ **`num(3)`**. Dieser kann ohne Probleme implizit in **`num(4)`** gewandelt werden. Nicht eindeutig ist dies im Fall von Zahlen mit Nachkommaanteil, deshalb erhalten diese immer den Typ **`num(?, 16)`**. Der Vorkommaanteil wird entsprechend einer ganzen Zahl gehandhabt. Der Nachkommaanteil hingegen wird auf 16 Bit festgelegt. Falls dies nicht von Entwerfer gewünscht wird, so hat er die Möglichkeit sich passende Konstanten zu definieren. Zusammengefaßt haben in *CoLa* Literale die folgenden Typen:

Boolsche Wert (true, false)	**`bool`**
Zeichen (z.B. 'a')	**`char`**
Ganze Zahlen (z.B. 123)	**`num(?, 0) / snum(?, 0)`**
Festkommazahlen (z.B. 0.1)	**`num(?,16) / snum(?, 16)`**

Literale mit führendem Minuszeichen erhalten den Typ **`snum`**.

3 Zuweisungen und Arithmetik

3.1 Zuweisungen

Zuweisungen erfolgen in *CoLa* mit Hilfe von Doppelpunkt und Gleichheitszeichen '**:=**':

```
num(32) alpha, beta;
alpha := 7;
beta := alpha;
```

Zuweisungen an Variablen vom Typ **vector** oder **struct** können in gebundener Form (Konkatenation) erfolgen:

```
vector(4) num(32) gamma;
struct struktur1(bool a, char b, num(20) c);
struktur1 delta;
gamma := (1, 2, 3, 4);
delta := (true, 'A', 27);
```

Eine Konkatenation von Werten ist nicht nur auf der rechten Seite einer Zuweisung erlaubt, sondern auch auf der linken. Variablen können so zusammengefaßt werden und ihnen können gesammelt Werte zugewiesen werden:

```
num(32) alpha, beta;
vector(2) num(32) gamma;
(alpha, beta) := gamma;
```

In diesem Beispiel wird der Vektor **gamma** aufgespalten. Die erste Komponente wird **alpha**, die zweite wird **beta** zugewiesen. Man könnte alternativ auch schreiben:

```
alpha := gamma[0];
beta := gamma[1];
```

Auf Komponenten eines **struct** wird mit dem Punkt ('.') zugegriffen:

```
struct struktur1(bool a, char b, num(20) c);
struktur1 delta;
delta.a := true;
delta.b := 'A';
delta.c := 27;
```

Hier wird genauso wie in dem weiter oben stehenden Beispiel zunächst der neue Typ **struktur1** erzeugt. Die Variable **delta** wird mit diesem Typ deklariert. Die Zuweisung von Werten erfolgt in diesem Fall nur nicht gesammelt, sondern einzeln.

3.2 Implizite Konvertierungen

Soll einer Variablen ein Wert mit einer kleineren Wortbreite zugewiesen werden, so erfolgt eine implizite Konvertierung. Soll ein zu großer Wert zugewiesen werden, erzeugt der Compiler eine Fehlermeldung.

```
num(8) acht_bit_breit;
num(16) sechzehn_bit_breit;
sechzehn_bit_breit := acht_bit_breit; // OK
acht_bit_breit := sechzehn_bit_breit; // Fehler
```

Wird einer vorzeichenbehafteten Variablen ein vorzeichenloser Wert zugewiesen, wird eine implizite Konvertierung vorgenommen. Im umgekehrten Fall gibt der Compiler eine Fehlermeldung aus. Eine implizierte Konvertierung kann hier nicht erfolgen, da eine negative Zahl nicht von einer vorzeichenlosen Variablen repräsentiert werden kann.

```
snum(16) mit_vorzeichen;
num(15) ohne_vorzeichen;
mit_vorzeichen := ohne_vorzeichen; // OK
ohne_vorzeichen := mit_vorzeichen; // Fehler
```

Analog hierzu darf einer ganzzahligen Variablen kein Wert mit einem Nachkommaanteil zugewiesen werden. Im umgekehrten Fall erfolgt wiederum eine implizite Konvertierung der Ganzzahl in eine Festkommazahl.

```
num(8) ganze_zahl;
num(8,8) festkomma_zahl;
festkomma_zahl := ganze_zahl; // OK
ganze_zahl := festkomma_zahl; // Fehler
```

3.3 Explizite Konvertierungen

In *CoLa* ist auch möglich Konvertierungen explizit vorzunehmen. Hierzu gibt es einen eigenen Zuweisungsoperator „**:~**“. Mit seiner Hilfe lassen sich Variablen auch Werte mit größeren Bitbreiten zuweisen. Hierbei werden überzählige Bit einfach abgeschnitten. Der Entwerfer ist für die Vermeidung eines möglichen Wertverlust selbst verantwortlich. Es folgen hierzu die gleichen Beispiele wie im vorherigen Abschnitt.

```
num(8) ganze_zahl1, ganze_zahl2;
num(8,8) festkomma_zahl;
num(16) sechzehn_bit_breit;
ganze_zahl1 :~ festkomma_zahl; // OK
ganze_zahl2 :~ sechzehn_bit_breit; // OK
```

Im ersten Fall erhält **`ganze_zahl1`** nur die acht Vorkommabits von **`festkomma_zahl`** zugewiesen. Im zweiten Fall gehen die vorderen acht Bits von **`sechzehn_bit_breit`** verloren.

3.4 Grundrechenarten

CoLa kennt die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division:

```
num(32) alpha, beta;
alpha := beta + 2;
alpha := beta - 3;
alpha := beta * 4;
alpha := beta / 5;
```

Eine „Division durch Null“ erzeugt einen Laufzeitfehler. Auf Variablen, die aus mehreren Elementen bestehen (**`vector`** und **`struct`**), werden die Operationen für jedes Element einzeln durchgeführt. Dies ist im Falle der Multiplikation **nicht** das Skalarprodukt. Hierfür gibt es eigens eine vordefinierte Funktion in *CoLa*.

```
vector(3) num(32) gamma, delta;
gamma := (1, 2, 3);
delta := gamma * gamma;
```

Nach dieser Berechnung hat **`delta`** den Wert (1, 4, 9). Die eingebauten Grundrechenarten sind nicht nur für typäquivalente Variablen definiert. Die Multiplikation eines Vektors mit einem Skalarwert ist ebenso vordefiniert. Die einzelnen Komponenten werden mit dem Skalar multipliziert. Das Ergebnis ist wieder ein Vektor.

3.5 Vergleichsoperatoren

Von *CoLa* werden die gängigen Vergleichsoperatoren unterstützt. Dies sind im einzelnen:

`>`	größer
`<`	kleiner
`==`	gleich
`!=`	ungleich
`>=`	größer oder gleich
`<=`	kleiner oder gleich

Es können beliebige Werte vom Typ **`num`** und **`snum`** verglichen werden, sowohl ganze Zahlen als auch Festkommazahlen. Variablen vom Typ **`vector`** oder **`struct`** werden komponentenweise verglichen. Ebenso können einzelne Komponenten als Argumente für einen Vergleichsoperator dienen. In jedem Fall müssen die Argumente einer Vergleichsoperation typäquivalent sein. Das Ergebnis einer Vergleichsoperation ist vom Typ **`bool`**.

3.6 Logische Operatoren

Mit Hilfe der folgenden logischen Operatoren können Vergleiche zu logischen Ausdrücken kombiniert werden:

`and`	Konjunktion
`or`	Disjunktion
`not`	Negation

xor Exklusives Oder

Die Argumente eines logischen Operators müssen vom Typ **bool** sein. Das Ergebnis ist ebenfalls vom Typ **bool**.

```
num(16) a;
bool c, b;
c := a > 10;
b := c and (a < 100);
```

In diesem Beispiel erhält **c** den Wert **true**, wenn **a** größer als zehn ist. Die Variable **b** wird **true**, wenn der Wert von **a** zwischen elf und 99 liegt.

