SCARLET 0·85

Ein Simulations-Programm
für Zellularautomaten

—

Eine Dokumentation

Institut für Informatik
Justus-Liebig-Universität, Giessen

Mai 1998

Inhaltsverzeichnis

Einleitung

> *„A cellular automata machine is sitting on our desk, waiting for us to give a rule - the law that will govern a world.“*
>
> *(T. Toffoli, N. Margolus)*

Als A. Turing 1937 sein mathematisches Modell einer berechnungsuniversellen Maschine vorstellte, gab er damit den Anstoß zu einer neuen Theorie: Schnell entstanden weitere Kalküle, deren Mächtigkeiten der der „Turing-Maschine“ entsprachen. In diesem Zusammenhang entwickelte auch
John von Neumann, inspiriert durch S. Ulam, gegen Ende der Vierziger Jahre seine Theorie der „selbstreproduzierenden Automaten“, zu dem Zweck, einen realistischeren Weg zur Konstruktion universeller Rechenmaschinen aufzuzeigen. Dieser Vorschlag machte ihn zum offiziellen Erfinder der Zellularautomaten:

Von Neumann ging dabei (grob gesprochen) von einem endlichen, in Zellen unterteilten Ausschnitt des d-dimensionalen euklidischen Raumes aus. Die Entwicklung eines Systems, das in diesem Raum beobachtet wird, erklärt sich lokal durch sein Verhalten in den einzelnen Zellen. Dabei wird angenommen, daß das System homogen sei, d.h. daß sich alle Zellen unter gleichen Bedingungen gleich verhalten. Die Bedingungen werden festgelegt durch das Verhalten gewisser Zellen in der Nachbarschaft der betrachteten Zelle. Von dieser Nachbarschaftsstruktur wird wiederum angenommen, daß sie für alle Zellen gleich sei.

Trotz dieses vielversprechenden Ansatzes führten die Zellularautomaten über längere Zeit hinweg ein Schattendasein; verschiedene Versuche, mehr über ihre Eigenschaften in Erfahrung zu bringen, verliefen im Sande, andere fanden keine allgemeine Verbreitung.
Die Bewegung erfuhr erst 1970 den entscheidenden Antrieb, als J. Conway - mehr durch Zufall - mit einem ganz speziellen Zellularautomat, dem „Game of Life“, selbst das Interesse von Nichtwissenschaftlern an der Materie weckte.

Gleichzeitig lenkte er die Aufmerksamkeit der Allgemeinheit wieder verstärkt auf das weite Anwendungsgebiet der Simulation, das nun mit wachsender Leistungsfähigkeit vorhandener Computersysteme an Attraktivität gewann.

Von da ab bis heute konnten endlich einige wichtige theoretische Fragen geklärt werden, z.B. die der Invertierbarkeit von Zellularautomaten, ihrer Eigenschaften bzgl. Spracherkennung, und viele andere.

Doch auch auf dem Gebiet der Simulation blieb die Entwicklung nicht stehen: Hatte man zunächst vornehmlich das globale Verhalten beliebiger Zellularautomaten untersucht, mit dem Ziel, sie in entsprechende Klassen einzuteilen, so erhob sich nun aus der Ecke der Naturwissenschaften immer häufiger die umgekehrte Frage (inverse problem): Welche dieser Modelle sind dazu geeignet, die Wirklichkeit zu beschreiben, also z.B. ein System im Raum, das gewissen Gesetzmäßigkeiten unterliegt, die i.a. durch partielle Differentialgleichungen charakterisiert werden können?
Viel Beachtung fand in diesem Zusammenhang die Entwicklung von Modellen für Teilchenbewegungen in Gasen (lattice gases). Trotz dieser und anderer Erfolge sind viele wichtige Zusammenhänge zwischen Zellularautomaten und ihren stetigen Gegenstücken nach wie vor ungeklärt.
In Anpassung an die unterschiedlichen Bedürfnisse wurden im Laufe der Zeit verschiedene Abwandlungen bzw. Verallgemeinerungen des v. Neumann'schen *Standard-Zellularautomaten* entwickelt: In Analogie zu anderen theoretischen Konzepten interessierte man sich beispielsweise auch hier für die *nichtdeterministische* Variante; die Betrachtung von Teilchensystemen legte gar die Entwicklung der *probabilistischen Zellularautomaten* nahe, die unterschiedliches Verhalten mit bestimmten Wahrscheinlichkeiten zeigen können. Die Pallette reicht weiter über *asynchron*, bzw. bzgl. der verschiedensten Aspekte *inhomogen* arbeitende Zellularautomaten, oder solchen, die Abhängigkeiten innerhalb der Zustände zulassen *(hierarchische)*, um nur einige Beispiele zu nennen.

Aufgrund der hohen Komplexität der Probleme gewannen Simulationssysteme für Zellularautomaten als Arbeitsunterstützung sowohl für theoretische als auch für praktische Ansätze immer mehr an Bedeutung. Abgesehen von denen von T. Toffoli und N. Margolus zur Effizienzsteigerung entwickelten Zellularautomaten-Maschinen (CAM) arbeiten wohl die meisten dieser kommerziellen oder nichtkommerziellen Systeme rein Software-unterstützt und sind auf die unterschiedlichsten Rechnertypen und Betriebssysteme ausgelegt. Genauso unterschiedlich sind ihre jeweiligen Zielsetzungen: Einige unter ihnen beschränken sich auf spezielle Standard-Zellularautomaten (z.B., indem sie nur bestimmte Nachbarschaftsarten oder Raumdimensionen zulassen). Andere (zum Teil sehr experimentierfreudige Varianten) zielen konkret auf erweiterte Modelle und deren Anwendungen ab; so existiert gleich eine ganze Reihe von Programmen, die probabilistische Überführungsfunktionen oder räumliche Inhomogenität unterstützen. - Und damit sind bei weitem noch nicht die ausgefallensten Konzepte benannt. Allerdings sind gerade viele dieser Software-Pakete nur auf niedrigere Raumdimensionen ausgerichtet.

Scarlet, in seiner derzeitigen Version 0.8, erlaubt die Simulation von Standard-Zellularautomaten praktisch ohne Einschränkungen. Darüber hinaus ermöglicht es in einem natürlichen Rahmen (unter Zuhilfenahme zusätzlicher Zustände und der eingebauten Random-Funktion) auch die Simulation von Modell-Erweiterungen.

Diese Dokumentation richtet sich nun an alle, die irgendwie von Scarlet gehört haben, und die den Umgang damit nun erlernen wollen, sei es, um in Zukunft wissenschaftlich damit zu arbeiten, oder einfach nur, weil sie Freude an Zellularautomaten haben und nun auf eigene Faust ein wenig damit herumexperimentieren wollen.

Wir wünschen allen Benutzern viel Spaß und Erfolg dabei!

Kapitel 1

Zellularautomaten

1.1 Formale Definition

Um auch Neueinsteigern auf dem Gebiet der Zellularautomaten die Möglichkeit zu geben, sich innerhalb kurzer Zeit schon in gewinnbringender Weise mit SCARLET beschäftigen zu können, soll im ersten Kapitel versucht werden, den Begriff des Zellularautomaten anhand einiger grundlegender Definitionen vorzustellen und durch Beispiele zu erläutern. Dem bereits mit der Thematik vertrauten Leser mag der erste Abschnitt dazu dienen, sich der Bedeutung der hier verwendeten Begrifflichkeiten zu vergewissern. Die Definitionen dieses Abschnitts lehnen sich zum größten Teil an die in dem Bericht über "Parallele Automaten" von Martin Kutrib[5] an.

Wir nähern uns der Definition des Zellularautomaten durch Betrachtung des Zellularraums:

*Ein 5-Tupel (S, d, N, σ, q_0) heißt **Zellularraum**, falls gilt:*

1. *S ist eine endliche, nichtleere Zustandsmenge*
2. *$d \in \mathbb{N}_0$ ist die Raumdimension*
3. *N ist ein n-Tupel von paarweise verschiedenen Elementen aus $\mathbb{Z}^d$, $n \in \mathbb{N}$, auch **d-dimensionaler Nachbarschaftsindex vom Grade n** genannt.*
4. *$\sigma : S^n \longrightarrow S$ ist lokale Überführungsfunktion*
5. *$q_0 \in S$ ist der Ruhezustand, für den gilt: $\sigma(q_0, \ldots, q_0) = q_0$.*

Erschlagen? Hoffentlich nicht, denn die Vorstellung von Zellularautomaten, die sich letztlich hinter dieser Formalisierung verbirgt, ist erstaunlich anschaulich:

Man stelle sich zunächst einen einzelnen Moore-Automat vor: Dieser besitzt

- eine endliche, nichtleere Zustandsmenge S - genau wie der Zellularraum,
- eine Funktion β, die lediglich in Abhängigkeit des Zustandes, in dem er sich gerade befindet, eine Ausgabe berechnet,
- sowie eine Funktion λ, die aus dem aktuellen Zustand sowie einer Eingabe den Folgezustand des Automaten ermittelt.

(Daß Moore-Automaten auch über ausgezeichnete Start- und Endzustandsmengen verfügen, spielt für die folgenden Überlegungen keine Rolle und wird daher außer acht gelassen.)

„Und wo bitte bleibt da die Anschauung???“ - Na, das liegt doch auf der Hand: Sprechen wir von einem Automaten im landläufigen Sinne, dann meinen wir damit meistens ein Gebilde, dessen Funktionsweise theoretisch durch einen Moore-Automat beschrieben werden kann!

Das glauben Sie nicht so ganz? Dann betrachten Sie doch einfach einmal das folgende Beispiel:

Beispiel 1: Der „Rein-Raus-Automat“(RRA)
Diese (zugegebenermaßen nicht besonders geistreiche) Maschine besitze zwei sich gegenüberliegende Öffnungen und sei so konzipiert, daß sie Kugeln, die auf einer Seite eingeworfen werden, nach einer kurzen Verzögerung, in der keine weiteren Kugeln akzeptiert werden, nacheinander auf der gegenüberliegenden Seite wieder ausspuckt. Dabei sei es erlaubt, beide Öffnungen gleichzeitig als Einwurf zu benutzen.

Um zu demonstrieren, daß wir den RRA auch als Moore-Automat modellieren können, benötigen wir zunächst eine geeignete Zustandsmenge. Wir entscheiden uns dafür, einen Zustand durch die Anzahl und die Herkunft der im RRA zur Zeit der Verzögerungsphase befindlichen Kugeln zu charakterisieren („Keine Kugel“; „eine Kugel von rechts, zwei von links“; ...). Dann ist die Menge aller möglichen Zustände S nichtleer und endlich, da das Fassungsvermögen des RRA sicher endlich ist.
Weiterhin gibt es eine eineindeutige Zuordnung zwischen den Zuständen und den

„Zwei Kugeln von rechts, eine von links“	$\longrightarrow$	„Zwei Kugeln nach links, eine nach rechts auswerfen“
„eine Kugel“	$\longrightarrow$	„Keine Kugel auswerfen“

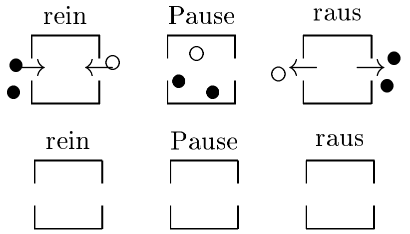


Abb. 1: Verschiedene, denkbare Zustände des RRA

Ausgaben des RRA (β existiert und ist bijektiv) :

Wie man sich leicht überlegen kann, besteht dieselbe Beziehung zwischen der Menge der Eingaben und der Menge der Zustände; insbesondere existiert eine Funktion λ wie oben gefordert (die speziell in diesem Beispiel nicht von S abhängt).

„Gut und schön, aber was haben Moore-Automaten mit Zellularräumen zu tun?“ Nicht gerade eben wenig, denn ein Zellularraum besteht nun letztendlich aus (unendlich) vielen miteinander kommunizierenden Automaten gleicher Bauart! - Aber, um noch einmal auf das Beispiel zurückzukommen:

Beispiel 2: Eindimensionaler RRA-Zellularraum:
Stellen wir uns nun unendlich viele RRA so in einer Reihe mit ihren Öffnungen aneinander gestellt vor, daß wir ihre Positionen mit Elementen aus $\mathbb{Z}$ bezeichnen können. Anders formuliert können wir uns die RRA also im (1-dimensionalen) Raum $\mathbb{Z}$ angeordnet vorstellen.
Weiterhin seien alle RRA „gleich getaktet“, d.h. alle wechseln in einem festen zeitlichen Rhythmus synchron zwischen Ein-/Auswurf- und Verzögerungsphase.
Dann besitzt jeder RRA einen linken und einen rechten Nachbarn und deren Ausgaben dienen ihm als Eingaben; umgekehrt tragen seine Ausgaben zu den Eingaben anderer RRA bei. Da wir bereits gesehen haben, daß β in unserem Fall bijektiv ist, können wir die Ausgabe eines RRA mit seinem Zustand identifizieren und somit auch sagen:

> *Der Folgezustand eines RRA wird durch die Zustände seiner Nachbarn bestimmt.*

(Da zur Konstruktion von Zellularräumen ausschließlich solche Moore-Automaten mit bijektiver Ausgabefunktion β betrachtet werden, braucht in Zukunft nie mehr

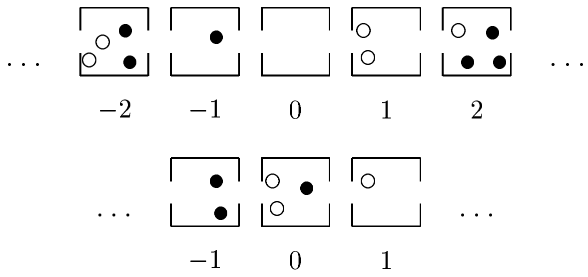


Abb. 2: Zwei aufeinanderfolgende Konfigurationen eines RRA-Zellularraums.

von Ein- oder Ausgabe die Rede sein, nur noch von Zuständen.)

Damit haben wir bereits implizit eine plausibel erscheinende lokale Überführungsfunktion $\sigma : S^n \longrightarrow S$ formuliert: Sie berechnet aus je zwei Zuständen einen neuen Zustand so, daß dieser der Folgezustand eines RRA ist, wenn die Ausgangszustände die Zustände seines linken und rechten Nachbars waren! Der Nachbarschaftsindex ist in diesem Fall also vom Grade zwei und der Ruhezustand q_0 ist schließlich der Zustand „keine Kugel", da ein RRA ebenfalls über „keine Kugel" verfügen wird, wenn seine Nachbarn zuvor keine besaßen.

„Plausibel hin, plausibel her: σ könnte genausogut auch ganz anders aussehen; da macht uns die Definition keinerlei Vorschriften! Und dieses seltsame Tupel des Nachbarschaftsindexes scheint schon ganz und gar überflüssig zu sein; deshalb ist es sicherheitshalber gar nicht erst zur Sprache gekommen!" Sachte, sachte! Diese Unklarheiten resultieren im wesentlichen daraus, daß wir Zellularräume bisher immer als eine Akkumulation im Prinzip eigenständiger Automaten angesehen haben. Tatsächlich ist man aber daran interessiert, den Zellularraum als Einheit zu verstehen - mit einem einheitlichen Verhalten - und die einzelnen Automaten als seine Bestandteile. (Man nennt sie deshalb auch ***Zellen***.) Die folgenden Definitionen garantieren uns eine solche „ganzheitliche" Sichtweise und schließen gleichzeitig die existierenden Lücken:

Eine Abbildung $c_t : \mathbb{Z}^d \longrightarrow S$ heißt ***Konfiguration*** *des Zellularraums im Zeitpunkt $t \in \mathbb{N}_0$. c_0 heißt auch* ***Startkonfiguration****.*
$c_t(i)$, $i \in \mathbb{Z}^d$ heißt ***Zustand*** *des Automaten i in der Konfiguration c_t.*

Sei A ein Alphabet, $N = (a_1, \dots, a_d)$ ein d-dimensionaler Nachbarschaftsindex und C_d die Menge aller Konfigurationen $c : \mathbb{Z}^d \longrightarrow A$. Eine Abbildung $\mathcal{T} : C_d \longrightarrow C_d$ heißt ***globale Überführungsfunktion****, wenn sie durch eine lokale Überführungsfunktion σ folgendermaßen induziert wird:*

Für alle $c \in C_d$ gilt:

$$\mathcal{T}(c) = c' \iff c'(i) = \sigma(c(i+a_1), \ldots, c(i+a_n)) \forall i \in \mathbb{Z}^d$$

Das wichtigste Hilfsmittel, um einen Zellularraum als ganzes zu betrachten, ist offenbar die Konfiguration: Sie liefert sozusagen die Zustände aller Automaten auf einen Blick. Statt die Zustandsveränderungen der einzelnen RRA von einer Verzögerungsphase zur nächsten zu betrachten, suchen wir nun eine Funktion $\mathcal{T}$, die in entsprechender Weise ganze Konfigurationen ineinander überführt. Gefunden haben wir $\mathcal{T}$, wenn wir deren globale Wirkungsweise auf die lokale Wirkungsweise von σ auf jede Zelle zurückführen können. - Dieses Prinzip funktioniert augenscheinlich nur dann, wenn der Zellularraum, wie in unserem Beispiel, aus gleichartigen Automaten besteht!

Beispiel 2: Fortsetzung:
Wählen wir nun als Nachbarschaftsindex das Tupel $N = (-1, 1)$, so erhalten wir für beliebiges $i \in \mathbb{Z}$ und $t \in \mathbb{N}_0$:

$$\sigma(c_t(i+a_1), c_t(i+a_2)) = \sigma(c_t(i-1), c_t(i+1))$$

Nun liefert σ angewandt auf die Zustände des linken und rechten Nachbars aber gerade den Folgezustand des RRA in i, also $c_{t+1}(i)$ für die Nachfolgekonfiguration c_{t+1} von c_t. Somit existiert die globale Überführungsfunktion $\mathcal{T}$. Gleichzeitig wurde hoffentlich etwas klarer, welche Rolle σ und N in diesem Zusammenhang spielen.

Im vorangegangenen Beispiel war es so gewesen, daß jene Zellen, die den Folgezustand einer anderen beeinflußten, gleichzeitig deren räumliche Nachbarn darstellten. Man hat sich daher angewöhnt, generell von „Nachbarn bezüglich eines bestimmten Nachbarschaftsindexes“ zu sprechen, wenn man die Zellen meint, die für eine bestimmte Zelle im Argumentbereich von σ liegen, oder etwas formaler:

Sei durch (S, d, N, σ, q_0) ein Zellularraum gegeben. Seien $i, j \in \mathbb{Z}^d$. Dann heißt Zelle i ***Nachbar*** *von Zelle j* ***bezüglich N****, falls eine Komponente k aus N existiert, so daß $i = j + k$ gilt.*

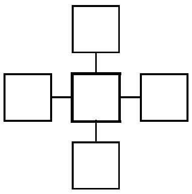
Mit anderen Worten enthält der Nachbarschaftsindex die relativen Positionen einer Zelle zu ihren Nachbarn.
Insbesondere darf eine Zelle auch ihr eigener Nachbar sein, wenn N die Komponente $(0, \ldots, 0) \in \mathbb{Z}^d$ enthält.