3.7 Schiebeoperationen

In *CoLa* gibt es zwei unterschiedliche Arten von Schiebeoperatoren. Es können sowohl einzelne Bits verschoben werden als auch Komponenten in einem Vektor. Die Komponenten eines Vektors werden mit dem Operator '**#>**' nach rechts und mit dem Operator '**<#**' nach links verschoben.

```
vector(10) num(16) alpha;
num(16) alt, neu;
alt := alpha <# neu;
```

In diesem Beispiel wird der Vektor **alpha** nach links verschoben. Hierbei ist es wichtig, daß in *CoLa* das Element mit dem kleinsten Index bzw. das niederwertigste Bit immer rechts steht. Somit werden hier alle Elemente einen Index höher geschoben. Das Element mit dem Index 0 wird frei und erhält den Wert der Variablen **neu**. Der Wert des Element mit dem Index 9 „fällt aus dem Vektor“ und wird der Variablen **alt** zugewiesen. Der Vektor verhält sich mit Hilfe des Schiebeoperators wie eine Pipeline. Eine Variable von Typ **num** verhält sich wie ein Vektor mit nur einem Element:

```
num(16) beta;
num(16) alt, neu;
alt := beta <# neu;
```

Hier erhält **beta** einen neuen Wert. Der alte Wert kann weiter verwendet werden. Alternativ kann man selbstverständlich schreiben:

```
alt := beta;
beta := neu;
```

Bitweises Verschieben wird mit den Operatoren '<<' (nach links) und '>>' (nach rechts) verwirklicht. Die Schiebeoperatoren können sowohl auf Variablen vom Typ **num** als auch vom Typ **vector** angewendet werden. Im ersten Fall werden die einzelnen Bits der jeweiligen Variablen geschoben, im zweiten Fall werden die Komponenten des Vektors parallel um jeweils ein Bit verschoben. Ein Schieberegister kann wie folgt beschrieben werden:

```
num(10) beta;
num(1) alt, neu;
alt := beta << neu;
```

Hier wird eine Variable vom Typ **num** mit 10 Bits deklariert. Mit Hilfe des Schiebeoperators werden alle Bits um ein Bit nach links geschoben. Das niederwertigste Bit erhält den Wert der Variablen **neu**. Die Variable **alt** erhält den Wert des herausfallenden höchstwertigen Bits. Im folgenden Beispiel erfolgt das Schieben der Bits parallel. Die einzelnen Bits des Vektors **neu** werden von rechts in die Komponenten des Vektors **gamma** geschoben.

```
vector(10) num(16) gamma;
vector(10) num(1) alt, neu;
alt := gamma << neu;
```

In den vorangegangenen Beispielen wurde das Schieben nach links demonstriert. Das Rechtsschieben erfolgt analog hierzu mit Hilfe der Operatoren '**>>**' und '**#>**'. Manchmal möchte man Elemente nicht schieben, sondern rotieren. Auch dies ist in *CoLa* möglich:

```
vector(10) num(16) gamma;
num(16) delta;
gamma <#;
delta <<;
```

Alle Elemente des Vektors **gamma** werden hier um ein Element nach links geschoben. Das Element mit dem Index 9 wird zum Element mit dem Index 0. Es wird **gamma** kein Element neu hinzugefügt, und es fällt auch keines heraus. Alle Elemente werden lediglich im Kreis geschoben. Ebenso werden die einzelnen Bits der Variablen **delta** rotiert.

4 Kontrollstrukturen

Berechnungen werden in *CoLa* zu Datenblöcken zusammengefaßt. Jeder Datenblock ist in Klammern '**(** ... **)**' eingeschlossen. Datenblöcke können alternativ oder auch wiederholt ausgeführt werden. Hierzu bietet *CoLa* unterschiedliche Kontrollstrukturen.

4.1 Gültigkeitsbereiche

Variablen dürfen nur am Anfang eines Blockes deklariert werden. Ihr Gültigkeitsbereich endet mit der schließenden Klammer '**)**' des jeweiligen Blockes. *CoLa* ist somit blockstrukturiert und besitzt lexikalische Gültigkeitsbereiche. Gültigkeitsbereiche von Variablen können Lücken haben:

```
...
num(16) a, b;
a := 7;
(
  num(16) a;
  a := 8;
)
b := a;
...
```

In diesem Beispiel wird der Variablen **b** der Wert **7** zugewiesen. Die zuerst deklarierte Variable ist innerhalb des inneren Block nicht sichtbar. Die lokale Variable **a** erhält den Wert **8**. Nach Verlassen des inneren Blocks wird die äußere Variable **a** wieder sichtbar und wird **b** zugewiesen.

4.2 Verzweigungen

Verzweigungen (bzw. alternative Ausführungen) werden in *CoLa* mit Hilfe der Befehle **if** und **else** realisiert, wobei der **else**-Teil optional ist.

```
if (alpha > 10) (
  beta := 2;
)
else (
  beta := 1;
)
```

Wenn **alpha** größer als zehn ist, erhält **beta** den Wert '2', sonst '1'. Die Semantik unterscheidet sich nicht von der Semantik gängiger Programmiersprachen.

4.3 Switch-Anweisung

Eine **switch**-Anweisung besteht aus einer Variablen und mehreren **case**-Blöcken:

```
num(8) alpha;
num(16) beta;
...
switch(alpha) (
  case 1 (beta := 27;)
  case 2 (beta := 9;)
  default (beta := 0;)
)
```

Dem Schlüsselwort **case** folgt jeweils ein konstanter Ausdruck. Ist dieser gleich dem Wert der Variablen der **switch**-Anweisung, so wird der jeweilige Block ausgeführt. Paßt keiner der konstanten Ausdrücke, wird der Block nach dem '**default**' ausgeführt. Anschließend wird das Programm nach der **switch**-Anweisung fortgesetzt. Eine Besonderheit ist, daß die Variable hinter dem **switch** einen beliebigen Typ haben darf. Sie kann z.B. auch von Typ **vector** sein.

4.4 Schleifen

CoLa kennt mit der **while**- und **loop**-Schleife zwei unterschiedliche Typen von Schleifen. Beide können auch zu einer Schleife kombiniert werden. Die einfachere von beiden ist die **loop**-Schleife:

```
erg := 1;
loop i (1..10) (
  erg := erg * i;
)
```

Die Schleifenvariable **i** wird implizit deklariert und wird mit der unteren Schleifengrenze initialisiert. Sie ist nur innerhalb der Schleife gültig. Ihr kann kein Wert direkt zugewiesen werden. In dem oben stehenden Beispiel wird der auf **loop** folgende Block zehnmal ausgeführt. Die Schleifenvariable **i** nimmt dabei Werte von eins bis zehn an. Sie wird somit in jedem Durchgang um Eins erhöht. Einer **loop**-Schleife kann eine optionale Abbruchbedingung hinzugefügt werden:

```
erg := 1;
loop i (1..10) while (erg <= ende) (
  erg := erg * i;
)
```

Diese Schleife wird maximal zehnmal ausgeführt. Wenn **erg** vorher größer als **ende** wird, wird die Schleife vorzeitig beendet. In *CoLa* kann auch eine „reine“ **while**-Schleife verwendet werden:

```
i := 1; erg := 1;
while (i <= ende) (
  erg := erg * i;
  i := i + 1;
)
```

Diese Schleife hat keine zur Übersetzungszeit bekannte maximale Anzahl von Iterationen. Sie wird solange ausgeführt, wie die Bedingung nach **`while`** erfüllt ist. *CoLa* kennt eine Compilervariable, die die maximale Anzahl von Schleifendurchläufen von reinen **`while`**-Schleifen festlegt. Wird diese Anzahl übertroffen, führt dies zu einem Laufzeitfehler. Die Maximalzahl kann vom Benutzer beliebig gesetzt werden. Der voreingestellte Wert ist 1000.

5 Funktionen und Prozesse

5.1 Funktionen

Jede Funktion muß zunächst deklariert werden. Diese Einschränkung wurde gewählt, um einen „single-pass“-Compiler für *CoLa* zu ermöglichen. Die Reihenfolge der Funktionen ist beliebig. Es muß lediglich die Deklaration vor der Definition und vor der ersten Verwendung stehen.

```
type i16 num(16);
(i16 aout)function anna(i16 ain1, i16 ain2);
(i16 bout)function berta(i16 bin1, i16 bin2);
(i16 cout)function caesar(i16 cin1, i16 cin2);
(i16 tout)function toplevel(i16 tin1, i16 tin2);
```

Bei der Deklaration werden einer Funktion ihre formalen Parameter zugeordnet. Es werden Ein- und Ausgangsparameter unterschieden. Eingangsparameter nehmen die Funktionsargumente auf. Sie dürfen innerhalb der Funktion nur auf der rechten Seite von Anweisungen stehen. Sie sind somit innerhalb der Funktion konstant. Die Übergabe der Eingangsparameter erfolgt „call-by-value“. Ausgangsparameter enthalten die Rückgabewerte einer Funktion. Sie dürfen nur auf der linken Seite einer Anweisung stehen. Werden die Rückgabewerte für weitere Berechnungen innerhalb der Funktion benötigt, müssen sie zunächst einer lokalen Variable zu gewiesen werden. Den Ausgangsparametern dürfen an unterschiedlichen Stellen der Funktion Werte zugewiesen werden. Dieses ist z.B. im Falle von bedingten Zuweisungen sinnvoll. Der Rückgabeparameter erhält seinen Wert erst beim Terminieren der Funktion („copy-out“). Diese Art der Parameterübergabe wird auch als Ergebnisübergabe („pass-by-result“) bezeichnet.