Bsp.3: Ein häufig angetroffenes Beispiel für einen d-dimensionalen Nachbarschaftsindex ist das sogenannte „verallgemeinerte von-Neumann-Raster“

$$H_k^d := \{i \in \mathbb{Z}^d : \sum_{n=1}^{d} |i_n| \leq k\}$$

Speziell für $d = 2$ und $k = 1$ entsteht die folgende Struktur:

H_1^2 :



Im Gegensatz zu den eingangs erwähnten Moore-Automaten liegt die Bedeutung der Zellularräume weniger in der Konstruktion von Maschinen nach ihrem Vorbild (wie in dem hier provozierten Beispiel vielleicht suggeriert wurde).
Eine wichtige Anwendung ist dagegen wohl die diskrete Simulation stetiger Systeme, wie sie beispielsweise in den Naturwissenschaften untersucht werden. Zellen repräsentieren daher nicht selten diskrete Punkte im $\mathbb{R}^3$, Zustände enthalten diskretisierte Werte physikalischer Größen oder geben einfach nur Auskunft über die An- oder Abwesenheit eines Teilchens in diesem Punkt, und die Überführungsfunktion soll die Wirkung der Naturgesetze nachahmen.

Da nun aber alle real existierenden Systeme gewissen materiellen Begrenzungen unterliegen (die zu simulierenden Systemen vielleicht weniger offensichtlich als die Simulationssysteme selbst), ist für uns, mehr als der Zellularraum, seine endliche Form von Interesse: der Zellularautomat.

Es sei $M = (S, d, N, \sigma, q_0)$ ein Zellularraum. M heißt ***Zellularautomat****, falls gilt:*

1. *Es gibt $\# \in S$ so daß für alle $t \in \mathbb{N}_0$ und für alle $i \in \mathbb{Z}$ gilt:*

$$c_{t+1}(i) = \# \iff c_t(i) = \#$$

Dabei heißt $\#$ ***Grenzzustand****.*

2. *Sei $G := \{l \in \mathbb{Z}^d \backslash R :$ Es gibt ein $i \in R,$ so daß l Nachbar von i ist $\}$*

Ein endliches, nichtleeres Gebiet $R \subset \mathbb{Z}^d$, ***Retina*** *genannt, hat folgende Eigenschaften:*

- *Für alle* $i \in R$ *gilt:* $c_0(i) \neq \#$
- *Für alle* $k \in G$ *gilt:* $c_0(k) = \#$
- *Für alle* $k \notin G \cup R$ *gilt:* $c_0(k) = q_0$
- *Für alle* $i, j \in R$ *gilt:* i *ist über beliebig viele Stufen hinweg Nachbar von* j *bezüglich* H_1^d*, oder umgekehrt.*

Im Gegensatz zu Zellularräumen ist der interessante Bereich eines Zellularautomaten innerhalb einer fest umrissenen Teilmenge des $\mathbb{Z}^d$ angesiedelt, der Retina: Durch die letzte Bedingung in 2., die besagt, daß neben den Nachbarn einer Zelle auch deren Nachbarn berücksichtigt werden sollen, usw., wird gewährleistet, daß die Retina ein zusammenhängendes Gebiet darstellt. Sie wird begrenzt durch die Menge G in dem Sinne, daß G alle Zellen außerhalb der Retina umfaßt, die die Nachbarn von Zellen innerhalb der Retina sind. Wir nennen G deshalb ***Grenzschicht***. Alle Zellen in der Grenzschicht - und nur diese - befinden sich zu jedem Zeitpunkt in einem ausgezeichneten Zustand $\#$. Der von ihnen ausgehende Einfluß auf die Retina ist also stets derselbe.
Diejenigen Zellen, die sich „jenseits" der Grenzschicht befinden, liegen aufgrund der Konstruktion von G nicht in der Nachbarschaft einer Zelle der Retina, haben demzufolge auch keinen Einfluß auf sie.
Interessiert uns also nur das Verhalten des Zellularraums innerhalb der Retina, so kann uns das Schicksal der Zellularraum-Welt jenseits der Grenzschicht guten Gewissens gleichgültig sein. OBdA befinden sich alle Zellen dort zum Zeitpunkt $t = 0$ im Ruhezustand.

1.2 Zellularautomaten in SCARLET, Beispiele

1.2.1 Allgemeine Eigenschaften SCARLETS

Das Zellularautomaten-Simulationsprogramm SCARLET 0.8 läuft auf allen Rechnern mit Betriebssystem SUN Solaris ab Version 2.0.
Die Steuerung des Programmablaufs und der Simulation erfolgt über eine X11-Oberfläche. Während der Simulation können beliebige 2-dimensionale Ausschnitte der Retina angezeigt werden, die Informationen über Positionen

und Zustände der einzelnen Zellen enthalten.

Zur Beschreibung der zu simulierenden Zellularautomaten dient eine aus zwei Komponenten (RDL und SDL) bestehende Programmiersprache mit C-ähnlicher Syntax. Während durch SDL im wesentlichen σ-Überführungsfunktionen mit korrespondierendem Nachbarschaftsindex charakterisiert werden, können durch RDL Startkonfigurationen beschrieben werden. Beide Vorgänge erfolgen zunächst getrennt voneinander in unterschiedlichen Dateien. Erst zum Zeitpunkt der Simulation werden je ein kompiliertes RDL- und SDL-Programm - sofern sie sich zu einem gültigen Zellularautomat kombinieren lassen - miteinander verbunden.

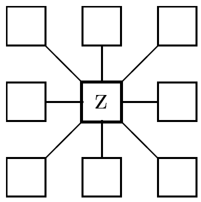
Mit SCARLET wird dem Benutzer nun eine Software zur Verfügung gestellt, die es ihm ermöglicht, beliebige (wie oben beschriebene) Standard-Zellularautomaten mit quaderförmiger Retina zu simulieren.
Darüber hinaus stellt SCARLET dem Benutzer eine Funktion zur Generierung von Zufallszahlen zur Verfügung, die in Verbindung mit dem spezifischen Sprachaufbau SCARLETS ein leistungsfähiges und einfach zu handhabendes Hilfsmittel zur Konstruktion von *probabilistischen* Zellularautomaten darstellt (siehe Bsp. 18, S. 45).
Für andere Verallgemeinerungen über das Standard-Modell hinaus kennt SCARLET keine eigenständigen Sprachstrukturen. In vielen Fällen ist es jedoch möglich, sie unter Zuhilfenahme zusätzlicher Zustände oder Nachbarn auf das Standard-Modell zurückzuführen. SCARLET erleichtert ein solches Vorhaben durch die Verwendung von Registern zur Bildung der Zustandsmenge (vgl. S. 34).
Zeitliche oder *räumliche Inhomogenität* können somit durch Hinzunahme von Registern, in denen Informationen über die Zeit oder die Lage der Zellen (siehe Bsp. 22) gespeichert sind, und eine angepaßte Überführungstafel verwirklicht werden.
Durch Vergrößerung des Nachbarschaftsindexes ist es auf ähnliche Art denkbar, einfache *nichtkonstante Randbedingungen* zu simulieren (siehe Bsp. 25, S. 49).

1.2.2 Programm-Beispiele

Abschließend soll hier noch anhand einiger Beispiele ein erster Eindruck von den Möglichkeiten SCARLETS vermittelt werden. Dabei kann es natürlich nicht darauf ankommen, die vorgestellten Programme bis in Details hinein zu verstehen. Viel wichtiger scheint es zu sein, sie als Prototypen für die folgende Sprachbeschreibung im Hinterkopf zu haben.

Beispiel 1 („Life“)

Wir beginnen mit einem Beispiel, das wohl einen wesentlichen Beitrag dazu geleistet hat, die Theorie der Zellularautomaten ins Bewußtsein der Allgemeinheit zu bringen: dem von J. H. Conway 1970 vorgestellten „Game of Life“. Er untersucht damit die Frage nach der Entwicklung einer Population von Lebewesen unter idealisierten Bedingungen. Aus naheliegenden Gründen können wir „Life“ nicht - wie von Conway gedacht - im Zellularraum betrachten und müssen uns daher mit einer endlichen Retina begnügen. Dazu verwenden wir einen 2-dimensionalen Zellularautomat mit Zuständen 0 und 1. Jede Zelle steht für einen Organismus, der lebendig ist, wenn ihr Zustand 1 ist, und tot falls er 0 ist. Der Nachbarschaftsindex umfaßt, außer der betrachteten Zelle z selbst, die acht räumlich an sie angrenzenden Zellen:



Für die Entwicklung gelten folgende Regeln:

- Haben genau drei der Nachbarzellen den Zustand 1, so erhält die Bezugszelle ebenfalls den Zustand 1.
- Besitzen genau vier der Nachbarzellen Zustand 1, so behält sie ihren alten Status bei.
- In allen anderen Fällen erhält die Bezugszelle den Zustand 0.

Obwohl durch diese Regeln Phänomene wie „Tod durch Über- bzw. Untervölkerung“ oder „Neuentwicklung von Leben unter günstigen Bedingungen“ in denkbar einfacher Weise modelliert werden, legt ein solches System bereits ein nahezu unüberschaubar komplexes Verhalten an den Tag.

```
SIGMA life;
DIMENSION 2;
REGISTER INT state;
NEIGHBOUR center       :( 0, 0);
```

```
          left         :(-1, 0);
          right        :( 1, 0);
          upper        :( 0, 1);
          lower        :( 0,-1);
          left_upper   :(-1, 1);
          right_upper  :( 1, 1);
          left_lower   :(-1,-1);
          right_lower  :( 1,-1);
VAR INT sum;
BORDER {0};

TABLE
sum =center.state
     + left.state + right.state + upper.state + lower.state
     + left_upper.state + right_upper.state
     + left_lower.state + right_lower.state;
sum==3; : SELF={1}; :

sum==4; : /* leere Anweisung */ :

/* TRUE */ : SELF={0}; :
```

Das angegebene SDL-Programm (als solches erkennbar durch das Schlüsselwort `SIGMA`) beschreibt nun dieses Verhalten während der Simulation für eine einzelne Zelle der Retina. Das Programm besteht aus zwei Teilen: Der erste Teil erstreckt sich bis `TABLE` und enthält verschiedene Vereinbarungen:

- Die Raumdimension wird auf `2` gesetzt.

- Die Repräsentation der Zustandsmenge erfolgt in SCARLET durch deren Aufteilung in kleinere Einheiten, die Registern. Wir unterscheiden zwei Typen von Registern, deren Wertebereich etwa der von Integer- bzw. Stringvariablen in C ist. Zur Darstellung der Zustände `0` und `1` wird hier ein numerisches Register mit Namen `state` verwandt.

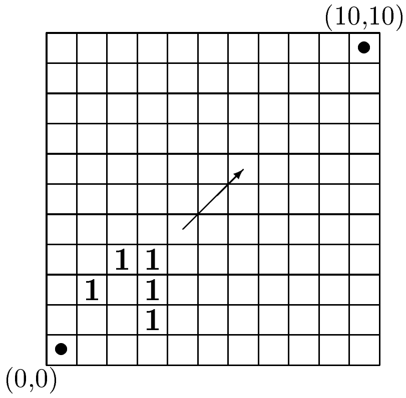
- Die Nachbarn werden durch Angabe ihrer relativen Positionen festgelegt. Auch sie können ab sofort durch die vergebenen Namen angesprochen werden.

- Eine numerische Variable mit Namen `sum` wird deklariert,

- und durch `BORDER {0};` wird schließlich vereinbart, daß sich alle Grenzzellen im Prinzip wie Retina-Zellen mit Zustand `0` verhalten sollen (was hinsichtlich der damit verbundenen Interpretation sicher eine sinnvolle Modell-Annahme ist).

Der zweite Teil formuliert die Entwicklungsregeln als eine Art (hier dreifach) geschachtelter IF-THEN-ELSE-Struktur (Dabei werden die IF- und THEN-, bzw. THEN- und ELSE-Teile jeweils durch : voneinander separiert.) Variablenzuweisungen sind hier jederzeit möglich - in diesem Fall wird in `sum` zunächst die Gesamtzahl der Nachbarn mit Zustand `1` ermittelt und in Abhängigkeit dieses Wertes zu verschiedenen Aktionen verzweigt.
Die leere Bedingung im dritten Eintrag ist stets erfüllt, und die leere Anweisung ändert den Zustand der betrachteten Zelle nicht.

```
RETINA glider;
RANGE [0,0]..[10,10];
REGISTER INT state;
START
  [1,2] = {1};
  [2,3].state = 1;
  [3,1]..[3,3] = {1};
```



Dieses RDL-Programm beschreibt eine zum SDL-Programm `Life` passende Startkonfiguration. Auch hier findet sich wieder eine Unterteilung in einen Deklarations- und einen Anweisungsteil:

- Die in Scarlet immer rechteckige Retina im $\mathbb{Z}^2$ besitzt die Eckkoordinaten $(0,0)$, $(0,10)$, $(10,0)$ und $(10,10)$, von denen nur zwei sich gegenüberliegende angegeben werden müssen.

- Korrespondierend zu dem numerischen Register in `SIGMA Life` muß auch hier ein solches verwendet werden, (das zufälligerweise denselben Namen hat).

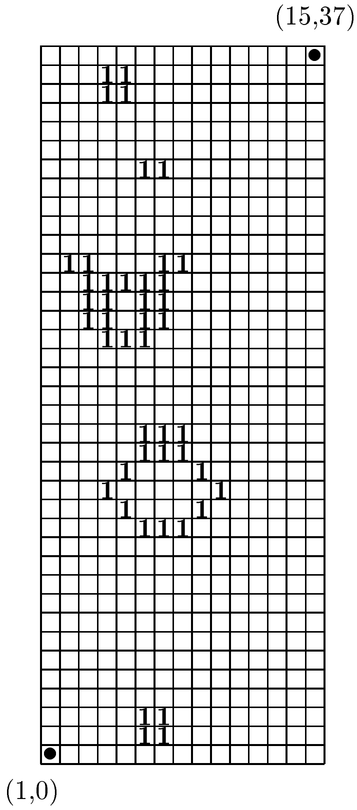
- `START` zeigt den Beginn des Anweisungsteils an, in dem diversen Zellen (angesprochen durch deren Koordinaten) der Zustand `1` zugewiesen wird. Dabei ist es sowohl möglich, den Zustand als ganzen in {} anzugeben (1. und 3. Zeile), als auch die Register einzeln zu belegen (2. Zeile). Durch die 3. Zeile werden alle Zellen im Rechteck zwischen den angegebenen Eckkoordinaten gleichzeitig behandelt.

Alle übrigen Zellen befinden sich infolge der automatischen Retina-Initialisierung bereits im Zustand `0`.

Die hier vorgestellte Figur aus Zustand-1-Zellen wird unter `Life` nach 5 Simulationsschritten wieder in sich selbst überführt. Ihre Lage hat sich dann allerdings um je eine Position nach oben und nach rechts verschoben.

Übrigens: Die folgende, etwas skurrile Startkonfiguration beantwortete seinerzeit die von Conway aufgeworfene Frage, ob es unter den Regeln von „Life“ Populationen geben kann, deren Größe unbeschränkt anwächst, sofern genügend Platz vorhanden ist. (Kleiner Tip: Die Lösung hat mit der ersten Figur zutun.)

```
RETINA strange;
RANGE [-10,-10]..[30,40];
REGISTER INT state;
START
  [6,1]..[7,2] = {1};

  [6,12]..[8,12] = {1};
  [5,13] (4) [9,13] = {1};
  [4,14] (6) [10,14] = {1};
  [5,15] (4) [9,15] = {1};
  [6,16]..[8,17] = {1};

  [4,22]..[6,22] = {1};
  [3,23]..[7,25] = {1};
  [5,23]..[5,24] = {0};
  [2,26] (6) [8,26] = {1};
  [3,26] (4) [7,26] = {1};

  [6,31]..[7,31] = {1};

  [4,35]..[5,36] = {1};
```



Anmerkung:
Der Befehl `[5,15] (4) [9,15] = {1};` weist innerhalb des Rechtecks `[5,15] .. [9,15]` jeder vierten Zelle in jeder Koordinatenrichtung beginnend bei der ersten Ecke den Zustand `1` zu.

Beispiel 2 („FSSP")

Das zweite berühmte (und schon etwas komplexere) Beispiel stammt noch aus der Anfangszeit der Zellularautomaten-Ära. Es behandelt die Frage, ob und wie es möglich sei, eine Reihe von Automaten zu synchronisieren, wenn jeder von ihnen nur seinen unmittelbaren linken und rechten Nachbarn in der Reihe wahrnehmen kann, und der Befehl zur Synchronisation von einem der beiden Randautomaten ausgeht. (Dieselbe Frage wurde oftmals scherzhaft - statt für Automaten - für einen (stimmschwachen) General und eine Kompanie (tuschelnder) Soldaten formuliert, daher der Name **f**iring **s**quad **s**ynchronization **p**roblem.)

Die hier vorgestellte Methode geht auf die 1966 von A. Waksman erdachte zeitoptimale Lösung zurück (vgl. auch [5]). Die Idee besteht in der fortwährenden Halbierung der Retina und der gleichzeitigen Erzeugung weiterer Generäle, die die entstandenen Hälften wiederum halbieren usw. bis schließlich alle Zellen Generäle sind und daran erkennen, daß sie im nächsten Schritt den Schuß abzufeuern haben. Die Halbierung erfolgt durch das Aufeinandertreffen verschieden schneller Signale, in deren Erzeugung und Koordinierung nun die eigentliche Aufgabe liegt.