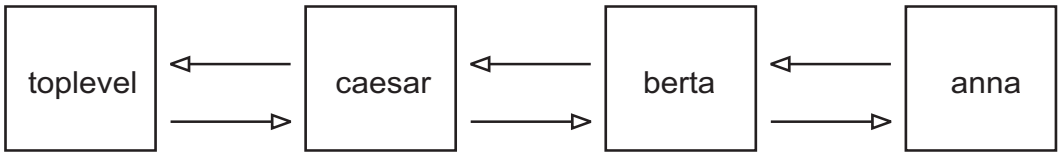
Nachdem die Funktionen deklariert sind, kann das jeweilige Verhalten definiert werden:

```
function anna (
  aout := ain1 * 2 + ain2;
);

function berta (
  bout := anna (bin1, bin2) * 2;
);

function caesar (
  cout := berta(cin1, cin2) * 2;
);

function toplevel (
  tout := caesar(tin1, tin2) * 2;
);
```



Die Funktionsaufrufe erfolgen in diesem Beispiel verschachtelt.

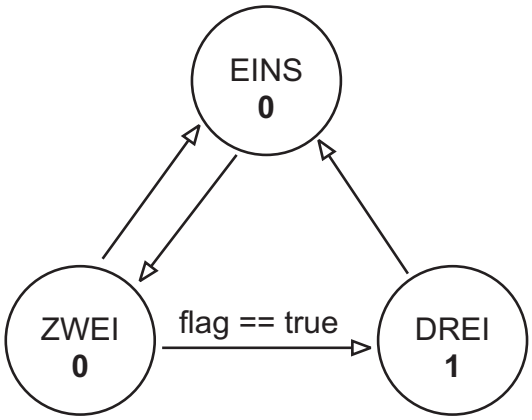
5.2 Automaten

Automaten werden von *CoLa* als spezielle Funktionen realisiert. Automaten werden analog zu Funktionen deklariert. Rechts vom Namen des Automaten stehen die Eingangsparameter, links die Ausgangsparameter (= Rückgabewerte).

```
(num(32) raus)automat alpha(bool flag);
```

Es können ausschließlich Moore-Automaten beschrieben werden. Ausgangsparameter dürfen somit nur von den Zuständen des Automaten und nicht direkt von den Eingangsparametern abhängen. Dieses hat zur Folge, daß den Ausgangsparametern nur zur Übersetzungszeit berechenbare Werte zugewiesen werden dürfen. Innerhalb eines Automat gibt es keine lokalen Variablen. Die logischen Ausdrücke zur Berechnung des Folgezustands bestehen aus logischen Verknüpfungen der Eingangsparameter.

Jeder Automat besteht aus mehren Zuständen. Nach jedem Zustandsnamen steht ein Doppelpunkt. Zu jedem Zustand gehören ein oder mehrere Zustandsübergänge. Diese bestehen aus einer Bedingung und einem Folgezustand. Die Zustandsübergänge werden bis zur ersten erfüllten Bedingung sequentiell abgearbeitet. In dem dazugehörigen Folgezustand wird die Bearbeitung fortgesetzt.




```
automat alpha (
  raus := 0;
  eins: -> zwei;
  zwei: (flag == true) -> drei:
        -> eins;
  drei: raus := 1;
        -> eins;
);
```

In diesem Beispiel wechselt der Automat im Normalfall zwischen den Zuständen '**eins**' und '**zwei**'. Nur wenn im Zustand '**zwei**' die boolsche Variable **flag** wahr ist, wird in den Zustand '**drei**' gewechselt. Einem Ausgangsparameter, in diesem Fall **raus**, kann ein Standardwert zugewiesen werden. Dieser gilt in allen Zuständen, in denen nicht explizit ein anderer Wert zugewiesen wird. Hier hat **raus** normalerweise den Wert '**0**'. Nur im Zustand '**drei**' erhält er den Wert '**1**'.

5.3 Rekursive Funktionen

Funktionsaufrufe können in *CoLa* rekursiv erfolgen. Die maximale Rekursionstiefe ist eine Compilervariable. Voreingestellt ist ein Wert von zehn rekursiven Aufrufen. Wird die maximale Rekursionstiefe zur Laufzeit überschritten, führt dieses zu einem Laufzeitfehler.

Ein Compiler muß erkennen, ob ein rekursiver Aufruf vorliegt. Er begrenzt in diesem Fall die Anzahl der rekursiv aufgerufenen Instanzen einer Funktion auf die maximale Rekursionstiefe. Falls die Rekursion durch das gegenseitige Aufrufen von zwei oder mehr Funktionen entsteht, werden von jeder der beteiligten Funktionen die maximale Anzahl von Instanzen (= maximale Rekursionstiefe) erzeugt.

5.4 Parallele Ausführungen

CoLa ermöglicht es, Anweisungen parallel auszuführen. Hierzu werden die Anweisungen nicht durch ein Semikolon '**;**', sondern durch einen Doppelpunkt '**:**' getrennt.

```
b := 3 * a : c := 2 * a;
```

In diesem Beispiel können die beiden Multiplikationen parallel ausgeführt werden. Es bleibt dem jeweiligem Compiler überlassen, ob er hieraus auch tatsächlich zwei parallel arbeitende Instruktionen generiert. Wenn die parallele Ausführung aufgrund von Datenabhängigkeiten unmöglich ist, muß der Compiler einen Fehler melden. Dieses wäre im folgenden Beispiel der Fall:

```
b := 3 * a : c := 2 * b;
```

Die Variable **b** erhält in der linken Zuweisung einen neuen Wert, während sie gleichzeitig in der rechten gelesen wird. Der Programmierer muß deshalb das gewünschte Verhalten deutlich machen. Entweder es soll der „alte“ Wert von **b** verwendet werden

```
b_alt := b;
b := 3 * a : c := 2 * b_alt;
```

oder der neue Wert von **b** dient zur weiteren Berechnung:

```
b := 3 * a;
c := 2 * b;
```

Dann ist aber, wie in dem vorstehenden Beispiel, eine parallele Berechnung nicht möglich. Die Multiplikationen müssen sequentiell ausgeführt werden. Ein ':' bindet stärker als ein ';', d.h. ein Semikolon ';' schließt einen Block von parallelen Anweisungen ab.

5.5 Prozesse

Funktionsaufrufe können auf die gleiche Weise wie andere Berechnungen parallel ausgeführt werden. Es müssen ebenso eventuelle Datenabhängigkeiten berücksichtigt werden. Auf diese Weise lassen sich in *CoLa* einfach Prozesse beschreiben.

```
function process1() (
...
)

function process2() (
...
)

function start (
  process1() : process2();
)
```

Die Funktionen **process1** und **process2** werden parallel ausgeführt. Echte Prozesse werden aus Funktionen nur, wenn ihre Aufrufe in der Toplevel-Funktion erfolgen. Nur so werden sie permanent angestoßen. Im obigen Beispiel muß **start** somit die Toplevel-Funktion sein.

5.6 Prozeßkommunikation

Der Austausch von Werten zwischen Prozessen ist nicht direkt möglich. Folgendes Beispiel verdeutlicht dieses:

```
function start (
  num(16) a, b;
  a := process1(b) : b := process2(a);
)
```

Der Compiler würde hier einen Fehler ausgeben. Aufgrund der Datenabhängigkeit ist eine parallele Ausführung der Zuweisung nicht gestattet. In *CoLa* ist ein Austausch von Daten während ein Prozeß läuft unmöglich. Um eine Kommunikation zu ermöglichen, muß jeder Prozeß eigene Variablen erhalten:

```
function start (
  num(16) a1, a2, b1, b2;
  a1 := process1(b1) : b2 := process2(a2);
  a2 := a1;
  b1 := b2;
)
```

Wenn beide Prozesse ihre Ergebnisse fertig berechnet haben, werden diese an die Eingangsparameter zugewiesen. Danach werden die Prozesse neu angestoßen. In *CoLa* lassen sich somit ausschließlich synchrone Prozesse beschreiben.

6 Vordefinierte Funktionen

In *CoLa* sind eine Reihe von Funktionen vordefiniert. Diese ermöglichen eine effiziente Umsetzung auf der jeweiligen Zielarchitektur. Bei den jeweiligen Funktionen sind die grundsätzlichen Typen der Ein- und Ausgangsparameter angegeben. Es ist die Aufgabe des Compilers, die genauen Bitbreiten der unterschiedlichen Parameter geeignet zu wählen. Die vordefinierten Funktionen sind also generische Funktionen. Die exakten Typen einer jeden Funktionsinstanz wählt der Compiler entsprechend des jeweiligen Kontextes.

Zum jetzigen Zeitpunkt sind nur einige rudimentär Funktionen vorgesehen. Der Sprachumfang von *CoLa* kann bei Bedarf leicht um weitere Funktionen erweitert werden.

6.1 abs

`num abs( snum )`

Die Funktion **`abs`** liefert den absoluten Wert einer vorzeichenbehafteten Zahl. Sie entfernt also das Vorzeichen einer Zahl.

6.2 cos

`snum cos( snum )`

Die Funktion **`cos`** berechnet den Kosinus des Arguments. Die Rundung des Ergebnisses erfolgt nach dem Verfahren „round-to-nearest“.

6.3 int

`num int( num )`

Die Funktion **`int`** liefert den ganzzahligen Anteil des Arguments.

6.4 sp

`snum sp( vector, vector )`

Die Funktion **`sp`** multipliziert die beiden Argumente komponentenweise und summiert die einzelnen Produkte auf (Skalarprodukt). Auf den Typen der Komponenten der Vektoren müssen Multiplikation und Addition definiert sein.

6.5 max

`num max( vector )`

Die Funktion **`max`** liefert den Index der Komponente des Arguments mit dem größten Wert aller Komponenten. Auf den Typen der Komponenten der Vektoren muß die Vergleichsoperation '**`>`**' definiert sein.

6.6 min

`num min( vector )`

Die Funktion **`min`** liefert den Index der Komponente des Arguments mit dem kleinsten Wert aller Komponenten. Auf den Typen der Komponenten der Vektoren muß die Vergleichsoperation '**`<`**' definiert sein.

6.7 sin

`snum sin( snum )`

Die Funktion **`sin`** berechnet den Sinus des Arguments. Die Rundung des Ergebnisses erfolgt nach dem Verfahren „round-to-nearest“.

6.8 sint

`snum sint( snum )`

Die Funktion **`sint`** liefert den ganzzahligen Anteil des Arguments.

6.9 sum

`snum sum( vector )`

Die Funktion **`sum`** liefert den aufsummierten Wert aller Komponenten des Arguments. Auf den Typen der Komponenten der Vektoren muß die Addition definiert sein.

7 Zeitbedingungen

Zur Beschreibung von Echtzeitanforderungen können in *CoLa* unterschiedliche Zeitbedingungen formuliert werden. Feste Berechnungszeiten werden in *CoLa* mit dem Schlüsselwort **`rate`** spezifiziert. Sie können absolut oder relativ angegeben werden. Mit **`step`** können zeitabhängige Ausführungszeitpunkte definiert werden.

7.1 Absolute Zeitbedingungen

Mit Hilfe einer absoluten Zeitbedingung wird einer Zuweisung eine feste Zeit, die ihr maximal zur Berechnung zur Verfügung steht, zugeordnet. Nach Ablauf dieser Zeitspanne muß das Ergebnis der Zuweisung vorliegen.

```
function alpha (
  b := beta(a) rate 1 ms;
  c := gamma(b) rate 2 ms;
)
```

In diesem Beispiel ist die Berechnung der Funktionen **`beta`** und **`gamma`** somit nach insgesamt 3 ms beendet. Wird jetzt die Funktion **`alpha`** benutzt, kann sie maximal mit einer Rate von 3 ms aufgerufen werden. Wird sie mit einer höheren Rate aufgerufen, erzeugt der Compiler eine Fehlermeldung. Eine höhere Rate bedeutet hier und im folgenden, daß weniger Zeit für die Ausführung zur Verfügung steht. Der Wert für die Zeit ist zwar kleiner, dennoch spricht man von einer höheren Rate. Ebenso ist mit einer niedrigeren Rate gemeint, daß mehr Zeit zur Verfügung steht.

Wird die Funktion **`alpha`** mit einer niedrigeren Rate aufgerufen, wird vom Compiler eine zusätzliche Wartezeit eingefügt. Wird also die Beispielfunktion mit einer Rate 10 ms aufgerufen, werden zunächst **`beta`** und **`gamma`** aufgerufen. Dann wird 7 ms gewartet, um anschließend die nächste Berechnung zu starten. Alternativ können die Zeitbedingungen weggelassen werden. Zur Verdeutlichung folgte ein leicht geändertes Beispiel:

```
function alpha (
  b := beta(a) rate 1 ms;
  c := gamma(b);
)
```

Wird die Funktion **`alpha`** hier nun mit einer Rate von 3 ms aufgerufen, ändert sich nichts. **`beta`** terminiert nach 1 ms und **`gamma`** nach 2 ms. Erfolgt der Aufruf aber mit der Rate 10 ms stehen **`gamma`** 9 ms zur Berechnung des Ergebnisses zur Verfügung. Der Compiler hat somit mehr Freiheiten und kann möglicherweise eine billigere Lösung finden.

7.2 Relative Zeitbedingungen

Manchmal kann es sinnvoll sein, als Rate keine absoluten Zeiten anzugeben, sondern nur einen prozentualen Anteil der zur Verfügung stehenden Zeit:

```
function alpha (
```

```
  b := beta(a) rate 0.5;
  c := gamma(b) rate 0.5;
)
```

Wird die Funktion **alpha** jetzt mit einer Rate von 10 ms aufgerufen, muß sowohl **beta** als auch **gamma** nach 5 ms terminieren. Selbstverständlich kann wieder eine oder mehrere Raten weggelassen werden. Der Compiler wird sie dann passend ergänzen. Ist die Summe der relativen Rate innerhalb einer Funktion größer als 1, erzeugt der Compiler eine Fehlermeldung. Absolute und relative Zeitbedingungen können gemischt werden. Hierbei ist wichtig, daß die Summe der Raten der Teilberechnungen nicht größer als die insgesamt für die Funktion angegebene Rate wird.

7.3 Zeitbedingungen bei parallelen Instruktionen

Bei Instruktionen, die parallel ausgeführt werden sollen, muß die Rate identisch oder nicht spezifiziert sein.

```
function alpha (
  b1 := beta(a1) rate 0.5 : b2 := beta(a1) rate 0.5;
  c := gamma(b1, b2) rate 0.5;
)
```

In diesem Beispiel werden zwei parallel arbeitende Instanzen von **beta** erzeugt. Beiden wird 50% der für die Berechnung von **alpha** zur Verfügung stehenden Zeit zu gewiesen. Da beide Instanzen sowieso zur gleichen Zeit terminieren müssen, reicht es aus, wenn die Rate einmal angegeben wird:

```
function alpha (
  b1 := beta(a1) rate 0.5 : b2 := beta(a1);
  c := gamma(b1, b2);
)
```

In der Praxis terminieren parallele Instruktionen natürlich nicht zur gleichen Zeit. Vielmehr muß die schnellere auf die langsamere warten. Es kann auch sein, daß beide Instruktionen warten müssen. Dies ist in diesem Beispiel der Fall, wenn beide Funktionsaufrufe bereits vor Ablauf der zur Verfügung stehenden Zeit terminieren. Auch bei der Funktion **gamma** kann die Rate in diesem Beispiel weggelassen werden, da in jedem Fall noch 50% der Berechnungszeit für die Ausführung der Berechnung übrig ist.

7.4 Zeitbedingungen für Verzweigungen und Schleifen

Verzweigungen (bedingte Ausführungen) und Schleifen können ebenso wie Zuweisungen Raten zugeordnet werden. Im Falle einer Verzweigung gilt die Rate für alle Alternativen. Auch wenn kein Default-Fall angegeben wurde, gilt für diesen Fall die jeweilige Rate.

Die Rate für eine Schleife legt die gesamte Ausführungszeit der Schleife fest. Dies bedeutet, daß für jeden Durchlauf der Schleife die Rate geteilt durch die maximale Anzahl der Schleifendurchläufe als Ausführungszeit zur Verfügung steht. Im Falle einer „reinen“

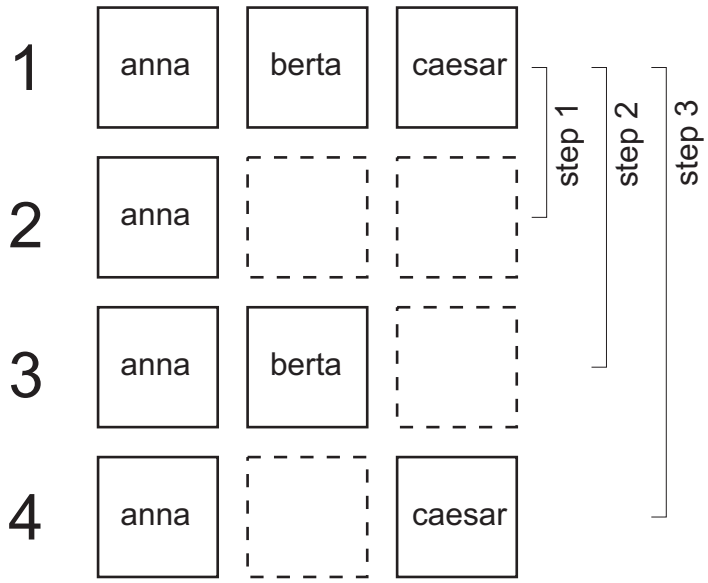
while-Schleife wird die mit Hilfe einer Compilervariablen spezifizierte maximale Wiederholungszahl zugrunde gelegt.