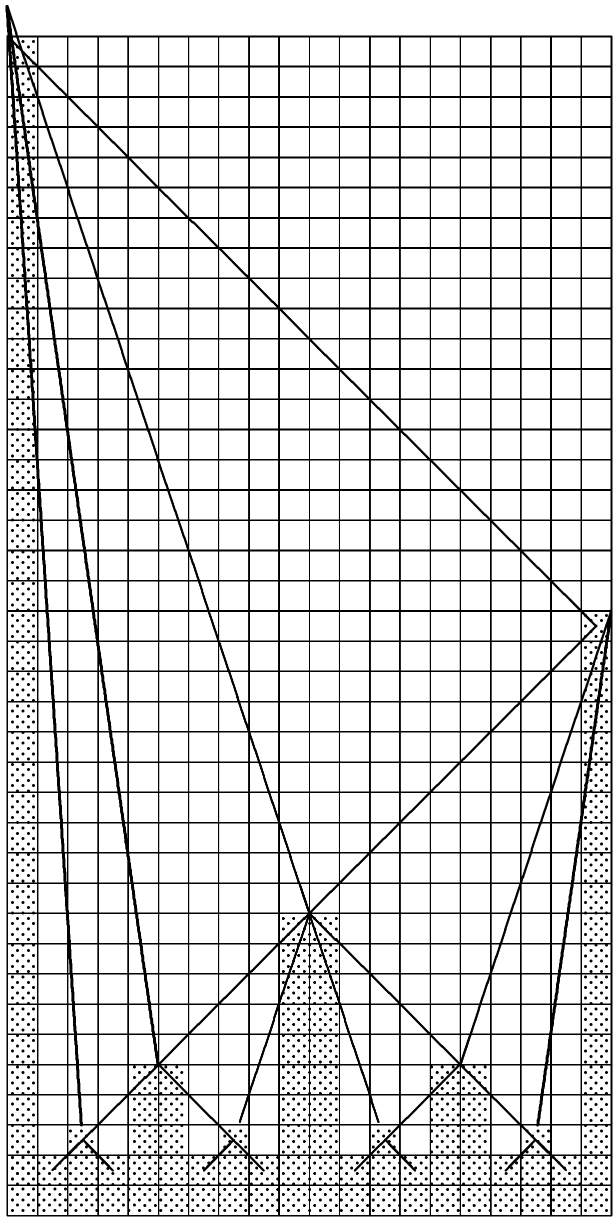
Zunächst sendet der General am Rand ein Hauptsignal aus, das sich mit Geschwindigkeit 1 (= eine Zelle pro Zeitschritt) auf den gegenüberliegenden Rand zubewegt, dort reflektiert wird und gleichzeitig einen zweiten General erzeugt.

Weitere Hilfssignale sollen von dem Ausgangsgeneral so entsandt werden, daß sie das erste Signal auf seinem Weg zurück jeweils bei seinem Eintritt in die erste Hälfte, (das erste Viertel, das erste Achtel, ...) der Retina treffen.

Dort erzeugen sie, je nach dem, ob sie eine Strecke ungeradzahliger oder geradzahliger Länge halbiert haben, einen oder zwei weitere Generäle. (Im ersten Fall nennen wir das Hauptsignal, das die Information darüber durch Abzählen erbracht hat und nun in sich birgt, 1G-Signal, im zweiten Fall 2G-Signal.)

Die neu entstandenen Generäle sind nun rekursiv für ihre Hälften zuständig.

Bsp.4: Betrachten wir die Evolution einer 1-dimensionalen Retina mit 20 Feldern, so ergibt sich etwa folgendes (gewünschtes) Verhalten: (Jede Kästchenzeile steht dabei für das Aussehen der Retina in einem gewissen Zeitschritt. Die schattierten Felder stellen Generäle dar.)



Das obige Beispiel wirft folgende Fragen auf:

1. Wie erzeugt man am besten Hilfssignale mit den oben genannten Eigenschaften?

2. Wie stimmt man Haupt- und Hilfssignale aufeinander ab, wo doch die Reflexion der Hauptsignale an einem General schneller erfolgt als an zweien (Zeitoptimalität!)?

Wird das Hauptsignal an zwei Generälen reflektiert, so macht das n-te Hilfssignal einen Schritt, während das Hauptsignal in derselben Zeit $2^{n+1} - 1$ Felder bewältigen muß ($1 : (2+1)$, $1 : (4+3)$, $1 : (8+7)$, ...). Weiterhin muß das Hilfssignal die Mitte vor oder gleichzeitig mit dem Hauptsignal erreichen. Aus solchen Vorüberlegungen ergibt sich z.B. die folgende Methode von Waksman:

Die Hauptsignale senden alle zwei Zeitschritte ein Steuersignal in die umgekehrte Richtung aus, das ein von dem General ständig erneuertes konstantes Signal beim Zusammentreffen um ein Feld in Richtung des Hauptsignals weiterbewegt.

Ein konstantes Signal läßt jedes zweite Steuersignal passieren, so daß es nachfolgende konstante Signale beeinflussen kann.

Die Antwort auf die zweite Frage wurde im Prinzip schon durch das Bild gegeben: Ist nur ein General vorhanden (durchquert das Hauptsignal also ein Feld weniger), so gaukelt man vor, das Hilfssignal sei einen Zeitschritt früher ausgesandt worden, oder mit anderen Worten: Die Steuersignale werden von 1G-Signalen einen Zeitschritt früher ausgeschickt als von 2G-Signalen.

Bsp.4: (Fortsetzung)
Die nachstehende Abbildung zeigt bereits die Zustands-Entwicklung der Retina aus dem vorherigen Beispiel für eine konkrete Realisierung mit 10 Zuständen.

Wir verwenden verschiedene Generäle `"G"` oder `"g"` je nach dem, ob ein 1G- oder ein 2G-Signal auf den Weg gebracht werden soll. Soldaten besitzen den Zustand `""`, Steuersignale `">"` und `"<"`, die konstanten Signale wechseln zwischen dem „durchlässigen“ Zustand `"1"` und dem „blockierenden“ Zustand `"2"` hin und her. Die Hauptsignale erhalten die Zustände `"L"` bzw. `"R"`. Ihre „Zweizeiligkeit“ wird hier benötigt, um sie als 1G- oder 2G-Signale identifizieren zu können. Gleichzeitig läßt sich anhand der Hilfszeile feststellen, ob die zurückgelegte Strecke zwischen zwei Generälen geradzahlig war, oder nicht. `"f"` schließlich bezeichnet den Feuerzustand.

G																			
G	R																		
G	1	R																	
G	1		R																
G	1	<	R	R															
G	<	2			R														
G	1	2		<	R	R													
G	1	2	<				R												
G	1		1			<	R	R											
G	1		1		<				R										
G	1		1	<				<	R	R									
G	1		<	2			<				R								
G	1	<		2		<				<	R	R							
G	<	2		2	<				<				R						
G	1	2			1			<				<	R	R					
G	1	2			1		<				<				R				
G	1	2			1	<				<				<	R	R			
G	1	2			<	2			<				<				R		
G	1	2		<		2		<				<				<	R	R	
G	1	2	<			2	<				<				<				g
G	1		1				1			<				<				L	g
G	1		1				1		<				<				L	>	g
G	1		1				1	<				<				L	<	1	g
G	1		1				<	2			<				L	>		1	g
G	1		1			<		2		<				L	<		>	1	g
G	1		1		<			2	<				L	>			2	>	g
G	1		1	<					1			L	<		>		2	1	g
G	1		<	2					1		L	>				>	2	1	g
G	1	<		2					1	L	<		>			1		1	g
G	<	2		2					g	g				>		1		1	g
G	1	2		2				L	g	g	R				>	1		1	g
G	1	2		2			L	>	g	g	<	R			2	>		1	g
G	1	2		2		L	<	1	g	g	1	>	R		2		>	1	g
G	1	2		2	L	>		1	g	g	1		<	R	2		2	>	g
G	1	2		g	G		>	1	g	g	1	<		G	g		2	1	g
G	1	2	L	G	G	R	2	>	g	g	<	2	L	G	G	R	2	1	g
G	1	g	1	G	G	1	g	1	g	g	1	g	1	G	G	1	g	1	g
G	G	g	G	G	G	G	g	G	g	g	G	g	G	G	G	G	g	G	g
f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f

Im folgenden stellen wir das komplette, zu der oben illustrierten Lösung zugehörige SDL-Programm vor. Trotz seines größeren Umfangs ist es frei von jeglicher programmiertechnischer Akrobatik!
Die verschiedenen Zustände wurden in einem Stringregister `state` abgelegt. Der besseren Lesbarkeit wegen wurden die Registerinhalte der drei Nachbarn zunächst in Variablen geschrieben.

Ein typischer Eintrag der Überführungstafel bestimmt dann für eine spezielle

Nachbarkonfiguration (z.B. `l=="G"; c=="g"; r=="G";`) oder auch mehrere (z.B. `c=="g"; l=="G";`) einen neuen Zustand für die Bezugszelle. In Fällen, die nicht explizit durch die Überführungstafel erfaßt werden, verändert sich der Zustand der Bezugszelle nicht. Wir verwenden hier nicht die bereits vom Beispiel „Life“ her vertraute `SELF`-Schreibweise, die einen Zellausdruck erfordert, sondern belegen direkt das Register neu; hierfür genügt die Angabe des betreffenden Registernamens, also `state`.

Bei größren Überführungstafeln wie dieser muß besonders sorgfältig auf die richtige Reihenfolge der Einträge geachtet werden. Spezialfälle müssen unbedingt vor solchen allgemeineren Charakters aufgeführt werden, sofern diese die ersteren umfassen, da die Spezialfälle sonst infolge der IF-THEN-ELSE-Struktur der Überführungstafel ohne Wirkung bleiben.

```
SIGMA fssp;
DIMENSION 1;
REGISTER STRING state;
NEIGHBOUR center       : (0);
          left         : (-1);
          right        : (1);
VAR STRING c, l, r;

TABLE
/* Variablenbelegung */
     /* (wird als Teil der ersten Bedingung immer durchgeführt) */
l=left.state; c=center.state; r=right.state;
/* feuern */
l=="g"|| l=="G"; c=="g"|| c=="G"; r=="g"|| r=="G"; :
     state="f"; :

/* vorletzter Schritt */
l=="g"|| l=="G"; r=="g"|| r=="G"; :
     state="G"; :

/* Entstehung eines 1G-Signals (Generalumwandlung) */
     /* durch 2G-Signal (bei zwei Generälen) */
c=="g"; l=="G"|| r=="G"; : state="G"; : /* nicht 3 Generäle */
     /* durch 1G-Signal (bei einem General) */
l==""; c=="g"; r==""; : state="g"; :
l==""; c=="g"; r=="L"; : state="G"; :
l=="R"; c=="g"; r==""; : state="G"; :

/* General-Erzeugung auf der Hälfte des Weges */
```

```
     /* 1. General immer */
c=="1"|| c=="2"; l=="R"|| r=="L"; : state="g"; :
     /* 2. General nur bei Ankunft eines 2G-Signals */
l=="1"|| l=="2"; c=="L"; r==">"; : state="G"; : /* 1G-Signal */
l=="<"; c=="R"; r=="1"|| r=="2"; : state="G"; :
l=="1"|| l=="2"; c=="L"; r=="<"; : state="g"; : /* 2G-Signal */
l==">"; c=="R"; r=="1"|| r=="2"; : state="g"; :
l=="1"|| l=="2"; c=="L"; r=="g"; : state="g"; : /* Länge==4 */
l=="g"; c=="R"; r=="1"|| r=="2"; : state="g"; :
     /* sonst: Informations-Weitergabe an 1. General */
l=="1"|| l=="2"; c=="L"; r=="L"; : state=""; : /* 1G-Signal */
l=="R"; c=="R"; r=="1"|| r=="2"; : state=""; :
l=="1"|| l=="2"; c=="L"; r==""; : state="L"; : /* 2G-Signal */
l==""; c=="R"; r=="1"|| r=="2"; : state="R"; :
l=="1"|| l=="2"; c=="L"; r=="G"; : state="1"; : /* Länge==5 */
l=="G"; c=="R"; r=="1"|| r=="2"; : state="1"; :

/* General-Erzeugung am Rand */
l=="R"; c==""; isborder(r); : state="g"; :
isborder(l); c==""; r=="L"; : state="g"; :
/* Umwandlung des Randgenerals für 1G-Signal */
l=="R"; c=="g"; isborder(r); : state="G"; :
isborder(l); c=="g"; r=="L"; : state="G"; :

/* Steuerung von 1G- und 2G-Signalen */
     /* Entstehung, Reflexion am Rand */
l=="g"; (c=="2"||c=="L"||c==""); : state="R"; :
(c=="2"||c=="R"||c==""); r=="g"; : state="L"; :
     /* Bewegung */
l=="R"; c==""; : state="R"; : /* isborder(r)==FALSE ! */
c==""; r=="L"; : state="L"; :

/* Speziell: Steuerung der 1G-Signale */
l=="G"; c==""; : state="R"; : /* Erzg.: 1G-Signal */
c==""; r=="G"; : state="L"; :
l=="G"; c=="R"; r==""; : state="1"; : /* Erzg.: 1. konst. Signal */
l==""; c=="L"; r=="G"; : state="1"; :
l=="1"|| l=="R"; c=="R"; r==""; : state=""; : /* Bewegung 1G */
l==""; c=="L"; r=="1"|| r=="L"; : state=""; :
c==""; r=="R"; : state="<"; : /* Erzg.: Steuersignal */
l=="L"; c==""; : state=">"; :
l=="<"; c=="R"; r=="R"; : state=""; :/* 1G-Erkennung */
l=="L"; c=="L"; r==">"; : state=""; :
```

```
l==""; c=="R"|| c=="L"; r==""; : state=c; :

/* Speziell: Steuerung 2G-Signale */
l=="g"; c=="R"; r==""; : state="<"; : /* Erzg.: 1.Steuersignal */
l==""; c=="L"; r=="g"; : state=">"; :
l==">"; c=="R"; r==""; : state="<"; : /* Erzg.: Steuersignal sonst */
l==""; c=="L"; r=="<"; : state=">"; :
l=="<"; c=="R"; r==""; : state=">"; : /* 2G-Erkennung */
l==""; c=="L"; r==">"; : state="<"; :

/* Steuersignale */
l=="G"|| l=="g"; c=="<"; : state="1"; : /* Erzg. konst. Signale */
c==">"; r=="G"|| r=="g"; : state="1"; :
l=="1"&&c=="<"|| c==">"&&r=="1"; : state="2"; : /* Shifts */
l=="2"&&c=="<"|| c==">"&&r=="2"; : state="1"; :
c=="1"; r=="<"; : state="<"; : /* durchdringt 1-Signale */
l==">"; c=="1"; : state=">"; :
c=="2"; r=="<"; : state=""; : /* gestoppt von 2-Signalen */
l==">"; c=="2"; : state=""; :
c=="2"; l==">"|| r=="<"; : state=""; :
l==">"; c==""; : state=">"; : /* Bewegung */
c==""; r=="<"; : state="<"; :
c==">"|| c=="<"; : state=""; : /* trifft nicht auf 1,2,g,G */
```

Bsp.4: (Fortsetzung)
Ein RDL-Programm, das die Startkonfiguration der Retina mit 20 Zellen (19 Soldaten und ein General) realisiert, würde dagegen schlicht so aussehen:

```
RETINA fssp;
RANGE [0]..[19];
REGISTER STRING state;
START
  [0].state = "G";
  [1]..[19].state = "";
```

Kapitel 2

Allgemeine Grundlagen von RDL und SDL

In den nächsten drei Kapiteln sollen nun endlich die beiden entscheidenden Hilfsmittel zur Konstruktion eines Programms, das einen vorgegebenen Zellularautomaten simuliert, vorgestellt werden. Wir vergegenwärtigen uns kurz, wodurch ein Zellularautomat im wesentlichen charakterisiert wird:

- Retina (endlicher Ausschnitt des $\mathbb{Z}^n$)
- endliche Zustandsmenge
- Startkonfiguration
- Nachbarschaftsindex
- Überführungsfunktion σ

Aus verschiedenen Gründen werden in SCARLET Retina und Startkonfiguration sowie Nachbarschaftsindex und Überführungsfunktion zunächst getrennt voneinander definiert. Dieses Vorgehen erlaubt es später z.B., verschiedene „zueinander passende“ Teile in gewünschter Art und Weise zu einem fertigen Automaten zu kombinieren. Die diffuse Umschreibung „zueinander passend“ bezieht sich dabei zum einen auf die Dimension der Retina, zum anderen auf die Zustandsmenge, die als Schnittstelle zwischen Retina und Überführungsfunktion ebenfalls in beiden Definitionen identisch sein muß.
Wir gehen später noch genauer darauf ein.

Die Beschreibung erfolgt durch die speziell hierfür vorgesehenen Programmiersprachen RDL (= **R**etina **D**efinition **L**anguage) und SDL (= **S**igma **D**efinition **L**anguage), wobei unschwer zu erraten sein dürfte, welche Sprache

welchem Teilproblem zugeordnet ist?!
Vielleicht werden Sie nun erschrocken einwenden: „Was, gleich *zwei Sprachen* soll ich lernen, nur um einen Zellularautomaten produzieren zu können??“ Es sei jedoch an dieser Stelle versichert: So verschieden, wie RDL und SDL im ersten Moment erscheinen mögen, sind sie glücklicherweise nicht: Es existieren - im Gegenteil - eine Reihe von universal gültigen Grundkonzepten, die deshalb im ersten Abschnitt über RDL und SDL gesondert beschrieben werden sollen. Wir wollen sogar so weit gehen, in Zukunft einfach von „SCARLET“ zu sprechen, wenn wir RDL *und* SDL meinen.
Dem mit C vertrauten Leser wird darüber hinaus bald auffallen, daß einige Konzepte in SCARLET den dortigen ähnlich oder sogar mit ihnen identisch sind. Sie sollen hier der Vollständigkeit halber aber trotzdem (mehr oder weniger ausführlich) erläutert werden.
Wer jetzt immer noch etwas skeptisch ist, möge sich damit trösten, daß der Sprachumfang von SCARLET aufgrund der speziellen Zielsetzung längst nicht so umfangreich ist, wie der „großer“ Programmiersprachen.

2.1 Absprachen

Wir halten uns bei der Beschreibung der Sprachkomponenten an die in der Grammatik verwandten Bezeichnungen für Nichtterminalsymbole (oder erwähnen diese zumindest, wo es nötig erscheint). Wir benutzen sie jedoch auch, wenn wir die gesamte Objektmenge meinen und nicht nur ein einzelnes Element daraus.
Findet sich der Name einer Nichtterminalsymbol-Klasse in der Überschrift eines Abschnitts, so handelt es sich bei den dort besprochenen Objekten meist ausschließlich um Elemente dieser Klasse.

Var, Expression, Constant-Identifier

Manchmal betrachten wir der Einfachheit halber auch allgemeinere Klassen, als durch die Grammatik vorgegeben:
So bezeichnen wir mit *Expression* die Menge aller Ausdrücke, d.h. die Vereinigung der Mengen *IntExpression*, *StringExpression*, *CellExpression* und *BoolExpression*.
In analoger Weise erhalten wir die Mengen *Var* und *ConstantIdentifier*.

Listen

Ein einzelnes oder mehrere durch Kommata getrennt voneinander stehende Objekte einer beliebigen Klasse *Class* bezeichnen wir als *ClassList*.