7.5 Ausführungszeitpunkte

In *CoLa* ist es möglich, die Ausführung einer Funktion von einem bestimmten Zeitpunkt abhängig zu machen. Da in *CoLa* Funktionen wiederholt angestoßen werden, erfolgt die Festlegung der Zeitpunkte eines Funktionsaufrufs durch Angabe einer Schrittweite. Die Schrittweite wird mit Hilfe des Schlüsselwortes **step** angegeben. Auf **step** muß eine ganze positive Zahl folgen. Die absoluten Zeitpunkte eines Funktionsaufrufs sind vom Kontext abhängig. Zwei Funktionen, die an unterschiedlichen Stellen einer Systembeschreibung mit der gleichen Schrittweite spezifiziert wurden, können zu unterschiedlichen Zeiten und unterschiedlich häufig aufgerufen werden. Der Zeitpunkt des Aufrufs hängt auch davon ab, wann der Kontrollfluß die Stelle des Aufrufs erreicht. Die Schrittweiten sind also keineswegs absolute Werte im ganzen System.

```
(int(8) out) function toplevel(int(8) in);

function toplevel (
  int(8) a, b, c;
  c := caesar(in) step 3;
  b := berta(in, c) step 2;
  out := anna(in, c, b);
)
```



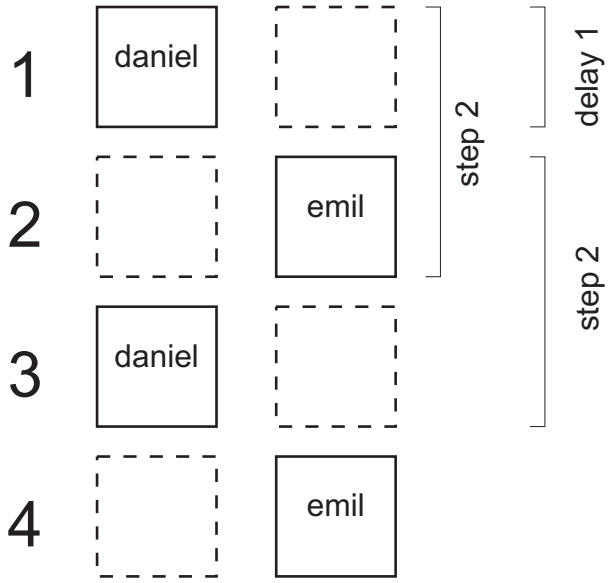
In diesem Beispiel werden in der Toplevel-Funktion die drei Funktionen **anna**, **berta** und **caesar** aufgerufen. Alle drei Aufrufe erfolgen mit einer unterschiedlichen Schrittweite. Wenn, wie im Fall der Funktion **anna**, keine Schrittweite angegeben wird, wird „**step 1**“ angenommen, d.h. die Funktion wird immer ausgeführt. „Immer“ bedeutet hier, daß bei jedem Aufruf der Funktion **toplevel** auch **anna** aufgerufen wird. Die Funktion **berta** wird beim 1, 3, 5, 7, ... Aufruf der Funktion **toplevel** angestoßen. Analog hierzu erfolgt der Aufruf von **caesar** an den Zeitpunkten 1, 4, 7, 10, ...

Manchmal ist es erforderlich, daß der Aufruf einer Funktion erst mit einer gewissen Verzögerung erfolgt.

```
(int(8) out) function toplevel(int(8) in);

function toplevel (
  out := daniel(in) step 2;
  out := emil(in) step 2 delay 1;
)
```

In diesem Beispiel werden die Funktionen abwechselnd aufgerufen. Die Funktion **`daniel`** wird an den Zeitpunkten 1, 3, 5, ... aufgerufen. Die Funktion **`emil`** wird mit der gleichen Schrittweite aufgerufen. Der erste Aufruf erfolgt mit einer Verzögerung von einem Zeitschritt, so daß der Aufruf an den Zeitpunkten 2, 4, 6, ... erfolgt.



8 Ausnahmen und Unterbrechungen

Um robuste und fehlertolerante Systeme zu beschreiben, ist eine Ausnahmebehandlung (*exception handling*) notwendig. Diese ermöglicht die Reaktion auf unerwartete oder fehlerhafte Ereignisse. Im folgenden wird zwischen eingebauten und benutzerdefinierten Ausnahmen unterschieden. Alle eingebauten Ausnahmen sind Laufzeitfehler, d.h. Fehler die zur Übersetzungszeit nicht erkannt werden können.

Ausnahmen werden innerhalb einer Funktion erzeugt und am Ende der Funktion abgefangen. Sie sind ein Spezialfall der Unterbrechungen. Unterbrechungen (*interrupts*) können auch außerhalb von Funktionen erzeugt werden. Sie müssen dann als Eingangsparameter der Funktion übergeben werden.

8.1 Abfangen von Unterbrechungen

In *CoLa* können Behandlungsroutinen für Unterbrechungen an Funktionen assoziiert werden. Für jede Funktion kann so ein eigenes Verhalten zur Reaktion auf Ausnahmen definiert werden. Tritt in einer *CoLa*-Funktion eine Ausnahme auf, wird der entsprechende Abschnitt am Ende der Funktion zur Fehlerbehandlung aufgerufen.

Grundsätzlich könnte eine Funktion nach der Behandlung einer Unterbrechung an der Stelle, an der sie unterbrochen wurde, fortgesetzt werden (Wiederaufnahmemodell) oder beendet werden (Terminierungsmodell). In *CoLa* wird das Terminierungsmodell verwendet. Ausnahmen können sich fortpflanzen. Wenn eine Funktion keine Routine zur Ausnahmebehandlung enthält, wird die Ausnahme an die darüber liegende Abstraktionsebene weitergereicht.

8.2 Laufzeitfehler

In *CoLa* gibt es drei verschiedene Laufzeitfehler:

- ★ Überschreiten der maximalen Rekursionstiefe (**`max_recursion_depth`**),
- ★ Überschreiten der maximalen Schleifenanzahl (**`max_loop_runs`**),
- ★ Division durch Null (**`division_by_zero`**).

Für jeden Laufzeitfehler sind bereits Unterbrechungen deklariert, so daß der Programmierer diese direkt nutzen kann. Sie werden automatisch vom System signalisiert.

```
(num(16) z) function teilen(num(16) a, num(16) b) (
  z := a / b;
)
catch division_by_zero (
  z := 0;
)
```

Wird die Funktion **`teilen`** mit **`b = 0`** aufgerufen, so erzeugt das System eine **`division_by_zero`** Ausnahme. Diese wird abgefangen und dem Ausgangsparameter **`z`** wird eine Null zugewiesen.

8.3 Benutzerdefinierte Ausnahmen

Der Entwerfer kann eigene Ausnahmen deklarieren und signalisieren. So könnte z.B. eine Ausnahme für den Fall einer Bereichsüberschreitungen erzeugt werden:

```
(num(16) z) function summe(num(16) a, num(16) b) (
  interrupt overflow;
  z := a + b;
  if (z > 999) (
    raise overflow;
  )
)
catch overflow (
  z := 999;
)
```

Die Ausnahme **`overflow`** wird zunächst deklariert. Ist das Ergebnis der Addition größer als 999 wird sie signalisiert. Am Ende der Funktion **`summe`** wird die Ausnahme abgefangen und **`z`** der maximal mögliche Wert 999 zugewiesen.

8.4 Unterbrechungen

Ausnahmen sind Ereignisse innerhalb einer Funktion, auf die eine angemessene Reaktion erfolgen muß. Auf Ereignisse, die hingegen außerhalb einer Funktion auftreten, kann mit einer Unterbrechung (*interrupt*) reagiert werden. Das Abfangen erfolgt anlog zum Abfangen einer Ausnahme.

```
(num(32) z) function zaehlen(interrupt stop);
function zaehlen (
  z := 0;
  while true (
    z := z + 1;
  )
)
catch stop (
)
catch max_loop_runs (
)
```

Die Funktion **`zaehlen`** erhöht **`z`** endlos um eins. Währenddessen können zwei unterschiedliche Unterbrechungen auftreten. Zum einen kann die externe Unterbrechung **`stop`** signalisiert werden. Zum anderen kann die maximale Anzahl der Schleifendurchläufe erreicht werden. In beiden Fällen wird die Funktion einfach beendet.