Syntaxdiagramme

Besteht die Syntax einer Klasse aus mehr als einem Terminal- oder Nichterminalsymbol, so geben wir deren Aufbau in Form eines Syntaxgraphen

an: Terminale Symbole stehen dabei in ovalen, nichtterminale in eckigen Kästchen. (Zusätzlich erscheinen - auch innerhalb des Textes - Terminalsymbole in Schreibmaschinenschrift, Nichtterminalsymbole kursiv gedruckt.)
Ein korrekter Durchlauf durch den Graph erfolgt in Pfeilrichtung, beginnend am alleinstehenden Pfeilende und endend an der einzelnen Pfeilspitze.
Syntaktisch korrekte Elemente der beschriebenen Klasse entstehen durch Aneinanderreihung der im Graphen enthaltenenen Terminal- und Nichtterminalsymbole während eines korrekten Graphendurchlaufs. Die einzelnen Symbole sollten dabei i.a. durch Leerzeichen voneinander separiert werden (vgl. S. 29).

Wenn wir im Rahmen einer Syntaxbeschreibung davon sprechen, daß ein Objekt „beliebig“ oft auftreten kann, so meinen wir selbstverständlich, daß die Anzahl lediglich durch die Größe des zur Verfügung stehenden Speichers begrenzt wird.

2.2 Quelltextkomponenten

2.2.1 Zeichensatz

Der einem gültigen RDL- oder SDL-Programmtext zugrunde liegende Zeichensatz gliedert sich folgendermaßen auf:

Groß- und Kleinbuchstaben (*Letter*) : `a`, `b`, .. , `z`, `A`, `B`, .. , `Z`
(ohne Umlaute und ß)
Ziffern (*Digit*) : `0`, `1`, .. , `9`
Sonderzeichen : `+`, `-`, `*`, `%` , `/`, `&` , `|`, `=`, `>`, `<`, `!`, `(`, `)`, `[`, `]`, { , } , `.`, `:`, `,`, `;`,
`_` , `'`,
`ä`, `ö`, `ü`, `Ä`, `Ö`, `Ü`, `^`, `°`, `"`, §, `$`, `?`, `\`, `` ` ``, `´`, `@`, `~`, `#`, ␣
(nur in Strings)

Trennzeichen: ␣, TAB, newline

Alle Groß- und Kleinbuchstaben, Ziffern und Sonderzeichen zusammen stellen die Menge der Zeichen (*Character*) dar.

Trennzeichen sind überall dort unverzichtbar, wo Bezeichner oder Schlüsselworte direkt aufeinander treffen, ansonsten können sie auch entfallen. Welches Trennzeichen im einzelnen benutzt wird, spielt im allgemeinen keine

Trennzeichen

Rolle. Allerdings empfiehlt sich aus Gründen der Übersichtlichkeit ein Programmaufbau in Zeilen oder Abschnitten. Besonders wichtig ist es, beim Anlegen der Überführungsfunktionstafel darauf zu achten, daß einzelne Einträge als solche erkennbar voneinander abgesetzt werden (siehe S. 96). Auch der Einsatz von Leerzeichen sollte so erfolgen, daß er der Lesbarkeit des Programmtextes zuträglich ist. Es ist allerdings untersagt, Schlüsselwörter, Bezeichner sowie zusammengesetzte Operatoren (wie z.B. `<=`, `++`, ...) durch Trennzeichen auseinanderzureißen: Sie werden sonst nicht mehr als solche erkannt.
Weiterhin dürfen bei der Aufteilung eines Stringausdrucks auf mehrere Zeilen keine Trennzeichen zwischen dem Fluchtsymbol `\` und dem Zeilenumbruchszeichen stehen, vgl. S. 34.

2.2.2 Schlüsselworte

Folgende Worte sind in SCARLET reserviert:

RETINA	SIGMA	TRUE	scopy
RANGE	DIMENSION	FALSE	itos
REGISTER	REGISTER	INT	abs
CONST	NEIGHBOUR	STRING	sgn
VAR	CONST	BOOL	pow
START	VAR	CELL	rand
	TABLE		stoi
	SELF		len
	VOID		instr
	BORDER		coord
			isborder

Diese sollten sie unter keinen Umständen als Bezeichner z.B. für Variablen, Konstanten, Register oder Nachbarn verwenden. Beachten Sie auch, daß die Groß- und Kleinschreibung hierbei irrelevant ist.

2.2.3 Kommentare

Unabhängig von der restlichen Grammatik SCARLETS ist es an (fast) jeder Stelle gestattet, Kommentare in den laufenden Programmtext einzufügen. Dies geschieht (wie in C) durch

`/*` *Kommentartext* `*/`

Dabei werden alle Zeichen - ausgenommen die Kombinationen `/*` und `*/` selbst - ab Kommentarbeginn bis zum zugehörigen Kommentarende durch den Compiler ignoriert; Die Schachtelung von Kommentaren ineinander ist daher möglich. - Umgekehrt führt die Verwendung von `*/` ohne vorangehendes `/*` in derselben Schachtelungsebene zu Fehlern. Da insbesondere auch Trennzeichen „übersehen“ werden, können sich Kommentare auch über mehrere Zeilen erstrecken.

Bemerkung:
Kommentare können nicht innerhalb von Stringkonstanten plaziert werden, da die Stringbegrenzungszeichen die Wirkung von `/* */` aufheben. **Strings**

2.3 Bezeichner *Identifier*

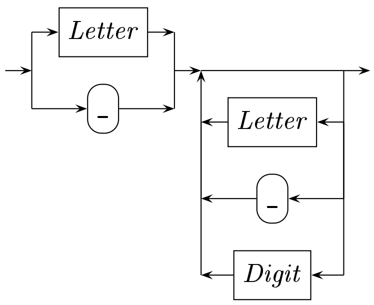
Immer, wenn es darum geht, einem neu geschaffenen Objekt einen Namen zu geben, wird dazu ein Bezeichner benötigt. Zu den Objekten, die es in SCARLET zu benamen gilt, gehören selbstverständlich Variablen und Konstanten, speziell in SCARLET aber auch Register und Nachbarn; und letztlich können auch Namen für die Retina und die Überführungsfunktion selbst vergeben werden.

Unter anderem um deutlich zu machen, für welches Oblekt genau ein Bezeichner benötigt wird, werden diese in vielen Syntaxdiagrammen nicht schlicht *Identifier* genannt, sondern je nach Bestimmung suggestiver *IntConstIdentifier*, *StringRegIdentifier*, usw. . Lassen Sie sich dadurch nicht irritieren. Einen Überblick gibt die nachfolgende Zusammenstellung:

Bezeichner für		**Nichtterminalsymbol**
Konstanten:	Ganzzahl-	*IntIdentifier*, *IntConstantIdentifier*
	String-	*StringIdentifier*, *StringConstantIdentifier*
	boolesche	*BoolIdentifier*, *BoolConstantIdentifier*
	Zell-	*CellIdentifier*, *CellConstantIdentifier*
Variablen:	Ganzzahl-	*IntVar, Identifier*
	String-	*StringVar, Identifier*
	boolesche	*BoolVar, Identifier*
	Zell-	*CellVar, Identifier*
Register:	Ganzzahl-	*IntRegIdentifier, Identifier*
	String-	*StringRegIdentifier, Identifier*
Nachbarn:		*NeighbourIdentifier*
Retinas:		*Identifier*
Überführungsfunktionen:		*Identifier*

Achten Sie stets darauf, daß die Benennung innerhalb eines Programms in eindeutiger Weise erfolgt.

Syntax:



Gültige Bezeichner sind Worte mit einer maximalen Länge von 32 Zeichen, gebildet aus Klein- und Großbuchstaben (ohne Umlaute und ß) (*Letter*), Unterstrichen und Ziffern (*Digit*), wobei die Verwendung von Ziffern am Wortbeginn untersagt ist.

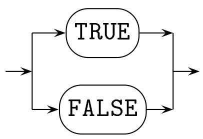
Auch hier gilt: Der Compiler differenziert nicht zwischen Groß- und Kleinschreibung.

Bsp.5:	√ `counter`	(richtig)
	√ `_control`	(richtig)
	× `1th_variable`	(falsch; Ziffer zu Beginn)
	× `left-neighbour`	(falsch; `-` enthalten)
	× `var`	(falsch; ist Schlüsselwort)
	√ `var1`	(richtig)
	√ `Counter`	(richtig; ist dasselbe wie `counter`)
	× `is_wrong?`	(falsch; `?` enthalten)
	× `ist_größer`	(falsch; ß und Umlaut enthalten)

2.4 Konstanten

2.4.1 Boolesche Konstanten *BoolConstant*

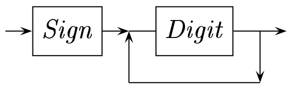
Syntax:



Die booleschen Werte werden in SCARLET repräsentiert durch die Worte `TRUE` für WAHR sowie `FALSE` für FALSCH. Boolesche Werte sind vom Typ `BOOL`.

2.4.2 Ganzzahlkonstanten *IntConstant*

Syntax:

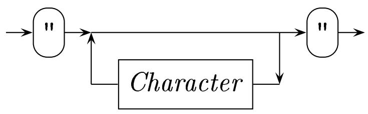


Ganzzahlkonstanten repräsentieren die ganzen Zahlen $\mathbb{Z}$ im Bereich von $-2.147.483.648$ bis $+2.147.483.647$. (Das entspricht einer internen Darstellung als 32-Bit Wort, wobei das Vorzeichen ein Bit belegt.) Die Angabe erfolgt - wie gewohnt - im Dezimalformat durch eine entsprechende Ziffernfolge (*Digit*), der optional ein `+` oder `-` (*Sign*) mit der üblichen Bedeutung als Vorzeichen vorangesetzt werden kann. Eine Zahlkonstante ohne Vorzeichen wird als positive Zahl interpretiert. Zahlkonstanten sind vom Typ `INT`.

Bemerkung:
Zahlen dienen im wesentlichen nur zur Bezeichnung von Zuständen und zur Angabe von absoluten und relativen Positionen der Zellen innerhalb der Retina. Es werden daher keine rationalen Zahlen benötigt.

2.4.3 Stringkonstanten *StringConstant*

Syntax:



Eine Stringkonstante ist eine Zeichenkette, bestehend aus höchstens 255 Zeichen der Klasse *Character*. Um Stringkonstanten beispielsweise von Bezeichnern abzuheben, werden jene in doppelte Anführungszeichen `""` eingeschlossen.
Stringkonstanten sind vom Typ `STRING`.

Lange Strings

Um auch längere Stringkonstanten ganz auf dem Bildschirm anzeigen zu können, besteht die Möglichkeit, deren Darstellung in der nächsten Zeile fortzusetzen. In diesem Fall muß der String am gewünschten Zeilenende mit \ abgeschlossen werden; der Rest des Strings muß sich unmittelbar am Anfang der nächsten Zeile anschließen.

Beachte: Zwischen \ und dem Zeilenumbruchszeichen dürfen sich keine weiteren Zeichen befinden - auch keine Leerzeichen!
Leerzeichen zwischen Zeilenbeginn und Stringfortsetzung werden als zum String dazugehörig gewertet.

Bsp.6:
```
"Dieser String ist in der ersten Zeile nicht zuende, \
sondern geht in der zweiten weiter."
```

2.4.4 Zellkonstanten

[1]

[1] Zellkonstanten bilden in SCARLET (aus mehr oder weniger formalen Gründen) keinen eigenständigen Grammatikteil; es ist vielmehr ausreichend, sie als Teilmenge aller zellwertigen Objekte *CellExpression* (und dort genauer als Teilmenge von {*RegExpressionList*})

Für die Konstruktion einer gut handhabbaren endlichen Zustandsmenge für den Zellularautomaten ist das Konzept der Register (wie eingangs schon erwähnt) von großer Bedeutung: Statt der Verwaltung eventuell unübersichtlicher komplexer Zustände in einer einzigen „Zustandsvariablen“ bietet SCARLET die Möglichkeit der Aufspaltung in kleinere Einheiten, den sogenannten Registern. Wir unterscheiden solche Register, die vom Typ `INT` sind, also numerische Werte aufnehmen, und analog dazu solche vom Typ `STRING`. (Wir bezeichnen sie im folgenden kurz mit Int- bzw. String-Register.) Anzahl und Reihenfolge der Register (sowie deren Bezeichnungen) werden zu Beginn eines jeden Programms für alle Zellen des Automaten einheitlich festgelegt. (Mehr dazu im Abschnitt 'Register-Deklaration', S. 70). Der Gesamtzustand einer Zelle ergibt sich dann als Kreuzprodukt über das (geordnete) Tupel ihrer verschiedenen Registerinhalte. Da jedes Register nur eine endliche Zahl verschiedener Werte annehmen kann, handelt es sich also auch bei deren Kreuzprodukt um eine endliche Menge.

Bsp.7: Wir betrachten die Zustandsmenge

$$S = \{z_0, \cdots, z_n\}\,, \quad n \in I\!N$$

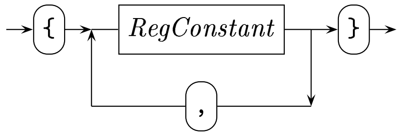
Sie kann mit Hilfe eines String- und eines Int-Registers auf folgende Art dargestellt werden:

$$\{\,\{„z“, 0\}\,, \{„z“, 1\}\,, \cdots, \{„z“, n\}\}\ ,$$

So muß für einen Zustandswechsel lediglich der Inhalt des Int-Registers durch eine geeignete arithmetische Operation auf den gewünschten Wert transformiert werden.

Zellkonstanten müssen selbstverständlich die vorgegebene Registerstruktur widerspiegeln: Die verschiedenen (hier konstanten) Registerinhalte stehen in er festgelegten Reihenfolge von links nach rechts durch Kommata voneinander getrennt. Das ganze Tupel wird in geschweifte Klammern eingeschlossen:

Syntax:

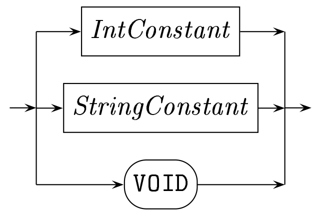


zu betrachten. Nichts desto trotz sollen sie an dieser Stelle gesondert beschrieben werden - schon weil sie es erlauben, dem Leser das Konzept der Zustandspartitionierung durch Register in SCARLET bereits zu diesem frühen Zeitpunkt vorzustellen.

Es müssen stets für alle deklarierten Register Werte angegeben werden, deren Datentyp mit dem jeweils korrespondierenden Registertyp identisch ist, d.h. *RegConstant* ist aus *IntConstant* oder aus *StringConstant* zu wählen oder es enthält den Wert `VOID`:

RegConstant

RegConstant:



`VOID`

`VOID` kann als einziger Wert sowohl in String-Registern als auch in Int-Registern stehen, da er eigentlich eine Anweisung darstellt:
Wird nämlich einem beliebigen Objekt vom Typ `CELL` eine Zellkonstante zugewiesen, so bleiben all diejenigen Register von der Wertzuweisung unberührt, die auf den Wert `VOID` treffen. (siehe auch S. 64) Zellkonstanten sind selbst auch vom Typ `CELL`.

Bsp.8: Wir betrachten für eine Zellstruktur bestehend aus drei Registern mit den Typen `INT`, `STRING` und `INT` (in genau dieser Reihenfolge) folgende Beispiele für Konstanten:

× `{"Wort", -567}`	(falsch)
√ `{-123, "Zustand", VOID}`	(richtig)
× `{0, , 0}`	(falsch)
√ `{0, "", 0}`	(richtig)

2.4.5 Konstantenbezeichner *ConstantIdentifier*

Über die einfache Verwendung von Konstanten durch deren explizite Angabe innerhalb des Programmtextes hinaus, gestattet SCARLET auch den Gebrauch von Pseudonymen aus *Identifier* an ihrer Stelle. Wir nennen sie Konstantenbezeichner. Solche Vereinbarungen müssen jeweils im Deklarationsteil eines (SDL- oder RDL-) Programmes getroffen werden. Im Rahmen einer solchen Aktion erhält ein Konstantenbezeichner automatisch einen der Typen `INT`, `STRING`, `BOOL` oder `CELL`. Je nach dem sprechen wir auch von Elementen aus

IntConstantIdentifier bzw. *IntIdentifier*,
StringConstantIdentifier bzw. *StringIdentifier*,
BoolConstantIdentifier bzw. *BoolIdentifier*,
CellConstantIdentifier bzw. *CellIdentifier*.

Da ihre Bedeutung an die jeweilige Programmumgebung gekoppelt ist, verschieben wir eine nähere Betrachtung bis zur Besprechung von RDL, Seite 73.

2.4.6 Variablen

Var

Für noch mehr Programmierkomfort sorgt SCARLET durch die Bereitstellung von Variablen. Bezeichnet werden sie stets mit einem Element aus *Identifier*. Im Gegensatz zu diversen anderen Programmiersprachen (wie z.B. C), müssen alle Variablen vor ihrer Verwendung am Beginn des entsprechenden Programms deklariert werden. Mehr hierzu finden Sie auf Seite 72. Sinn und Zweck dieses Vorgehens ist die Zuordnung zu einem der in SCARLET bekannten Datentypen `INT`, `STRING`, `BOOL` oder `CELL` (und damit die Reservierung von Speicherplatz in der jeweils benötigten Größe). Der durch die Deklaration festgelegte Typ einer Variablen kann im Laufe des Programmes nicht mehr geändert werden. Wir sprechen daher statt von Objekten aus *Var* vielmehr von Elementen aus

IntVar	(für Variablen vom Typ	`INT`)
StringVar	(-“-	`STRING`)
BoolVar	(-“-	`BOOL`)
CellVar	(-“-	`CELL`)

Bilden wir bei einer Zellvariablen die Projektion auf ein bestimmtes Register durch Anhängen des Registernamens (vgl. S. 50), so erhalten wir wiederum ein Objekt aus *IntVar*, falls das Register vom Typ `INT` ist,

Einzelne Register

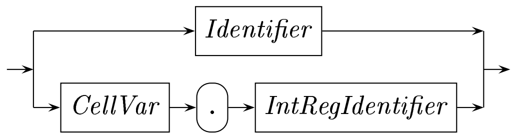
CellVar.*IntRegIdentifier*

und ein Element aus *StringVar*, falls es sich um ein Register vom Typ `STRING` handelt.

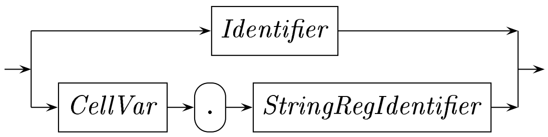
CellVar.*StringRegIdentifier*

Die Bildungen für numerische und Stringvariablen noch einmal auf einen Blick

IntVar:



StringVar:



2.5 Operatoren

Die nun folgenden Abschnitte sind den in SCARLET auftretenden Operatoren gewidmet. Je nach ihrer Wirkungsweise unterteilen sie sich in die Gruppen der vergleichenden, logischen, binären und arithmetischen Operatoren. Manche der Operatorzeichen werden je nach Kontext gleich in mehreren, ähnlichen Bedeutungen verwendet, die hier gemeinsam besprochen werden. Auf einige spezielle Operatoren, wie `[]` zur Extraktion einzelner Zeichen aus Stringausdrücken oder `.` als Projektionsoperator auf Zellausdrücken, sowie auf die Gruppe der Zuweisungsoperatoren gehen wir in späteren Abschnitten gesondert ein.

Zweistellige Operatoren

Alle der hier besprochenen Operatoren mit Ausnahme des logischen Negationszeichens `!` sowie der Vorzeichen `+` und `-` benötigen zwei Argumente in Infixnotation. Die Syntax lautet daher allgemein:

$$\textit{Expression}_1\ \textit{Op}\ \textit{Expression}_2$$

Op steht dabei für den betreffenden zweistelligen Operator, während die beiden Argumente $\textit{Expression}_1$ und $\textit{Expression}_2$ Elemente einer Menge von Ausdrücken eines bestimmten Typs sind. Das Ergebnis einer solchen Aktion ist ebenfalls ein Ausdruck. (Mehr hierzu finden Sie in Abschnitt 2.8, Seite 54.)

Einstellige Operatoren

Die einstelligen Operatoren werden ihrem Argument direkt, d.h. ohne Leerzeichen, vorangestellt:

$$\textit{Op}\ \textit{Expression}$$

Die obigen Bemerkungen über Argument- und Ergebnismenge gelten für sie in gleicher Weise.

2.5.1 Vergleichsoperatoren *RelOp*

Numerische Ausdrücke

Wie gewohnt verfügen wir auch in SCARLET über die üblichen zweistelligen Vergleichsoperatoren auf der Menge dernumerischen Ausdrücke *IntExpression* (siehe dazu auch S. 54). Im einzelnen sind das

Relop:

`==`	gleich
`<`	(echt) kleiner
`>`	(echt) größer
`!=`	ungleich
`>=`	größer oder gleich
`<=`	kleiner oder gleich

Als Ergebnis liefern sie jeweils einen booleschen Ausdruck, ihrem Wahrheitswert entsprechend.

Strings

Dieselben Operatoren lassen sich aber auch auf String-Ausdrücke (kurz: Strings) (siehe S. 56) anwenden. Dabei tritt anstelle des numerischen Vergleichs der lexikalische, d.h. der Vergleich zweier Strings erfolgt zeichenweise von links nach rechts, und es gilt: Zwei Strings sind gleich, wenn beide gleich lang und alle Zeichen identisch sind, sonst sind sie ungleich. Ein String S_1 der Länge n, $n \in I\!N$, ist größer als ein zweiter String S_2, falls die ersten $i-1$, $i \leq n$, Zeichen identisch sind und entweder die ASCII-Zeichen-Nummer des i-ten Zeichens von S_1 größer ist als die des i-ten Zeichens von S_2, oder S_2 nur $i-1$ Zeichen lang ist.

Beachten Sie hierbei die Besonderheiten in der Anordnung des ASCII-Zeichen-Codes:

ASCII-Code

```
§ ! " # $ % & ' ( ) * + , - . / 0..9 : ; < = > ? @ A..Z [
\ ] ^ _ ` a..z { | } ~ ü ä Ä ö Ö Ü ° ␣
```

Bsp.9:

× `"Register"> "Register-Inhalt"`⇝ `FALSE`
√ `"register"> "Register-Inhalt"`⇝ `TRUE`
× `"99_Register"> "Register"`⇝ `FALSE`

Zell-ausdrücke

Auch ***Zellausdrücke*** (siehe S. 58) lassen sich bedingt miteinander vergleichen: Zwar sind die `>` und `<` Operatoren hier verständlicherweise nicht mehr sinnvoll einsetzbar, jedoch aber `==` sowie `!=`. Dann erfolgt der Vergleich registerweise. Zwei Zellausdrücke sind gleich, wenn sie registerweise gleich sind, sonst ungleich. (Daß beide Zellausdrücke über diesselbe Registerstruktur verfügen, wird vorausgesetzt.)

Partielle Vergleiche

Der Vergleich eines Registerinhalts mit `VOID` ergibt beim Test zweier Zellausdrücke auf Gleichheit stets TRUE, beim Test auf Ungleichheit FALSE. Auf diese Art werden partielle Vergleiche von Zellausdrücken ermöglicht.

Bsp.10:

$\surd$ `{"z", 25, VOID} == {"z", 25, "a"}` $\leadsto$ `TRUE`
$\times$ `{"z", 25, VOID} != {"z", 25, "a"}` $\leadsto$ `FALSE`

2.5.2 Logische Operatoren *LogOp*

Für die drei grundlegenden logischen Operatoren UND, ODER und NICHT stehen in SCARLET `!`, `&&` sowie `||` zur Verfügung:

LogOp:

`!`	logische Negation
`&&`	logisches UND
`||`	logisches ODER

Sie operieren innerhalb der Menge der booleschen Ausdrücke (siehe S. 59). `!` ist im Gegensatz zu `&&` und `||` ein einstelliger Operator[2] und wird seinem Argument direkt (ohne Leerzeichen) vorangestellt.

2.5.3 Binäre Operatoren *ShiftOp, BitOp*

Des weiteren haben wir in SCARLET auch die Binäroperatoren, die auf der Menge der numerischen Ausdrücke, (siehe S. 54) also letztendlich auf $\mathbb{Z}$ agieren. Allerdings bedienen sie sich der maschineninternen Binärdarstellung der Ausdrücke als 32-stelliges Maschinenwort. (Für das folgende ist es nicht ganz unpraktisch, sich ein solches Maschinenwort als Maske mit 32 Feldern

[2] Aus diesem Grund gehört er nicht in die syntaktische Klasse *LogOp*.

vorzustellen. Davon dient das am weitesten links stehende zur Vorzeichendarstellung (`+` entspricht 0, `-` entspricht 1), die restlichen 31 nehmen die eigentliche Binärzahl auf.) Wir unterscheiden

ShiftOp:

`<<`	Linksshift
`>>`	Rechtsshift

BitOp:

`&`	bitweises UND
`|`	bitweises ODER

Shift-Operatoren

Die Shiftoperatoren in SCARLET orientieren sich in ihrer Funktionsweise an den entsprechenden in C für signed-Variablen.

$$IntExpression_1 \texttt{<<} IntExpression_2$$

bedeutet, daß das Bitmuster von $IntExpression_1$ um $IntExpression_2$ Stellen nach links geschoben wird. Dabei gehen alle Bits verloren, die nach links aus der Maske „herausfallen“; von rechts werden Bits der Wertigkeit `0` nachgeschoben.
Ähnlich ist die Funktionsweise für den Rechtsshift:

$$IntExpression_1 \texttt{>>} IntExpression_2$$

$IntExpression_1$ wird um $IntExpression_2$ Positionen nach rechts verschoben. Allerdings entscheidet hier das Vorzeichen von $IntExpression_1$ über die Wertigkeit der nachgeschobenen Bits: Bei positivem Vorzeichen 0, bei negativem 1.

Bsp.11:	`11 << 4`	$\leadsto 00\cdots0000001011$
		$\leadsto 00\cdots0010110000 \leadsto$ `176`
	`-14 >> 1`	$\leadsto 1000\cdots001101$
		$\leadsto 1100\cdots000110 \leadsto$ `-1.073.741.830`

Logische Operatoren

Für `&` und `|` werden alle Einsen der Binärdarstellung als TRUE, alle Nullen als FALSE interpretiert. Schließlich werden beide Ausdrücke bitweise miteinander logisch UND bzw. ODER verknüpft und die Ergebnisse der Einzelvergleiche wieder in Form von Einsen und Nullen festgehalten.

Bsp.12: `13 & -7` $\leadsto$

$$\begin{array}{rl} 00\cdots01101 & (=13) \\ \underline{\&\ \ 10\cdots00111} & (=-7) \\ 00\cdots00101 & \leadsto \texttt{5} \end{array}$$

Alle binären Operatoren liefern also wiederum numerische Ergebnisse.

2.5.4 Arithmetische Operatoren *AddOp, MulOp*

Selbstverständlich kennt SCARLET auch die zweistelligen arithmetischen Operatoren für die vier Grundrechenarten.

MulOp:

`*`	Multiplikation
`%`	Modulo-Operator
`/`	Ganzzahl-Division

AddOp:

`+`	Addition
`-`	Subtraktion

Modulo

Sie operieren in gewohnter Weise auf der Menge der numerischen Ausdrücke (siehe S. 54) und haben auch solche zum Ergebnis. Es sei darauf hingewiesen, daß alle Operationen auf einer Teilmenge der ganzen Zahlen stattfinden, und daß aus diesem Grunde die „normale“ Division ersetzt werden muß durch die Variante `/`, die den Ganzzahlanteil des Divisionsergebnisses liefert und den Modulo-Operator `%` für den ganzzahligen Divisionsrest.

Bsp.13:
`7 % 3` $\leadsto$ `1`
`7 / 3` $\leadsto$ `2`
`7 % 3 + 7 / 3` $\leadsto$ `3`

Vorzeichen

`+` und `-` werden jedoch auch als einstellige Operatoren gebraucht: In diesem Fall stehen sie ihrem Argument voran und spielen die Rolle von Vorzeichen.

Konkatenation

In Bezug auf String-Ausdrücke (siehe S. 56) verwenden wir den `+`-Operator jedoch in einer anderen Funktion, nämlich als Konkatenationszeichen:

$$\mathit{StringExpression}_1 \texttt{ + } \mathit{StringExpression}_2$$

steht für den String-Ausdruck, der entsteht, wenn *StringExpression*$_2$ an *String-Expression*$_1$ angehängt wird. Dabei darf natürlich die vorgegebene maximale Stringlänge von 255 Zeichen nicht überschritten werden.

Bsp.14: `"hundert"+ "fünf"`$\leadsto$ `"hundertfünf"`
`"fünf"+ "hundert"`$\leadsto$ `"fünfhundert"`

2.5.5 Rangfolge und Assoziativität

Die nachstehende Tabelle enthält noch einmal alle Operatoren auf einen Blick, darunter auch die Zuweisungsoperatoren (`=`, `+=`, `-=`, ..., `++`, `--`) sowie `[ ]` und `.`, auf die wir später noch zu sprechen kommen.
Um eventuellen Verwechslungen vorzubeugen, sei jedoch bereits an dieser Stelle erwähnt, daß mit Hilfe des Klammeroperators `[ ]` der Zugriff auf eine bestimmte Position innerhalb eines Strings realisiert wird (und daß damit nicht etwa die eckigen Klammern als Begrenzungszeichen einer Zellposition gemeint sind). Der Punkt-Operator `.` realisiert den Zugriff auf einzelne Register eines Zellausdrucks (siehe S. 50).

Die Auflistung der Operatoren erfolgt zeilenweise in absteigender Rangfolge, d.h. Auswertungspriorität. Zusätzlich geben wir für Operatoren gleichen Rangs an, ob deren Abarbeitung durch den Compiler von links nach rechts (= **L**inksassoziativität) oder umgekehrt (= **R**echtassoziativität) erfolgt.
SCARLET orientiert sich in Bezug auf Rangfolge und Assoziativität vollständig an den Vorgaben für die Sprache C.

<table>
<tr><th>Operatoren</th><th>Assoziativität</th></tr>
<tr><td>( ), [ ], .</td><td>L</td></tr>
<tr><td>++, --, !, + (Sign), - (Sign)</td><td>R</td></tr>
<tr><td>*, /, %</td><td>L</td></tr>
<tr><td>+, -</td><td>L</td></tr>
<tr><td><<, >></td><td>L</td></tr>
<tr><td><, <=, >, >=</td><td>L</td></tr>
<tr><td>==, !=</td><td>L</td></tr>
<tr><td>&</td><td>L</td></tr>
<tr><td>|</td><td>L</td></tr>
<tr><td>&&</td><td>L</td></tr>
<tr><td>||</td><td>L</td></tr>
<tr><td>=, +=, -=, *=, %=, /=, <<=, >>=, &=, |=</td><td>R</td></tr>
</table>

Bsp.15: Die Rangfolge wie oben angegeben macht an einigen Stellen die Verwendung von Klammern unumgänglich:

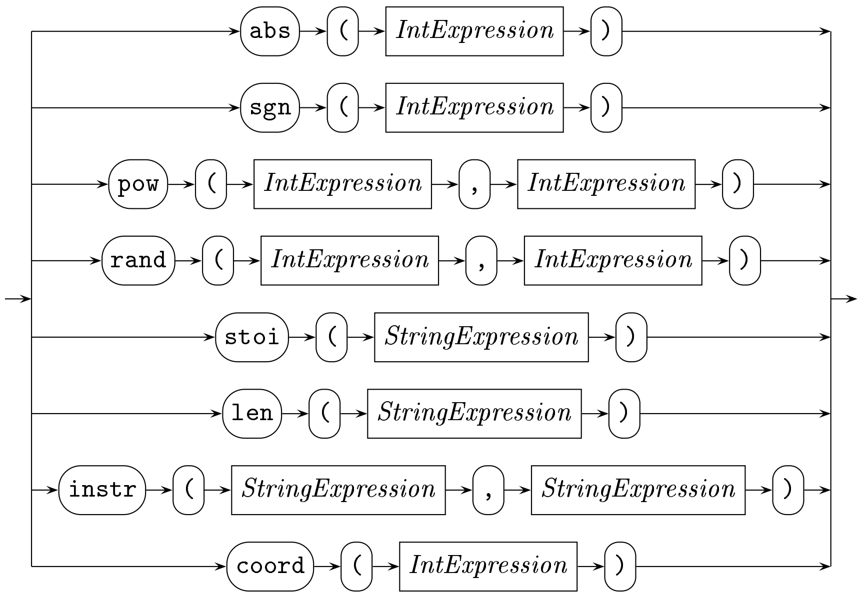
$\surd$ `!TRUE || TRUE` $\leadsto$ `TRUE`
$\times$ `!(TRUE || TRUE)` $\leadsto$ `FALSE`
$\surd$ `1000 < 1024 && !("dies"=="das")` $\leadsto$ `TRUE`
$\surd$ `16 >> 1 == 2 << 2` $\leadsto$ `TRUE`

2.6 Funktionen

Außer den grundlegenden Operatoren verfügt SCARLET auch über einige eingebaute Funktionen, die den Umgang mit Strings (S. 56) und numerischen Ausdrücken (S. 54) erleichtern sollen. Bei der Vorstellung der einzelnen Funktionen unterscheiden wir solche, die als Ergebnis einen numerischen Wert haben, und solche, die einen String liefern.

2.6.1 Numerische Funktionen *IntFunctionCall*

Syntax:



`abs(` *IntExpression* `)`

berechnet den Absolutwert von *IntExpression*.

`sgn(` *IntExpression* `)`

liefert im wesentlichen das Vorzeichen von *IntExpression*. Da das Ergebnis ein numerischer Ausdruck sein soll, ist der Rückgabewert `1` bei positivem *IntExpression*, `-1` bei negativem Argument und `0`, falls *IntExpression* selbst Null ist.

`pow(` $IntExpression_1$, $IntExpression_2$ `)`

berechnet die Potenz von $IntExpression_1$ zum Exponent $IntExpression_2$. Um den Bereich der ganzen Zahlen nicht zu verlassen, sind negative Werte für den Exponenten untersagt. Bedenken Sie auch, daß der Absolutbetrag des Ergebnisses $2^{31} - 1$ nicht überschreiten darf.

Bsp.16:

`pow(10,5)` $\leadsto$ `100000`
`pow(34,0)` $\leadsto$ `1`
`pow(-2,3)` $\leadsto$ `-8`

`rand(` i_1 , i_2 `)`

mit $i_1, i_2 \in$ *IntExpression*, ermittelt zufällig eine ganze Zahl im abgeschlossenen Intervall $[i_1,\ i_2]$, falls $i_1 \leq i_2$, und aus $[i_2,\ i_1]$, falls $i_2 < i_1$.

Bsp.17:

`rand(1,49)`	Zufallszahl zwischen 1 und 49.
`rand(49,1)`	
`rand(9,9)`	liefert immer `9`.

Bsp.18: Die Random-Funktion erlaubt die Konstruktion von **probabilistischen Zellularautomaten**nach folgendem Prinzip:
Eine für jede Zelle der Retina neu ermittelte Zufallszahl zwischen 1 und $n \in I\!N$ wird in der Variablen `number` gespeichert. Für ein und dieselbe Situation <*Neighbour configuration*> erfolgen nun verschiedene Handlungen <*Consequence 1*>, <*Consequence 2*>, usw. mit den Wahrscheinlichkeiten $\frac{n_1-1}{n}$, $\frac{n_2-n_1}{n}$, ..., $\frac{n-n_m}{n}$, $1 < n_1 < \ldots < n_m \leq n$.
Ist die Bezugszelle selbst im Nachbarschaftsindex enthalten, so ist $n_m < n$ erlaubt - in diesem Fall behält sie mit Wahrscheinlichkeit $\frac{n-n_m}{n}$ ihren alten Zustand bei. Somit wird automatisch gewährleistet, daß sich alle Wahrscheinlichkeiten zu 1 aufsummieren.

```
     ...
TABLE
  number=rand(1,n);

  number>0; number<=n1;
  <Neighbour configuration> : <Consequence 1> :

  number>n1; number<=n2;
  <Neighbour configuration> : <Consequence 2> :
       ...
```

Zum Verständis des SDL-Programmaufbaus siehe auch Abschnitt 4.2.2 und 4.2.3.

`stoi(` *StringExpression* `)`

verwandelt *StringExpression* bei Position 0 beginnend so weit wie möglich in eine Ganzzahlkonstante. Dabei gilt:

- Das erste Zeichen, das nicht in das Format einer Ganzzahl paßt, beendet die Konvertierung; alle nachfolgenden Zeichen werden ignoriert. Insbesondere liefert `stoi()` den Wert `0`, falls die Konvertierung bereits fehlschlägt, bevor sie auf eine Ziffer gestoßen ist.
- Führende Leerzeichen - und nur diese - werden übergangen.
- Das Ergebnis einer Konvertierung darf $2^{32} - 1$ nicht überschreiten. Für Strings, die größere Rückgabewerte liefern würden, ist die Funktion nicht definiert.

Bsp.19:

```
stoi("     -431332"); ⇝ -431332
stoi("120.982.566"); ⇝ 120
stoi("Zustand 23"); ⇝ 0
```

`len(` *StringExpression* `)`

ermittelt die Anzahl Zeichen, aus denen *StringExpression* besteht.