Eine externe Unterbrechung kann auch von einer parallel gestarteten Funktion signalisiert werden:

```
() function toplevel();
function toplevel(
  num(32) zahl;
  interrupt stop_zaehlen;
  z := zaehlen(stop_zaehlen) :
    stop_zaehlen := erzeuge_interrupt();
)

(interrupt stop) function  erzeuge_interrupt();
function erzeuge_interrupt(
  ...
  raise stop;
  ...
)
```

Die Deklaration einer Unterbrechung als Rückgabewert ermöglicht es, einer Funktion eine Unterbrechung nach außen zu signalisieren. Zu beachten ist, daß eine Unterbrechung sofort aktiv wird, wenn sie signalisiert wird. „Normale“ Rückgabewerte erhalten ihren Wert erst, wenn die Funktion terminiert. Die Zuweisung zu **`stop_zaehlen`** ist wichtig, um eine eindeutige Zuordnung der Unterbrechung zu erhalten.

9 Kurzreferenz

9.1 ':='

Syntax:

```
<AssignExpression> ::=
<PrimaryExpression> ':=' <ArithExpression> <RateStep> ';'
```

Semantik:
Variablen in *CoLa* erhalten mit Hilfe des Zuweisungsoperators ':=' einen Wert.

9.2 ':~'

Syntax:

```
<AssignExpression> ::=
<PrimaryExpression> ':~' <ArithExpression> <RateStep> ';'
```

Semantik:
Variablen können auch mit Hilfe des Zuweisungsoperators ':~' einen neuen Wert erhalten. Hierbei wird ggf. eine Konvertierung der Bitbreiten vorgenommen. Dies kann zu Wertverlusten führen.

9.3 '==' / '!=' / '<' / '>=' / '>' / '<='

Syntax:

```
<Relation> ::=
'(' <Relation> ')' |
<ArithExpression> '==' <ArithExpression> |
<ArithExpression> '!=' <ArithExpression> |
<ArithExpression> '<' <ArithExpression> |
<ArithExpression> '>=' <ArithExpression> |
<ArithExpression> '>' <ArithExpression> |
<ArithExpression> '<=' <ArithExpression>
```

Semantik:
Mit Hilfe der Vergleichsoperatoren können Variablen verglichen werden. Das Ergebnis einer Vergleichsoperation ist vom Typ **`bool`**. Komplexe Typen werden komponentenweise verglichen.

9.4 '>>' / '<<'

Syntax:

```
<ShiftExpression> ::=
```

```
<AdditiveExpression> |
<ShiftExpression> '<<' <AdditiveExpression> |
<ShiftExpression> '>>' <AdditiveExpression>


<AssignExpression> ::=
<PrimaryExpression> '<<' <RateStep> ';' |
<PrimaryExpression> '>>' <RateStep> ';'
```

Semantik:
Der Operator '**>>**' bewirkt das bitweise Schieben des Werts einer Variablen nach rechts. Der Operator '**<<**' schiebt den Wert nach links. Wenn der Operator ohne Argument angewendet wird, wird der Wert der Variablen rotiert, d.h. der „rausfallende“ Wert wird auf der anderen Seite wieder hinein geschoben. Wird ein Argument angegeben, so wird sein Wert an Stelle dessen hinzugefügt. Der „rausfallende“ Wert wird verworfen.

Die beiden Schiebeoperatoren können sowohl auf Variablen vom Typ **num** als auch vom Typ **vector** angewendet werden. Im ersten Fall werden einzelne Bits geschoben, im zweiten Fall werden die Elemente des Vektors parallel bitweise verschoben.

9.5 '#>' / '<#'

Syntax:

```
<ShiftExpression> ::=
<AdditiveExpression> |
<ShiftExpression> '<#' <AdditiveExpression> |
<ShiftExpression> '#>' <AdditiveExpression>


<AssignExpression> ::=
<PrimaryExpression> '<#' <RateStep> ';' |
<PrimaryExpression> '#>' <RateStep> ';'
```

Semantik:
Der Operator '**#>**' bewirkt das Schieben der Komponenten eines Vektors nach rechts. Der Operator '**<#**' schiebt die Komponenten nach links. Wenn der Operator ohne Argument angewendet wird, werden die Komponenten rotiert, d.h. der „rausfallende“ Wert wird auf der anderen Seite wieder hinein geschoben. Wird ein Argument angegeben, so wird sein Wert an Stelle dessen hinzugefügt. Der „rausfallende“ Wert wird verworfen.

9.6 '&' / '|'

Syntax:

```
<ArithExpression> ::=
<ShiftExpression> |
<ArithExpression> '&' <ShiftExpression> |
<ArithExpression> '|' <ShiftExpression>
```

Semantik:
'&' und '|' gehören zu den arithmetischen Operatoren. Sie bewirken das die beiden Argumente bitweise „UND“ bzw. „ODER“ verknüpft werden. Beide können auch auf Vektoren angewendet werden. In diesem Fall werden die einzelnen Komponenten des Vektors „UND“ bzw. „ODER“ verknüpft. Das Ergebnis der Operation ist in diesem Fall wieder ein Vektor.

9.7 AND

Syntax:

```
<LogicExpression> ::=
<LogicExpression> AND <Relation>
```

Semantik:
Der Operator '**and**' gehört zu den logischen Operatoren. Er verknüpft zwei Argumente vom Typ **bool** gemäß der folgenden Wertetabelle:

a	b	a and b
false	false	false
false	true	false
true	false	false
true	true	true

9.8 AUTOMAT

Syntax:

```
<AutomatDeclaration> ::=
'(' <ParaList> ')' AUTOMAT IDENTIFIER '('<ParaList>')' ';'

<AutomatName> ::=
AUTOMAT IDENTIFIER

<AutomatDefinition> ::=
<AutomatName> '(' <AutomatBlock> ')' <ExceptionHandler>
```

Semantik:
Automaten werden analog zu Funktionen zunächst deklariert und dann definiert. Jeder Automat besteht aus einzelnen Zuständen. Die Zustände wiederum bestehen aus mehreren optionalen bedingten Zustandsübergängen und einem unbedingten Default-Übergang.

9.9 BOOL

Syntax:

```
<TypeSpecifier> ::=
BOOL
```

Semantik:
Mit dem Schlüsselwort **`bool`** werden boolsche Variablen deklariert. Sie können die Werte '**`true`**' und '**`false`**' annehmen. Variablen vom Typ **`bool`** können implizit in Variablen vom Typ **`num(1)`** gewandelt werden. Der Wert '**`true`**' wird in eine Eins und der Wert '**`false`**' in eine Null gewandelt.

9.10 CHAR

Syntax:

```
<TypeSpecifier> ::=
CHAR
```

Semantik:
Mit dem Schlüsselwort **`char`** werden Variablen, die ein ASCII-Zeichen aufnehmen können, deklariert. Variablen vom Typ **`char`** können implizit in Variablen vom Typ **`num(8)`** umgewandelt werden.

9.11 CONST

Syntax:

```
<ConstDeclaration> ::=
CONST <VarDeclaration> ':=' <ConstExpressionSingle>
```

Semantik:
Mit **`const`** werden Variablen als konstant deklariert. Innerhalb der Deklaration wird ihr konstanter Wert festgelegt.

9.12 DELAY

Syntax:

```
<Delay> ::=
DELAY INTVALUE
```

Semantik:
Mit Hilfe von **`delay`** werden Anweisungen bei ihrem ersten Aufruf um die hinter **`delay`** spezifizierten Zeiteinheiten verzögert. Wird kein **`delay`** angegeben wird „**`delay 0`**“ angenommen.

9.13 FUNCTION

Syntax:

```
<FuncDeclaration> ::=
'('<ParaList>')' FUNCTION IDENTIFIER '('<ParaList>')' ';'
```

```
<FuncName> ::=
FUNCTION IDENTIFIER

<FuncDefinition> ::=
<FuncName> <Block> <ExceptionHandler>
```

Semantik:
Mit Hilfe des Schlüsselwortes **`function`** werden Funktionen sowohl deklariert als auch definiert. Bei der Deklaration werden die Ein- und Ausgangsparameter festgelegt. Bei der Definition wird ein Block aus einzelnen Anweisungen an die Funktion gebunden. Die Toplevel-Funktion, die den gleichen Namen wie die Datei hat, wird vom Laufzeitsystem wiederholt und permanent abgearbeitet.

9.14 IF / ELSE

Syntax:

```
<SelectionStatement> ::=
IF '(' <LogicExpression> ')' <CreateBlock> <Block> |
IF '(' <LogicExpression> ')' <CreateBlock> <Block> ELSE
<CreateBlock> <Block>
```

Semantik:
Mit Hilfe des **`if`** / **`else`** Konstrukts werden, wie in gängigen Programmiersprachen, Anweisungen bedingt ausgeführt. Wird der logische Ausdruck nach dem **`if`** zu **`true`** ausgewertet, werden die folgenden Anweisungen ausgeführt. Anderenfalls werden die nach dem **`else`** stehenden Anweisungen ausgeführt. Die **`else`** Anweisung ist optional.

9.15 INTERRUPT

Syntax:

```
<TypeSpecifier> ::=
INTERRUPT
```

Semantik:
Mit dem Schlüsselwort **`interrupt`** wird eine Unterbrechung deklariert. Die Unterbrechung wird mit dem Befehl **`raise`** ausgelöst und ist für eine Zeiteinheit aktiv.

9.16 LOOP / WHILE

Syntax:

```
<IterationStatement> ::=
LOOP IDENTIFIER <Range> WHILE '(' <LogicExpression> ')' |
LOOP IDENTIFIER <Range> |
WHILE '(' <LogicExpression> ')'
```

Semantik:

In *CoLa* gibt es zwei Arten von Schleifen die **loop**- und die **while**-Schleife. Die **loop**-Schleife deklariert eine Schleifenvariable und ein festes Intervall, das diese durchlaufen soll. Im Gegensatz hierzu wird die **while**-Schleife solange durchlaufen, wie der auf das **while** folgende logische Ausdruck zu **true** ausgewertet werden kann. Beide Schleifenarten können auch zu einer kombiniert werden. Die Schleife wird dann beendet, wenn das Ende des Intervalls erreicht ist *oder* die Bedingung hinter **while** nicht mehr wahr ist.

9.17 NOT

Syntax:

```
<LogicExpression> ::=
NOT <Relation>
```

Semantik:
Der Operator '**not**' gehört zu den logischen Operatoren. Er invertiert ein Argument vom Typ **bool** gemäß der folgenden Wertetabelle:

a	not a
false	true
true	false

9.18 NUM / SNUM

Syntax:

```
<TypeSpecifier> ::=
NUM '(' INTVALUE ')' |
NUM '(' INTVALUE ',' INTVALUE ')' |
SNUM '(' INTVALUE ')' |
SNUM '(' INTVALUE ',' INTVALUE ')'
```

Semantik:
Mit Hilfe von **num** und **snum** werden ganze Zahlen und Festkommazahlen repräsentiert. Es wird zwischen Werten ohne Vorzeichen (**num**) und Werten mit Vorzeichen (**snum**) unterschieden. Vor- und Nachkommastellen der Variablen werden durch ein Komma in der Deklaration getrennt:

```
num (7,3) alpha;
num (7,0) beta;
num (7) gamma;
```

Die Deklaration der Variablen **beta** und **gamma** ist äquivalent. Wenn die Anzahl der Nachkommastellen gleich Null ist, kann auf die Angabe verzichtet werden.

9.19 OR

Syntax:

```
<LogicExpression> ::=
<LogicExpression> OR <Relation>
```

Semantik:
Der Operator '**or**' gehört zu den logischen Operatoren. Er verknüpft zwei Argumente vom Typ **bool** gemäß der folgenden Wertetabelle:

a	b	a or b
false	false	false
false	true	true
true	false	true
true	true	true

9.20 RATE

Syntax:

```
<Rate> ::=
RATE INTVALUE <Time> |
RATE FPVALUE

<Time> ::=
MS |
US |
NS
```

Semantik:
Mit Hilfe von **rate** werden einzelnen Anweisungen Ausführungszeiten zugeordnet. Die Ausführungszeiten können als absolute Zeit in Milli-, Mikro- oder Nanosekunden angegeben werden, alternativ auch als relative Zeit, d.h. als prozentualer Anteil der insgesamt zur Verfügung stehenden Zeit.

9.21 RAISE / CATCH

Syntax:

```
<ExceptionHandler> ::=
CATCH IDENTIFIER <CreateBlock> <Block> <ExceptionHandler>
```

Semantik:
Mit Hilfe von **raise** wird eine Unterbrechung ausgelöst. Sie ist genau eine Zeiteinheit aktiv. Auf **catch** folgt der Programmcode, der auf die Unterbrechung reagiert.

9.22 STEP

Syntax:

```
<Step> ::=
STEP INTVALUE
```

Semantik:
Mit Hilfe von **`step`** werden Anweisungen nur zu dem hinter **`step`** spezifizierten Zeitpunkt ausgeführt. Wird kein **`step`** angegeben, wird „**`step 1`**“ angenommen, d.h. die Anweisung wird zu jedem Zeitpunkt angestoßen.

9.23 STRUCT

Syntax:

```
<TypeDefinition> ::=
STRUCT IDENTIFIER '(' <StructList> ')'
```

Semantik:
Mit einem **`struct`** können Variablen zu einer Struktur zusammengefaßt werden.

9.24 SWITCH

Syntax:

```
<SelectionStatement> ::=
<Switch> <CaseList> ')'

<Switch> ::=
SWITCH IDENTIFIER '('

<CaseList> ::=
CASE <ConstExpressionSingle> <Block> <CaseList> |
DEFAULT <Block>
```

Semantik:
In Abhängigkeit des Wertes der Variable nach **`switch`** wird der entsprechende Block hinter **`case`** ausgeführt. Paßt keiner der **`case`**-Anweisungen, so wird der Block nach **`default`** ausgeführt. Die Auswahlvariable kann einen beliebigen Typen haben.

9.25 TRUE / FALSE

Syntax:

```
<ConstExpressionSingle> ::=
TRUE |
FALSE
```

Semantik:
Mit **`true`** und **`false`** werden die boolschen Werte repräsentiert. Sie können Variablen vom Typ **`bool`** zugewiesen werden oder mit Hilfe der logischen Operatoren zu komplexen logischen Ausdrücken kombiniert werden. Werden die boolschen Werte zu ganzzahligen Werten konvertiert, wird **`true`** zu 1 und **`false`** zu 0.

9.26 TYPE

Syntax:

```
<TypeDefinition> ::=
TYPE IDENTIFIER <Type>
```

Semantik:
`type` gestattet die Definition von benutzerdefinierten Typen. Genaugenommen wird ein Alias auf einen Typ definiert, um kurze und aussagekräftige Bezeichner für Typen zu ermöglichen. *CoLa* unterscheidet Typen nicht anhand ihres Namen (Namesäquivalenz), sondern anhand ihrer Struktur (strukturelle Äquivalenz).

9.27 VECTOR

Syntax:

```
<Type> ::=
VECTOR '(' INTVALUE ')' <Type>
```

Semantik:
Mit Hilfe von **`vector`** können mehrere Komponenten zu einem Array zusammengefaßt werden. Die einzelnen Komponenten dürfen sowohl vordefinierte Typen als auch benutzerdefinierte Typen sein. Alle Komponenten müssen allerdings den gleichen Typ haben. Um Komponenten von unterschiedlichem Typ zusammenzufassen kann **`struct`** verwendet werden. Auf einzelne Komponenten kann mit Hilfe der eckigen Klammern '**`[`**' und '**`]`**' zugegriffen werden.