Bsp.20: Sei `dreizehn` eine String-Variable mit Inhalt `"13"`.

```
len("Achtzehn Buchstaben") ⇝ 19
len(dreizehn) ⇝ 2
```

`instr(` *StringExpression*$_1$ `,` *StringExpression*$_2$ `)`

durchläuft *StringExpression*$_1$ von links nach rechts und stellt fest, ob *StringExpression*$_2$ als Teilzeichenkette enthalten ist. Wenn ja, so ist der Rückgabewert die Position des ersten übereinstimmenden Zeichens, sonst `-1`. (Das erste Zeichen eines Strings steht an Position 0.) Ist *StringExpression*$_2$ mehrfach in *StringExpression*$_1$ enthalten, so zählt die erste Übereinstimmung.

Bsp.21:

```
instr("PAPPERLAPAPP!, rief der Papagei","Pa") ⇝ 24
instr("PAPPERLAPAPP!, rief der Papagei","P") ⇝ 0
instr("PAPPERLAPAPP!, rief der Papagei","pp") ⇝ -1
```

`coord(` *IntExpression* `)`

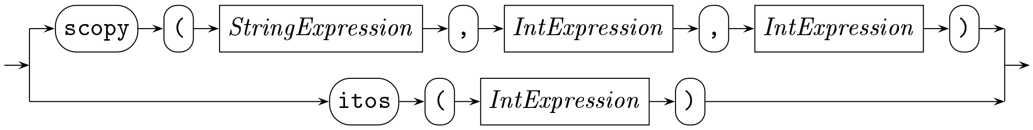
existiert nur in RDL und auch dort nur im Rahmen von Zellzuweisungen *CellAssignment* an Stellen, an denen numerische Ausdrücke zugelassen sind. Bezeichnet d die Dimension des Raumes, so muß *IntExpression* so gewählt sein, daß gilt: $1 \leq IntExpression \leq d$.
Besteht *CellDomain* aus einer einzelnen Zelle *Cell*, so liefert `coord(`i`)` die i-te Komponente der Koordinate von *Cell*.
Stellt *CellDomain* eine Menge von Zellen *Loop* dar, so wird `coord(`i`)` intern auf alle Zellen der Menge einzeln angewandt. (D.h. jeder Zelle aus *CellDomain* wird die i-te Komponente ihrer Koordinate zugewiesen.) Genaueres über *CellAssignment*, *Cell* und *Loop* finden Sie in Kapitel 4 über RDL, Seite 74 ff..

Bsp.22: Wir betrachten für eine Retina im $\mathbb{Z}^2$ eine Zustandsstruktur mit zwei numerischen Registern `x1` und `x2`. Alle Zellen in dem Quadrat mit den Eckpunkten $(0, 0)$ und $(10, 10)$ seien in der Retina enthalten. Durch die folgenden Befehle im Anweisungsteil des zugehörigen RDL-Programms werden die `x1`- und `x2`-Register dieser Zellen mit deren Koordinaten initialisiert:

```
[0,0]..[10,10].x1 = coord(1);
[0,0]..[10,10].x2 = coord(2);
```

2.6.2 String-Funktionen *StringFunctionCall*

Syntax:



`scopy(` *StringExpression* , $IntExpression_1$, $IntExpression_2$)

liefert einen Teilstring von *StringExpression*, startend bei $IntExpression_1$. (Auch hier gilt: Das erste Zeichen eines Strings steht an Position 0.)
Mit $IntExpression_2 \geq 0$ wird die Länge des Teilstrings festgelegt. Achten Sie darauf, daß er vollständig in *StringExpression* enthalten ist, d.h. daß
$IntExpression_1 + IntExpression_2 \leq$ `len(` *StringExpression* `)` erfüllt ist.

Bsp.23: `scopy("Zustand 13",8,2)` $\rightsquigarrow$ `13`
`scopy("1234567890",3,1)` $\rightsquigarrow$ `4`

`itos(` *IntExpression*)

wandelt *IntExpression* um in eine Stringkonstante. Dabei darf *IntExpression* keine führenden Nullen besitzen.

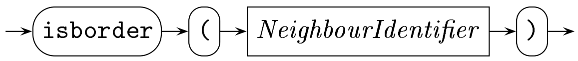
Bsp.24: `itos(-10203)` $\rightsquigarrow$ `"-10203"`

Sei `str` eine Stringvariable mit Inhalt `"23. Zustand"`.

`itos(stoi(str)+1) + ". Zustand"` $\rightsquigarrow$ `"24. Zustand"`

2.6.3 Boolesche Funktionen *BoolFunctionCall*

Syntax:



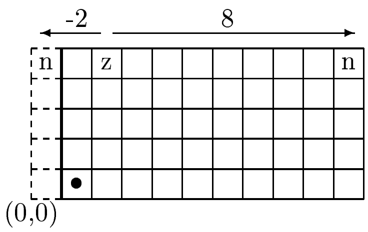
`isborder(` *NeighbourIdentifier*)

Diese Funktion steht nur im Anweisungsteil von SDL-Programmen zur Verfügung. Dort ermittelt sie, ob der Nachbar *NeighbourIdentifier* der aktuellen Bezugszelle außerhalb des gültigen Retina-Bereichs liegt, also eine Zelle der Grenzschicht ist, oder nicht. Im ersten Fall liefert sie den Wert TRUE, im zweiten FALSE. Siehe auch: RDL, 'Die Bereichsvereinbarung', S. 69, und: SDL, 'Die Überführungstafel', S. 96

Bsp.25: Mit Hilfe der `isborder`-Funktion kann auf einfache Art eine torusförmige Retina realisiert werden. Zu diesem Zweck fügt man für jeden Nachbarn, der in Konflikt mit einer „Klebestelle" gerät, einen weiteren in den Nachbarschaftsindex ein, bei dem die Komponenten der zu verklebenden Koordinatenrichtungen durch die betraglich kleinste äquivalente umgekehrten Vorzeichens in der zyklischen Gruppe $(Z_n, +)$ ersetzt wird (wenn n die Ausdehnung der Retina in dieser Richtung bezeichnet). Oder anders formuliert: Man beschreibt dieselbe Zelle unter Verwendung der umgekehrten Laufrichtung. Außerdem ergänzt man die Überführungstafel an den kritischen Stellen um solche Einträge, die überprüfen, ob der betreffende Nachbar in der Grenzschicht liegt, und in diesem Fall den Ersatz-Nachbar aktivieren.

Wir betrachten einen **Torus**, entstanden aus einer 2-dimensionalen Retina `[0,0]..[9,4]` durch Verkleben der x_1-Koordinaten, auf dem die Zellzustände gegen den Uhrzeigersinn im Kreis herumlaufen, d.h. in jedem Simulationsschritt an die übernächste Zelle in x_1-Richtung übergeben werden:

```
SIGMA torus;
DIMENSION 2;      /* 2-dimensionale Retina */
REGISTER INT state;      /* numerisches Register state */
NEIGHBOUR l_left : (-2,0);      /* richtiger Nachbar */
          l_left_behind_border : (8,0); /* Pseudo-Nachbar */

TABLE
  isborder(l_left); : SELF=l_left_behind_border; :

  /* andernfalls */   : SELF=l_left; :
```



Zwei Arten, den Nachbar n von z auf dem Torus zu adressieren.

2.7 Ansprechen einzelner Register

Wie wir bereits gesehen haben, setzt sich der Zustand einer Zelle eines Zellularautomaten in SCARLET aus verschiedenen Teilzustandsinformationen zusammen, die vom Typ `INT` oder `STRING` sind und in Registern entsprechenden Typs abgelegt werden. Der Typ `CELL` in SCARLET ist somit vergleichbar mit den zusammengesetzten Datentypen `struct` in C oder `RECORD` in Pascal. Es scheint daher auch hier mehr als wünschenswert, Register einzeln ansprechen und deren Inhalt weiterverarbeiten zu können. Ermöglicht wird dies durch deren Deklaration und Benennung im Vereinbarungsteil jedes RDL- und SDL-Programms (siehe S. 70). Der Registername dient dann in Verbindung mit einem Objekt vom Typ `CELL` zum gezielten Ansprechen der gewünschten Zellkomponente (hier also: eines bestimmten Registers).
Das so entstandene neue Objekt ist vom Typ des angesprochenen Registers, also vom Typ `INT` oder `STRING`. Wie in C oder Pascal werden Objekt- und Registername durch den sogenannten Punktoperator `.` voneinander getrennt: Allgemein:

<Objektname> . <Registername>

Einzige Ausnahme hierfür stellt die *SelfAssignment*-Umgebung in SDL dar: Dort genügt die Angabe des Registernamens, meint man die „Bezugszelle“ der Überführungsfunktion, die bei der Durchführung eines Simulationsschrittes alle möglichen Zellen der Retina durchläuft.

Beachte: An der Vorgehensweise ändert sich nichts, wenn die Zellstruktur nur aus einem einzigen Register besteht. Selbst in diesem Fall ist der Zellinhalt als solcher vom Typ `CELL`, nicht etwa schon vom Typ `INT` oder `STRING`. Auch hier muß erst ein Zugriff auf dieses einzige Register erfolgen, um den Inhalt als Zahl oder String zu erhalten.

Es ist nötig, zwischen vier Möglichkeiten des Zugriffs zu differenzieren:

a) Zugriff auf Zellkonstanten und Zellvariablen

mit	*CellIdentifier.IntRegIdentifier*	(*IntReg*)
	CellIdentifier.StringRegIdentifier	(*StringReg*)

Objekte dieser Art sind wie normale numerische oder String-Konstanten oder Variablen zu behandeln und sowohl in RDL als auch in SDL einsetzbar. Zu beachten ist lediglich, daß Register niemals einzeln als Konstanten oder Variablen deklariert werden dürfen, stets nur im Zellverband.

Bsp.26:

```
     ...
REGISTER INT num;
         STRING str;
VAR CELL a_variable;
CONST a_constant={5,"nichts bestimmtes"};

     ...
a_variable.str=a_constant.str;
a_variacle.num=a_constant.num * 3;
     ...
```

b) Zugriff auf Zellen durch Positionsangabe

mit	*CellDomain.IntRegIdentifier*	*StringRegSelect*
	CellDomain.StringRegIdentifier	*IntRegSelect*

ist nur in RDL zum Zwecke der Bildung einer Startkonfiguration möglich. Dabei stellt *CellDomain* eine (oder mehrere) gültige Koordinate(n) innerhalb der vereinbarten Retina dar, während *IntRegIdentifier* bzw. *StringRegIdentifier* Namen für Register vom Typ `INT` bzw. `STRING` darstellen (siehe auch: 'Wertzuweisungen an Zellen', S. 79).
Der Zugriff erfolgt aus naheliegenden Gründen nur schreibend.

Bsp.27: Das folgende Retina-Beispiel (vgl. [6]) soll die Momentaufnahme eines `1`-verschneiten `0`-Himmels darstellen: Dafür wird in die Register aller Zellen der 2-dimensionalen Retina mit den Eckkoordinaten $(0,0)$ und $(5,10)$ zufällig Nullen oder Einsen geschrieben. Die Zelle mit der Koordinate $(2,4)$ soll in jedem Fall eine Schneeflocke enthalten.

```
RETINA falling_snow;
RANGE [0,0]..[5,10];      /* rechteckige Retina */
REGISTER INT state;     /* numerisches Register state */

START
  [0,0]..[5,10].state = rand(0,1);     /* zuf. 0/1-Muster */
  [2,4].state = 1;     /* eine 'sichere' Schneeflocke */
```

c) Zugriff auf Nachbarn

mit	*NeighbourIdentifier . IntRegIdentifier*	(*IntReg*)
	NeighbourIdentifier . StringRegIdentifier	(*StringReg*)

ist eine Spezialität von SDL: Bei *NeighbourIdentifier* handelt es sich um einen gültigen Bezeichner für einen sogenannten Nachbarn, festgelegt durch den Nachbarschaftsindex, während *IntRegIdentifier* und *StringRegIdentifier* wieder Namen für numerische bzw. Stringregister darstellen (siehe SDL, Theorieteil). Da Nachbarschaft im Zusammenhang mit der Retina-Definition keine Rolle spielt, sind solche Objekte in RDL nicht definiert.
Der Zugriff erfolgt lediglich lesend. (Der Zustand eines Nachbarn hat zwar i.a. einen Einfluß auf die Bezugszelle, er selbst darf aber nicht geändert werden.)

Bsp.28: (Fortsetzung von Bsp. 27, vgl. [6])
Passend zu der eben konstruierten Retina haben wir hier nun eine Überführungstafel mit zwei Einträgen, die das Fallen und Liegenbleiben des Einser-Schnees simuliert:

1. Ist die Bezugszelle noch nicht belegt, und befindet sich über ihr eine Schneeflocke, so nimmt sie diese auf.

2. Befindet sich an der Stelle der Bezugszelle bereits eine Schneeflocke, und ist unter ihr noch Platz, so gibt sie sie ab. (Die untere Zelle wird sie nach der oberen Regel auffangen.)

Die Register der Grenzschichtzellen enthalten den Wert `35` und bilden somit eine natürliche Begrenzung.

```
SIGMA falling_snow;
DIMENSION 2;
REGISTER INT state;
NEIGHBOUR upper: (0,1);     /* räumlich oberer Nachbar */
          lower: (0,-1);     /* räumlich unterer Nachbar */
          center:(0,0);     /* Bezugszelle selbst */
```

```
TABLE
  /* 1. Eintrag: */
  upper.state == 1;     /* Zustände d. Nachbarn */
  center.state == 0; :     /* -"- */
  state = 1; :     /* Zuweisung an die Bezugszelle */

  /* 2. Eintrag: */
  lower.state == 0;     /* analog */
  center.state == 1; :
  state = 0; :
```

d) Zugriff auf die Bezugszelle

mit *IntRegIdentifier*
StringRegIdentifier

- wobei *IntRegIdentifier* und *StringRegIdentifier* jeweils Namen für numerische bzw. Stringregister darstellen - erfolgt ohne Verwendung des Punktoperators, da sich die Register-Angaben stets auf die Bezugszelle `SELF` beziehen. Zugriffe dieser Art sind nur in SDL gestattet und erfolgen stets schreibend zu dem Zweck, der Bezugszelle einen anderen Zustand zuzuweisen (vgl. Beispiel 28).

Bemerkung zur Syntax

Durch die oben vorgenommene Aufschlüsselung verschiedener Sprachelemente nach unterschiedlichen Funktionsweisen ist der syntaktische Aufbau der hierdurch vorgestellten Klassen nicht mehr unmittelbar überschaubar. Wir reichen an dieser Stelle deshalb einen Überblick über die Klassen *IntReg* und *StringReg* nach. Wie oben erwähnt, muß zwischen deren Verwendung in RDL und SDL unterschieden werden:

IntReg (RDL):

→ *CellIdentifier* → . → *IntRegIdentifier* →

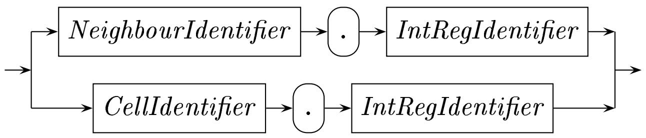
SDL↔RDL

StringReg (RDL):

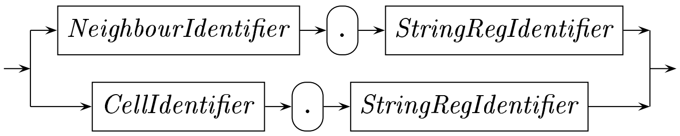
→ *CellIdentifier* → . → *StringRegIdentifier* →

SDL erweitert diese Syntax um diejenigen Anteile, in denen Nachbarn gebraucht werden:

IntReg (SDL):



StringReg (SDL):

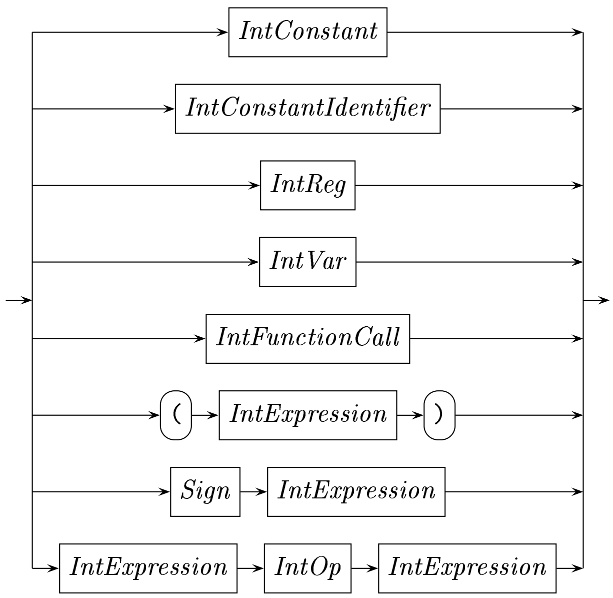


2.8 Ausdrücke

2.8.1 Numerische Ausdrücke

IntExpression

Syntax:



Als numerischen Ausdruck bezeichnen wir im Prinzip ein beliebiges Objekt vom Typ `INT`, oder anders ausgedrückt: einen „korrekt geklammerten arithmetischen Ausdruck“. Spätestens jetzt haben Sie bestimmt eine recht konkrete Vorstellung davon, was gemeint sein könnte - und im wesentlichen sind solche Assoziationen auch berechtigt, ja sogar beabsichtigt; aber, vergessen wir nicht: Unser Kommunikationspartner ist in diesem Fall ein Compiler, dessen Weltverständnis aus der Natur der Sache heraus nicht beliebig flexibel sein kann! Wir wollen uns daher trotzdem noch um eine etwas präzisere Beschreibung bemühen:

Elementare Ausdrücke

Zunächst haben wir bereits eine Reihe von elementaren numerischen Ausdrücken kennengelernt, deren genauere Beschreibung sich weiter vorne in demselben Kapitel findet, und die die Grundbausteine für alle anderen numerischen Ausdücke darstellen. Es handelt sich bei diesen Elementarausdrücken um

- Ganzzahlkonstanten (*IntConstant*)
- numerische Konstantenbezeichner (*IntConstantIdentifier*)
- numerische Variablen (*IntVar*)
- Ergebnisse numerischer Funktionen (*IntFunctionCall*)
- numerische Register (*IntReg*)

Achten Sie besonders auf bestehende Einschränkungen und Unterschiede in Bezug auf die Verwendung von *IntReg*-Objekten in RDL und SDL (siehe S. 53 im Abschnitt 'Ansprechen einzelner Register').

Zusammengesetzte Ausdrücke

Darüber hinaus besteht nun aber vielmehr die Möglichkeit, durch Anwendung der ebenfalls bereits vorgestellten zweistelligen Operatoren auf numerischen Ausdrücken *IntOp* (`*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `|`) komplexere Ausdrücke der Form

$$IntExpression_1 \ IntOp \ IntExpression_2$$

zu erzeugen.

Operatoren-Rangfolge

Auf diese Art ist es selbstverständlich möglich, Ausdrücke zu schaffen, in denen mehr als ein Operator auftritt. In diesem Fall wird die Reihenfolge ihrer Auswertung durch den Compiler mit Hilfe einer bestehenden Operatoren-Rangfolge geregelt: Operatoren höheren Rangs werden dabei vor solchen niedrigeren Rangs ausgewertet. (`*`, `/` und `%` besitzen beispielsweise einen höheren Rang als `+` und `-` in Anlehnung an die allgemeine Regel „Punktrechnung geht vor Strichrechnung“.) Eine vollständige Auflistung finden Sie auf Seite 43. Die Auswertung mehrerer Operatoren gleichen Rangs erfolgt hier stets von links nach rechts. (Die einzige Ausnahme bilden die - bis jetzt noch nicht erwähnten - Vorzeichen.)

Klammerausdrücke

Weiterhin erzeugen Sie durch Klammerung eines Ausdrucks wiederum einen Ausdruck:

(*IntExpression*)

Die Klammern bewirken (wie gewohnt) eine vorrangige Auswertung des durch sie eingefaßten Ausdrucks; gemeint ist natürlich: innerhalb der normalen Abarbeitungsfolge. Es ist also nicht so zu verstehen, daß jener Ausdruck innerhalb der höchsten Klammerebene automatisch zuerst bedient wird. Folgendes Beispiel soll helfen, den Sachverhalt etwas zu illustrieren:

Bsp.29: `(15+2)-(3*2)*(6/2*(27-25))`

$\leadsto (15+2) - 6 * (6/2 * (27 - 25))$
$\leadsto (15+2) - 6 * (3 * (27 - 25))$
$\leadsto (15+2) - 6 * (3 * 2)$
$\dots$
$\leadsto 17 - 12 = 2$

Insbesondere kann auf diese Art und Weise (wie am Beispiel zu sehen) die vorgegebene Operatorenrangfolge durchbrochen werden, d.h. Operatoren niedrigerer Priorität können vor solchen höherer Priorität ausgewertet werden.

Vorzeichen

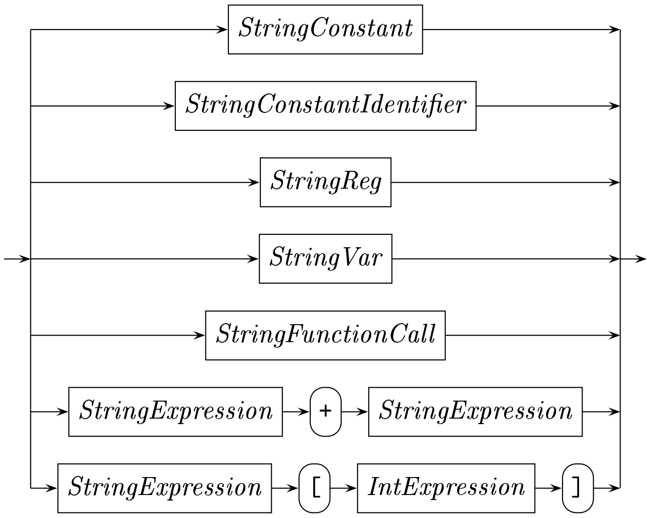
Prinzipiell ist es möglich, einen beliebigen Ausdruck durch Voransetzen eines Vorzeichens (`+`, `-`) aus *Sign* in einen neuen Ausdruck zu transformieren, und dadurch Ausdrücke zu erzeugen, in denen beliebig viele Operatoren (speziell `+` und `-`) aufeinanderfolgen. Solche Konstruktionen wollen wir aber nicht zulassen: Sie können sie durch entsprechende Klammersetzung umgehen.

Bsp.30: Ein paar Beispiele für numerische Ausdrücke: (`int_var` sei dabei eine numerische Variable mit Inhalt `-15`.

`-10 * +23` $\leadsto$ `-230`
`(1 + abs(int_var) / 13) << 2` $\leadsto$ `4`
`stoi("1001 Nacht") + (8+4+2) & (4+2+1)` $\leadsto$ `1007`

2.8.2 String-Ausdrücke *StringExpression*

Syntax:



Sehr viel unkomplizierter als der Aufbau numerischer Ausdrücke gestaltet sich jener der String-Ausdrücke (Strings). Das liegt ganz einfach daran, daß es nur einen einzigen innerhalb der Menge der Strings agierenden Operator gibt, nämlich den Konkatenationsoperator, der zwei Strings einfach aneinanderhängt.

In elementarer Weise Objekte der Klasse *StringExpression* sind

- Stringkonstanten (*StringConstant*)
- String-Konstantenbezeichner (*StringConstantIdentifier*)
- String-Variablen (*StringVar*)
- String-Register (*StringReg*)
- Ergebnisse von String-Funktionen (*StringFunctionCall*)

All diese Objekte wurden bereits zu einem früheren Zeitpunkt erläutert. Wie für *IntReg*, so ist auch hier zu beachten, daß nicht alle Objekte der Menge *StringReg* in RDL und SDL gleichermaßen verfügbar sind (siehe S. 53).

Teilstring-Bildung

Neben der allgemeinen `scopy`-Funktion existiert außerdem noch eine spezielle Form der Teilstringbildung. So liefert

$$\textit{StringExpression}\ \texttt{[}\ i\ \texttt{]}$$

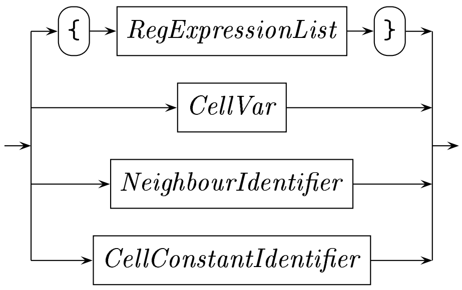
das i-te Zeichen aus *StringExpression*, wobei i aus *IntExpression* ist, $0 \leq i \leq$ `len`(*StringExpression*) $- 1$. (Die Zählung beginnt bei Null).

Bsp.31: Sei `language` eine Stringvariable mit Inhalt `"Retina Definition Language"`.

`language[0] + language[7] + language[18]` $\rightsquigarrow$ `"RDL"`

2.8.3 Zell-Ausdrücke *CellExpression*

CellExpression (SDL):



Noch unkomplizierter als die Beschreibung der Strings bereits war, gestaltet sie sich bei Zell-Ausdrücken: Für diese kennen wir nämlich keine rekursive Konstruktionsform. Damit werden alle denkbaren Zell-Ausdrücke durch die nachstehende Liste erfaßt:

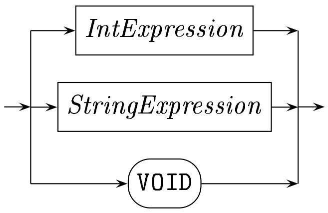
- Zellkonstanten (*CellConstant*)
- Konstantenbezeichner vom Typ `CELL` (*CellConstantIdentifier*)
- Zellvariablen (*CellVar*)
- Konstruktionen der Form { *RegExpressionList* }
- Nachbarn (*NeighbourIdentifier*)

Die als vorletztes aufgeführten Zell-Ausdrücke der Form

{ *RegExpressionList* }

wollen wir nun noch etwas genauer unter die Lupe nehmen: Bei den einzelnen Einträgen *RegExpression* handelt es sich um beliebige numerische oder String-Ausdrücke oder um den Wert `VOID`:

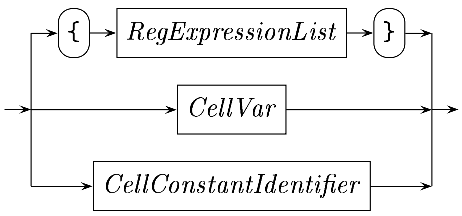
RegExpression:



Natürlich gilt auch hier (wie schon bei den Zellkonstanten beschrieben) einschränkend: Die genaue Struktur von *RegExpressionList*, d.h. Anzahl und Art der Einträge, hängt direkt von den Vereinbarungen über Anzahl und Art der verwendeten Register ab. Genaueres entnehmen Sie bitte den Abschnitten über Zellkonstanten (S. 34) und Register-Deklaration (S. 70).

Beachten Sie, daß die Verwendung von Nachbarn nur in SDL, nicht aber in RDL gestattet ist. Aus diesem Grund verwenden wir *CellExpression* in RDL in leicht veränderter Definition ohne *NeighbourIdentifier*: **SDL↔RDL**

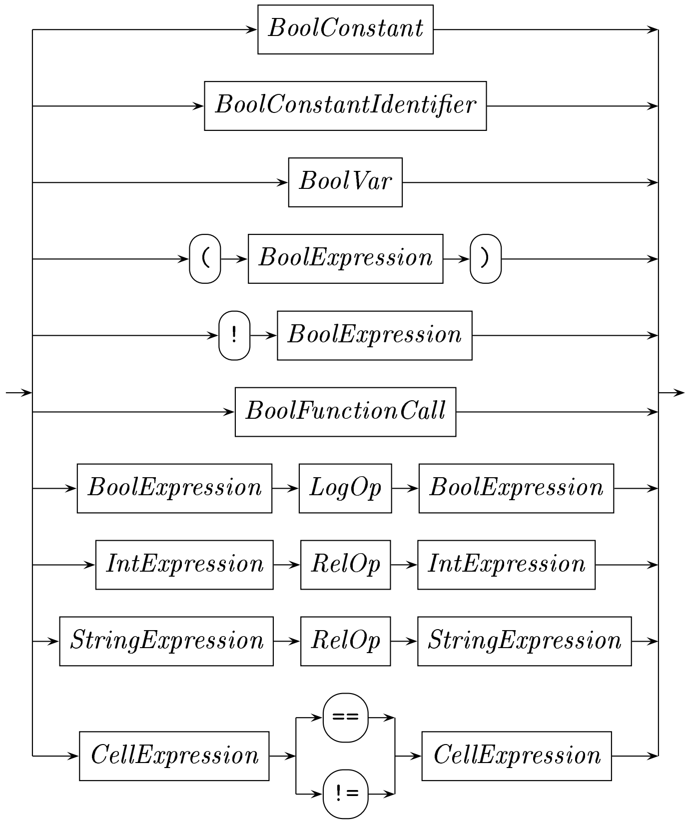
CellExpression (RDL):



Da es zuweilen auch in SDL nötig sein wird, das Auftreten von Nachbarn in Zellausdrücken zu verbieten, betrachten wir dort die Menge *StrictCellExpression* mit gleicher Syntax und Bedeutung wie *CellExpression* in RDL. **StrictCell-Expression**

2.8.4 Boolesche Ausdrücke *BoolExpression*

Syntax:



Wie schon für die anderen Gruppen, so gilt auch hier als Faustregel: Alle Objekte, die boolesche Wahrheitswerte repräsentieren, sind boolesche Ausdrücke. Konkreter können wir wieder zwischen elementaren und zusammengesetzten Ausdrücken differenzieren. Zur ersten Kategorie zählen

Elementare Ausdrücke

- boolesche Konstanten (*BoolConstant*)
- boolesche Konstantenbezeichner (*BoolConstantIdentifier*)
- boolesche Variablen (*BoolVar*)

Sie wurden in den vorangegangenen Abschnitten bereits erläutert.

Boolesche Werte erhalten wir jedoch auch infolge von Vergleichen zwischen Ausdrücken anderer Typen. Auch sie haben wir im Abschnitt über Operatoren bereits besprochen, wollen sie aber hier in syntaktischer Form im einzelnen noch einmal auflisten:

- $IntExpression_1$ *RelOp* $IntExpression_2$
- $StringExpression_1$ *RelOp* $StringExpression_2$

Hierbei steht *RelOp* für die Vergleichsoperatoren `==`, `!=`, `<`, `<=`, `>` und `>=`.
Da Zellausdrücke nur bedingt miteinander verglichen werden können, erhalten wir für sie nur die folgenden beiden Möglichkeiten

- $CellExpression_1$ `==` $CellExpression_2$
- $CellExpression_1$ `!=` $CellExpression_2$

Soweit die Darstellung der elementaren booleschen Ausdrücke.

Zusammengesetzte Ausdrücke

Zunächst halten wir fest, daß wir durch Klammerung eines booleschen Ausdrucks wieder einen solchen erhalten:

(*BoolExpression*)

Die Klammer übernimmt hier dieselbe Funktion wie im Rahmen von numerischen Ausdrücken: Trifft der Compiler während des Auswertungsvorgangs auf einen durch eine Klammer eingefaßten Ausdruck, so wird er ungeachtet der durch die Operatorenrangfolge festgelegten Abarbeitungsfolge diesen vorrangig auswerten.

Boolesche Ausdrücke können nun durch Verknüpfung mit den logischen Operatoren `&&`, `||` sowie durch die logische Negation `!` zu komplexeren Ausdrücken desselben Typs weiterverarbeitet werden:

$BoolExpression_1$ `&&` $BoolExpression_2$
$BoolExpression_1$ `||` $BoolExpression_2$

`!`*BoolExpression*

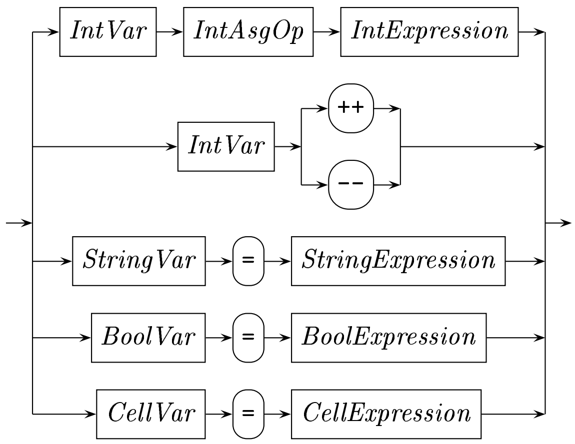
Auch sie wurden bereits beschrieben.

Bemerkung:
Treten Vergleiche und logische Verknüpfungen in demselben Ausdruck auf, so erkennt der Compiler die Vergleiche auch ohne entsprechende Klammersetzung als boolesche Ausdrücke, da die Vergleichsoperatoren einen höheren Rang als `&&` und `||` besitzen.
Anders verhält es sich bei der logischen Negation: Ihr Rang übersteigt den der Operatoren aus *RelOp* und *LogOp*. Soll also insbesondere das Ergebnis einer Vergleichsoperation negiert werden, so <u>muß</u> der Vergleich zwingend in Klammern stehen. $\boxed{!}$
Außerdem weisen wir darauf hin, daß der Rang von `&&` höher ist als der von `||`. (Eine detaillierte Auflistung für alle Operatoren finden Sie im Abschnitt 'Assoziativität und Rangfolge' (S. 43).)
Beispiele für boolesche Ausdrücke finden Sie u.a. in Beispiel 15, S. 44.

2.9 Wert-Zuweisungen bei Variablen

VarAssignment ;

Syntax:



2.9.1 Allgemeine Form

Unter einer Wert-Zuweisung an eine Variable verstehen wir eine Belegung des für die Variable reservierten Speicherplatzes durch eben diesen Wert. (Dabei wollen wir uns nicht um dessen interne Darstellung kümmern.) Die frühere Belegung der betreffenden Variablen wird dadurch überschrieben.
Die allgemeine Syntax für eine solche Aktion lautet[3]

$$\textit{Var} = \textit{Expression} \ ;$$

Der Variablen *Var* wird also der Ausdruck *Expression* zugewiesen, wobei beide von einem der Typen `INT`, `STRING`, `BOOL` oder `CELL` sein können. Entscheidend ist lediglich, daß Variablen- und Ausdrucks-Typen innerhalb einer Zuweisung identisch sind. (Der Typ einer Variablen wird bereits zu Beginn eines Programms fixiert. Daher wird *Var* - je nach Typ - im obigen Syntaxdiagramm durch *IntVar*, *StringVar*, ... ersetzt.)

[3]Formal ist das Semikolon kein Bestandteil von *VarAssignment*, sondern gehört übergeordneten Grammatikteilen an, auf die wir aber nicht namentlich eingehen werden.

Bsp.32: Folgende (nicht unbedingt Zellularautomaten-spezifische) Spielerei soll den Umgang mit verschiedenen Variablentypen noch ein bißchen illustrieren. - Die dargestellten Programmausschnitte könnten sowohl Teil eines RDL- als auch eines SDL-Programms sein:

```
      ...
VAR INT Summe;
    CELL Fischbestand, Sonstige, Gesamtbestand;
    BOOL ok;
REGISTER STRING Art;     /* Stringregister Art */
         INT Anzahl;     /* Int-Register Anzahl */
      ...
Fischbestand = {"Karpfen", 23};
Sonstige = {"Wasserflöhe", 1842103};
Summe = Fischbestand.anzahl;
Summe = Summe + Sonstige.anzahl;
Gesamtbestand = {"Teichbewohner", Summe};
ok = Gesamtbestand.anzahl == 1842126;
      ...
```

2.9.2 Kurzschreibweisen

Bleiben wir in der obigen Notation, so ist es insbesondere gestattet, daß der Name der Variablen auf der linken Seite der Zuweisung auch auf der rechten Seite auftritt, daß also der neue Wert einer Variablen durch Transformation des alten Wertes zustande kommt.
Speziell bei numerischen Variablen beobachten wir häufig Zuweisungen der folgenden Bauart, wobei *IntVar* in diesem Fall zweimal für dieselbe Variable steht:

IntVar `=` *IntVar IntOp IntExpression* `;`

Genau wie in C stellt SCARLET hierfür abkürzende Schreibweisen zur Verfügung. Allgemein lauten sie:

IntVar IntOp`=` *IntExpression* `;`

Dabei darf zwischen *IntOp* (also `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `|`) und `=` kein Leerzeichen stehen. Wir bezeichnen die Menge dieser zusammengesetzten Zuweisungsoperatoren inklusive des einfachen Gleichheitszeichens auch als *IntAsgOp*. ***IntAsgOp***

Bsp.32: (Fortsetzung)
Statt `Summe = Summe + Sonstige.anzahl;` hätte man auch schreiben können:

```
Summe += Sonstige.anzahl;
```

Die Situation nach dem (für die Karpfen fairen) Mittagessen:

```
Gesamtbestand.anzahl -= (Sonstige.anzahl / 23) * 23;
Sonstige.anzahl %= 23;
```

Inkrement, Dekrement

Für die einfache In- bzw. Dekrementation numerischer Variablen sind sogar eigene Operatoren `++` bzw. `--` im Sprachumfang enthalten:

IntVar `++` ;
IntVar `--` ;

Im ersten Fall wird `1` zum Wert von *IntVar* hinzuaddiert; im zweiten `1` subtrahiert.