9.28 XOR

Syntax:

```
<LogicExpression> ::=
<LogicExpression> XOR <Relation>
```

Semantik:

Der Operator '**xor**' gehört zu den logischen Operatoren. Er verknüpft zwei Argumente vom Typ **bool** gemäß der folgenden Wertetabelle:

a	b	a xor b
false	false	false
false	true	true
true	false	true
true	true	false

10 Literaturverzeichnis

[GAJSKI]
Gajski, Daniel D.: „Silicon Compilation“, 450 Seiten, Reading, Mass. : Addison-Wesley, 1988

[LOUDEN]
Louden, Kenneth C.: „Programmiersprachen : Grundlagen, Konzepte, Entwurf“, 1. Auflage, 810 Seiten, Bonn [u.a.] : Internat. Thomson Publ., 1994

[MARWEDEL]
Marwedel, Peter: „Synthese und Simulation von VLSI-Systemen“, 196 Seiten, München [u.a.] : Hanser, 1993

[SCHNEIDER]
Schneider, Hans-Jochen: „Lexikon der Informatik und Datenverarbeitung“, 3., aktualisierte und wesentl. erw. Auflage, 989 Seiten, München [u.a.] : Oldenbourg, 1991

[TEICH]
Teich, Jürgen: „Digitale Hardware/Software-Systeme: Synthese und Optimierung“, 514 Seiten, Berlin [u.a.] : Springer, 1997

11 Anhang

11.1 Reservierte Wörter

```
AND
AUTOMAT
BOOL
CASE
CATCH
CHAR
CONST
DEFAULT
DELAY
ELSE
FALSE
FUNCTION
IF
INTERRUPT
LOOP
MS
NEXTSTATE
NOT
NS
NUM
OR
RAISE
RATE
SNUM
STEP
STRUCT
SWITCH
TO
TRUE
TYPE
US
VECTOR
WHILE
XOR
```

11.2 Vollständige Syntax

```
<Start> ::=
<Program>

<Range> ::=
'(' INTVALUE TO INTVALUE ')'
```

```
<Rate> ::=
RATE INTVALUE <Time> |
RATE FPVALUE

<Time> ::=
MS |
US |
NS

<Step> ::=
STEP INTVALUE

<Delay> ::=
DELAY INTVALUE

<RateStep> ::=
[ <Rate> ] [ <Step> ] [ <Delay> ]

<Program> ::=
<FuncDeclarationList> <FuncDefinitionList> |
<TypeDefinitionList> <FuncDeclarationList>
<FuncDefinitionList>

<FuncDeclarationList> ::=
<FuncDeclarationList> <FuncDeclaration> |
<FuncDeclarationList> <AutomatDeclaration> |
<AutomatDeclaration> |
<FuncDeclaration>

<FuncDefinitionList> ::=
<FuncDefinitionList> <FuncDefinition> |
<FuncDefinitionList> <AutomatDefinition> |
<AutomatDefinition> |
<FuncDefinition>

<FuncDeclaration> ::=
'(' <ParaList> ')' FUNCTION IDENTIFIER '(' <ParaList> ')' ';'

<FuncName> ::=
FUNCTION IDENTIFIER

<FuncDefinition> ::=
<FuncName> <Block> <ExceptionHandler>

<AutomatDeclaration> ::=
'(' <ParaList> ')' AUTOMAT IDENTIFIER '(' <ParaList> ')' ';'

<AutomatName> ::=
AUTOMAT IDENTIFIER

<AutomatDefinition> ::=
<AutomatName> '(' <AutomatBlock> ')' <ExceptionHandler>
```

```
<ExceptionHandler> ::=
CATCH IDENTIFIER <CreateBlock> <Block> <ExceptionHandler>

<TypeDefinitionList> ::=
<TypeDefinitionList> <TypeDefinition> ';' |
<TypeDefinition> ';'

<TypeDefinition> ::=
TYPE IDENTIFIER <Type> |
STRUCT IDENTIFIER '(' <StructList> ')'

<StructList> ::=
<StructList> ',' <VarDeclaration> |
<VarDeclaration>

<ParaList> ::=
<ParaListEx>

<ParaListEx> ::=
<VarDeclaration> |
<ParaListEx> ',' <VarDeclaration>

<VarDeclarationList> ::=
<VarDeclarationList> <VarDeclarationLine> ';' |
<VarDeclarationLine> ';'

<VarDeclarationLine> ::=
<VarDeclaration> <VarDeclarationEx> |
<ConstDeclaration>

<VarDeclarationEx> ::=
<VarDeclarationEx> ',' IDENTIFIER

<VarDeclaration> ::=
<Type> IDENTIFIER |
IDENTIFIER IDENTIFIER <PrimaryExpressionVector> ','
<ConstExpressionSingle>

<ConstDeclaration> ::=
CONST <VarDeclaration> ASSIGN <ConstExpressionSingle>

<Type> ::=
<TypeSpecifier> |
VECTOR '(' INTVALUE ')' <Type>

<TypeSpecifier> ::=
BOOL |
CHAR |
NUM '(' INTVALUE ')' |
NUM '(' INTVALUE ',' INTVALUE ')' |
SNUM '(' INTVALUE ')' |
```

```
SNUM '(' INTVALUE ',' INTVALUE ')' |
INTERRUPT

<Block> ::=
'(' ')' |
'(' <DataStatement> ')' |
'(' <VarDeclarationList> <DataStatement> ')'

<DataStatement> ::=
<DataBlock> |
<DataBlock> <ControlStatement> |
<ControlStatement>

<ControlStatement> ::=
<ControlBlock> |
<ControlBlock> <DataStatement>

<DataBlock> ::=
<DataBlock> <AssignExpression> |
<AssignExpression>

<ControlBlock> ::=
<SelectionStatement> <RateStep> |
<IterationStatement> <RateStep> <Block>

<SelectionStatement> ::=
IF '(' <LogicExpression> ')' <CreateBlock> <Block> |
IF '(' <LogicExpression> ')' <CreateBlock> <Block> ELSE
<CreateBlock> <Block> |
<Switch> <CaseList> ')'

<CreateBlock> ::=

<Switch> ::=
SWITCH IDENTIFIER '('

<CaseList> ::=
CASE <ConstExpressionSingle> [ <CreateBlock> ] <Block>
<CaseList> |
DEFAULT [ <CreateBlock> ] <Block>

<IterationStatement> ::=
LOOP IDENTIFIER <Range> WHILE '(' <LogicExpression> ')' |
LOOP IDENTIFIER <Range> |
WHILE '(' <LogicExpression> ')'

<AutomatBlock> ::=
<AutomatAssign> <AutomatBlock> |
<AutomatBody>

<AutomatAssign> ::=
<AssignExpression>
```

```
<AutomatBody> ::=
<AutomatBody> <AutomatState> |
<AutomatState>

<AutomatState> ::=
<StateIdentifer> <StateAssignments>

<StateIdentifer> ::=
IDENTIFIER ':'

<StateAssignments> ::=
<AutomatTransition> |
<AutomatAssign> <StateAssignments>

<AutomatTransition> ::=
<DefaultTransition> |
<StandardTransition> <AutomatTransition>

<StandardTransition> ::=
'(' <LogicExpression> ')' NEXTSTATE IDENTIFIER ';'

<DefaultTransition> ::=
NEXTSTATE IDENTIFIER ';'

<LogicExpression> ::=
<LogicExpression> AND <Relation> |
<LogicExpression> OR <Relation> |
<LogicExpression> XOR <Relation> |
NOT <Relation> |
<Relation>

<Relation> ::=
'(' <Relation> ')' |
<ArithExpression> EQ <ArithExpression> |
<ArithExpression> NEQ <ArithExpression> |
<ArithExpression> LESS <ArithExpression> |
<ArithExpression> NLESS <ArithExpression> |
<ArithExpression> GREATER <ArithExpression> |
<ArithExpression> NGREATER <ArithExpression>

<AssignExpression> ::=
<PrimaryExpression> ASSIGN <ArithExpression> <RateStep> ';' |
<PrimaryExpression> SL <RateStep> ';' |
<PrimaryExpression> SR <RateStep> ';'

<ArithExpression> ::=
<ShiftExpression> |
<ArithExpression> '&' <ShiftExpression> |
<ArithExpression> '|' <ShiftExpression>

<ShiftExpression> ::=
```

```
<AdditiveExpression> |
<ShiftExpression> SL <AdditiveExpression> |
<ShiftExpression> SR <AdditiveExpression>

<AdditiveExpression> ::=
<MultExpression> |
<AdditiveExpression> '+' <MultExpression> |
<AdditiveExpression> '-' <MultExpression>

<MultExpression> ::=
<PrimaryExpression> |
<MultExpression> '*' <PrimaryExpression> |
<MultExpression> '/' <PrimaryExpression>

<PrimaryExpression> ::=
<PrimaryExpressionSingle> |
'(' <PrimaryExpressionVector> ')'

<ConstExpressionSingle> ::=
INTVALUE |
FPVALUE |
TRUE |
FALSE |
CHARACTER

<PrimaryExpressionSingle> ::=
<SimpleAccess> <ExtAccess> |
<ConstExpressionSingle> |
'(' <ArithExpression> ')'

<PrimaryExpressionVector> ::=
<PrimaryExpressionVector> ',' <ConstExpressionSingle> |
<ConstExpressionSingle> ',' <ConstExpressionSingle>

<SimpleAccess> ::=
IDENTIFIER |
<ProcIdentifier> <ExpressionList> ')'

<ExtAccess> ::=
<ExtAccess> '[' <ArithExpression> ']' |
<ExtAccess> '.' IDENTIFIER

<ProcIdentifier> ::=
IDENTIFIER '('

<ExpressionList> ::=
<ExpressionList> ',' <ArithExpression> |
<ArithExpression>
```