Bemerkung:
! Anders als in C wird die In- bzw. Dekrementbildung in ihrer Kurzform von SCARLET streng als eigenständige Zuweisung angesehen; sie kann also in keinem Fall als Bestandteil von *Expression* in Erscheinung treten!

Bsp.32: (Fortsetzung)
Einer hat sich dabei wohl totgefressen...:

```
Fischbestand.anzahl--;
```

2.9.3 Die Sonderstellung von `VOID`

Die zusammengesetzte Struktur von Zellobjekten beschert uns ein kleines Kuriosum: den Wert `VOID` als zulässigen Registerinhalt (nicht zu verwechseln mit dem String `"VOID"`).
Tatsächlich ist `VOID` kein String und in seinem Auftreten auch nicht an Register eines bestimmten Typs gebunden. Trotzdem wird man ihn lediglich in Registern von Zellkonstanten und Zellkonstantenbezeichnern ausmachen können; in Variablen sowie Zellen der Retina oder der Grenzschicht findet er dagegen keinen Zugang, denn:

Wann immer zwei Zellausdrücke durch einen Operator zueinander in Beziehung gesetzt werden, werden alle Register mit Inhalt `VOID` von der Betrachtung ausgeschlossen. Konkret heißt das:

- Steht `VOID` auf der rechten Seite einer Zuweisung, so wird der Wert des zugehörigen Zielregisters nicht verändert.
- Werden zwei Zellausdrücke auf Gleichheit geprüft, so liefern alle Register mit Inhalt `VOID` `TRUE`, beim Test auf Ungleichheit FALSE [4].

Da sowohl die Zellen als auch Zellvariablen ihre Inhalte stets über eine Zuweisung erhalten, ist klar, weshalb deren Register niemals `VOID` enthalten können.
Zellkonstantenbezeichner dagegen schon, da `VOID` als konstanter Wert die Auswertung der einzelnen Registerausdrücke unverändert passiert und das Gleichheitszeichen hier nicht als Zuweisungszeichen fungiert. Zum Verständnis dieser letzten Aussage sei zusätzlich auf Abschnitt 3.1.4, S. 73 verwiesen. Dort werden die Vorgänge bei der Deklaration von Konstantenbezeichnern genauer beschrieben.

Bsp.33:

```
RETINA;

RANGE [0]..[10];
REGISTER INT     r1;
         STRING  r2;
CONST cellconst_1 = {VOID, "ready"};
      cellconst_2 = {0, VOID};
VAR STRING stringvar;
    CELL   cellvar;
    BOOL   boolvar;
    INT    intvar;

START
cellvar = {0, ""};
boolvar = cellvar == cellconst_2;               (= TRUE)
boolvar = cellconst_1 != {25, VOID};            (= FALSE)
cellvar.r1 ++ ;
cellvar = cellconst_1;
boolvar = cellvar == {0, "ready"};              (= FALSE)
cellvar = cellconst_2;
boolvar = cellvar == {0, "ready"};              (= TRUE)
```

[4] Die allgemeine Beschreibung der Wirkung von `VOID` war hier also nicht ganz exakt: Werden nämlich alle Register mit `VOID` verglichen, so liefert sie keine Aussage mehr

! Wichtig: `VOID` ist weder vom Typ `INT`, noch vom Typ `STRING`. Soll also ein Registerinhalt als String oder als Zahl interpretiert und weiterverarbeitet werden, so muß sichergestellt sein, daß es sich dabei nicht um `VOID` handelt. (Diese Bedingung ist aber leicht zu erfüllen, da der Registerinhalt `VOID` nur in expliziter Form im Programmtext oder aber in Konstantenbezeichnern auftreten kann, und somit auch keiner unkontrollierten Ausbreitung unterliegt. Letztlich muß somit nur der Zugriff auf Konstantenbezeichner abgesichert werden, da die Extraktion einzelner Registerinhalte aus explizit im Text auftretenden Zellkonstanten sowieso nicht möglich ist.)

Bsp.34: Im vorangegangenen Beispiel wären z.B. folgende Aktionen <u>verboten</u>:

```
stringvar = cellconst_2.r2;                    (= VOID)
intvar = abs(cellconst_1.r1) + 42;
             \_____________/
                  =VOID
```

Kapitel 3

RDL - Retina Definition Language

Nachdem wir nun alle grundlegenden Bausteine für die Konstruktion von RDL- und SDL-Programmen zusammengetragen haben, wollen wir nun daran gehen, die solchen Programmen zugrundeliegende *Struktur* zu verstehen. Dabei müssen wir nun selbstverständlich zwischen RDL und SDL unterscheiden. Wir wenden uns in diesem Abschnitt zunächst RDL zu:

Wie bereits gesagt, ist es das Ziel eines RDL-Programms, die Retina eines Zellularautomaten, d.h. seine Ausdehnung im $\mathbb{Z}^n$ (für ein konkretes $n \in \mathbb{N}$) festzulegen, sowie eine gültige Startkonfiguration zu bilden. Zu diesem Zweck müssen selbstverständlich auch Aussagen über die Zustandsmenge des Automaten gemacht werden, was bei der SCARLET zugrundeliegenden Implementierung nichts anderes heißt, als Anzahl, Reihenfolge und Typus der Register zu bestimmen.

Im Gegensatz dazu spielt der Nachbarschaftsindex in RDL keine Rolle. Er existiert nur in SDL. **Folglich dürfen in RDL niemals Nachbarn zur Bildung von Ausdrücken verwendet werden.** Konkret heißt das:

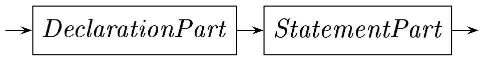
Keine Nachbarn

- *CellExpression* verbietet den Gebrauch von *NeighbourIdentifier* (vgl. S. 58),
- für *IntExpression* ist die Konstruktion *NeighbourIdentifier.IntRegIdentifier* untersagt,
- und *StringExpression* verbietet die Verwendung von *NeighbourIdentifier.StringRegIdentifier*, (vgl. S. 53).

Formal gesehen untergliedert sich ein RDL-Programm (ähnlich wie in Pascal) in einen Vereinbarungs- und einen Anweisungsteil:

Retina

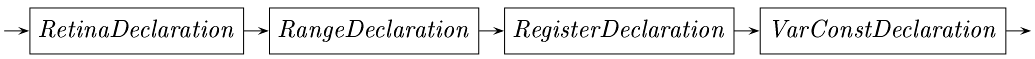
Retina:



3.1 Der Vereinbarungsteil *DeclarationPart*

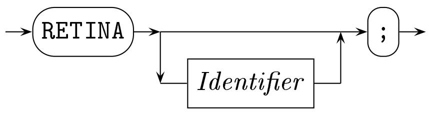
Der Vereinbarungsteil seinerseits enthält nun eine Programmstart-Markierung sowie diverse Angaben über das Ausmaß der Retina, über verwendete Register, Variablen und Konstanten. Die im Syntaxdiagramm angegebene Reihenfolge der Deklarationsblöcke muß dabei unbedingt eingehalten werden:

Syntax:



3.1.1 Der Programmbeginn *RetinaDeclaration*

Syntax:



Um ein RDL-Programm als solches kenntlich zu machen, beginnen wir es stets mit dem Schlüsselwort

```
RETINA ;
```

An dieser Stelle wird bereits eine wesentliche und leicht zu merkende Regel beim Erstellen von RDL- (und SDL-)Programmen sichtbar:

> **Alle Vereinbarungs- und Anweisungseinheiten werden stets mit Semikolon abgeschlossen!**

Es ist darüber hinaus möglich, dem Programm einen beliebigen Namen zu geben, der z.B. der persönlichen Erinnerung an die wesentlichen Eigenschaften der nachstehenden Retina dienen könnte. Darüberhinaus wird SCARLET eben diesen Namen anzeigen, wenn das Programm später zum Zwecke einer Simulation geladen wird. In diesem Fall würde die Programmeröffnung folgendermaßen lauten:

`RETINA` *Identifier*;

Alle nachfolgenden Vereinbarungen werden nun in der Reihenfolge ihres Erscheinens innerhalb des Programmtextes sukzessive beschrieben.

3.1.2 Die Bereichsvereinbarung *RangeDeclaration*

Die erste wichtige Handlung besteht nun in der Festlegung des Retina-Bereichs. Wie bereits mehrfach erwähnt, stellen wir uns diesen als endlichen Auschnitt des $\mathbb{Z}^n$ vor, der in SCARLET speziell Quaderform annimmt. Jeder Koordinate innerhalb dieses Bereichs stellen wir uns eine Zelle des Zellularautomaten zugeordnet vor. Eine Koordinate wird in SCARLET dargestellt durch

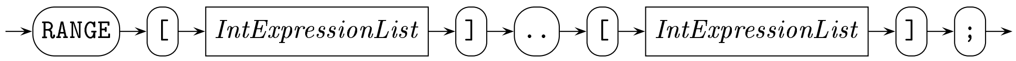
`[` *IntExpression*$_1$, *IntExpression*$_2$, ... , *IntExpression*$_n$ `]`

oder kurz:

`[` *IntExpressionList* `]`

Dies entspricht der gewohnten Schreibweise für n-dimensionale Vektoren. Die Quaderform der Retina ergibt sich dann wie automatisch durch die folgende Syntax der Bereichsdefinition:

Syntax:



Der Quader wird also festgelegt durch die Angabe zweier „Eckkoordinaten“. Vergleicht man diese komponentenweise, so erhält man dessen Ausdehnung in den einzelnen Koordinatenrichtungen. Die Ecken selbst sind dabei immer zur Retina dazugehörig. Sie kann also in keiner Koordinatenrichtung die Ausdehnung Null besitzen (womit sie zu einer „verschwindenden“ Menge würde), sondern enthält stets mindestens eine Zelle. Eine solche Syntax setzt natürlich voraus, daß *IntExpressionList*$_1$ und *IntExpressionList*$_2$ die gleiche Länge haben. Machen Sie sich klar, daß durch diese Länge implizit auch die **Dimension**

Dimension der Retina festlegt wird!

Bemerkung:
Da die Bereichsdefinition ganz zu Anfang erfolgt, stehen Ihnen für *IntExpression* an dieser Stelle noch keine Variablen oder Konstantenbezeichner zur Verfügung, da diese vor ihrer Verwendung deklariert werden müssen.

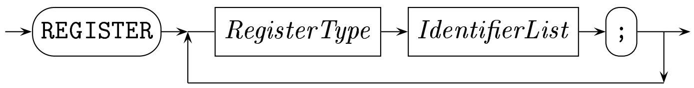
Bsp.35: `RANGE [20]..[0];` definiert eine 1-dimensionale Retina bestehend aus den Zellen mit den Koordinaten 0 bis 20. - Die erste Eckenangabe muß nicht zwingend die kleineren Koordinaten enthalten!

`RANGE [-3,2,5]..[3,4,5];` definiert eine Retina im 3-dimensionalen Raum. Die zweimalige Angabe von `5` in den dritten Koordinatenrichtung bewirkt, daß die Retina nur „eine Zelle dick“ ist, also eigentlich die Eigenschaften einer 2-dimensionalen Retina besitzt.

3.1.3 Register-Deklaration *RegisterDeclaration*

Als nächstes müssen nun zwingend die zu verwendenden Register angegeben werden.

Syntax:



Der Programmteil startet für den Compiler verständlich mit dem Schlüsselwort

`REGISTER`

und wird gefolgt von einer Aufzählung der einzelnen Registernamen samt ihrer Typen. Ein einzelnes Element daraus besitzt (zunächst grundsätzlich) folgenden syntaktischen Aufbau:

RegisterType Identifier ;

Hierbei kann *RegisterType* wahlweise `INT` oder `STRING` sein und gibt damit an, ob ein numerisches oder ein String-Register geschaffen werden soll.
Die gesamte Aufzählung besteht nun aus einer beliebig langen Folge solcher Elemente. (Sinnvollerweise müssen Sie jedoch mindestens ein Register deklarieren.) Versehen wir sie (in Gedanken) bei Null beginnend von oben nach

Anordnung unten mit einer Numerierung, so wird mit einem Mal eine Vektorstruktur auf die Register induziert. Dieses Vorgehen mag zunächst einmal überflüssig erscheinen, da Mächtigkeit und qualitative Eigenschaften der repräsentierten Zustandsmenge (als Kreuzprodukt) unter einer sich ändernden Reihenfolge invariant bleiben. Um jedoch in Zukunft bei Zellausdrücken auf die Angabe der einzelnen Registernamen verzichten zu können, erweist es sich als äußerst praktisch, lediglich die Registerinhalte in der hier vereinbarten Reihenfolge als Vektor wiedergeben zu können. Das Register mit der Nummer Null steht dabei ganz links, es folgt Nummer Eins, usw..
Zum Schluß noch eine vereinfachende Schreibweise: Folgen mehrere Register gleichen Typs direkt aufeinander, so kann **_RegisterList_**

$$\begin{array}{c} \textit{RegisterType Identifier}_1 \ ; \\ \textit{RegisterType Identifier}_2 \ ; \\ \cdots \\ \textit{RegisterType Identifier}_N \ ; \end{array}$$

ersetzt werden durch

$$\textit{RegisterType Identifier}_1 \ , \ \textit{Identifier}_2 \ , \ \cdots \ , \ \textit{Identifier}_N \ ;$$

Bsp.36: Die Konstante {`1, "1. String", 2, 3`} paßt etwa zu folgender Registerdeklaration:

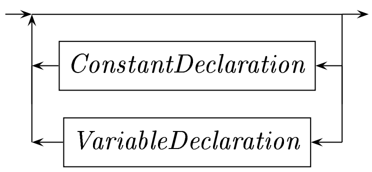
```
REGISTER INT first;
         STRING second;
         INT third, fourth;
```

3.1.4 Konstantenbezeichner- und Variablen-Deklaration

VarConstDeclaration

Der verbleibende Rest des Vereinbarungsteils ist nun der Deklaration von Konstantenbezeichnern und Variablen gewidmet:

Syntax:



Er kann ganz oder in Teilen entfallen, falls Sie ohne solche Strukturen auskommen. Weiterhin bleibt es im wesentlichen ihrer Phantasie überlassen, auf wieviele Blöcke Sie Ihre Variablen und Konstantenbezeichner verteilen wollen und in welcher Reihenfolge Sie diese anlegen. Sie müssen lediglich dafür Sorge tragen, daß die evtl. bei einer Konstantenbezeichner-Deklaration verwandten Objekte vorher bereits deklariert wurden.

Variablen *VariableDeclaration*

Haben Sie sich dazu entschlossen, in Ihrem Programm Variablen zu verwenden, so müssen Sie diese zunächst deklarieren:

Syntax:

Ein solcher Deklarationsblock startet wie oben angegeben stets mit dem Schlüsselwort

`VAR`

Die eigentlichen Einträge haben nun die Gestalt

VariableType IdentifierList ;

Hierbei kann *VariableType* die Werte `INT`, `STRING`, `BOOL` oder `CELL` annehmen, und *IdentifierList* beinhaltet eine Reihe von Namen für Variablen, die vom angegebenen Typ sein sollen.
Sie können beliebig viele dieser Einträge aneinanderreihen.

Initialisierung

Die auf diese Art vereinbarten Variablen werden von SCARLET zunächst initialisiert: Numerische erhalten den Wert `0`, boolesche den Wert `TRUE` und String-Variablen beinhalten nach der Initialisierung den leeren String `""`. Die Initialisierung von Zellvariablen erfolgt, indem mit den einzelnen Registern wie mit numerischen oder String-Variablen verfahren wird.

Bsp.37:

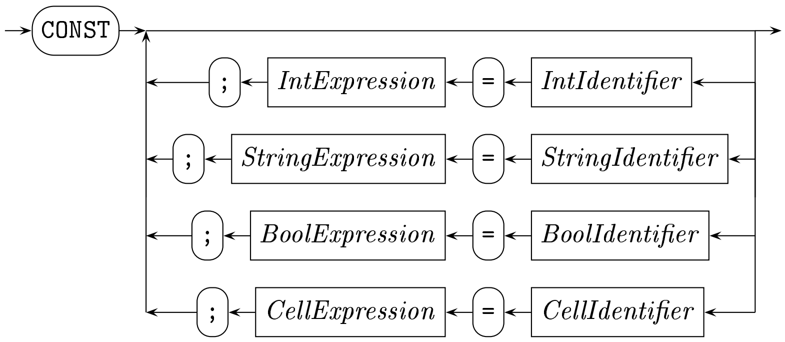
```
     ...
VAR INT counter;
    STRING name1, name2;
    INT result;
VAR BOOL is_enough;
    CELL a_state;
     ...
```

Durch den Aufbau von *VarConstDeclaration* bedingt kann `VAR` beliebig oft auftauchen.

Konstantenbezeichner

ConstantDeclaration

Syntax:



Konstantenbezeichner-Deklarationen werden in SCARLET eingeleitet mit dem Schlüsselwort

`CONST`

Es folgen beliebig viele Einträge der Form

Identifier = *Expression* ;

Dabei ist *Identifier* der Name des zu vereinbarenden Konstantenbezeichners (der je nach Typ der Konstanten auch mit *IntIdentifier*, *StringIdentifier*, ... bezeichnet wird).
Bei dem Ausdruck auf der rechten Seite des Gleichheitszeichens handelt es sich um ein Element der Klassen *IntExpression*, *StringExpression*, *BoolExpression* oder *CellExpression*. Die Verwendung von Variablen ist nur dann gestattet, wenn diese zuvor bereits deklariert wurden.

Auswertung

Bei Programmaufruf werden zunächst einmalig die den einzelnen Konstantenbezeichnern zugeordneten Ausdrücke in der Reihenfolge ihres Auftretens zu konstanten Ausdrücken ausgewertet und intern gespeichert: Durch die Auswertung bleiben konstante Ausdrücke unverändert, für Konstantenbezeichner wird der zuvor ermittelte Wert verwendet und Variablen werden durch ihre aktuelle Belegung ersetzt (also stets durch ihren Initialisierungswert). Ausdrücke mit Funktionsaufrufen und Operatoren werden nach den entsprechenden Regeln vereinfacht, und die Auswertung der Registerausdrücke innerhalb eines Zellausdrucks erfolgt schließlich von links nach rechts.

Anmerkung:
`VOID` Da es sich bei `VOID` um einen konstanten Ausdruck handelt, bleibt es als Registerinhalt in dieser Form erhalten (vgl. Bsp.33).

Eine auf diese Art entstandene Konstante wird Ihnen dann unter dem Namen *Identifier* zur Verfügung gestellt, d.h., wann immer im nachfolgenden Programmtext ein dieser Konstantenbezeichner auftaucht, wird er durch den zuvor für ihn ermittelten konstanten Ausdruck ersetzt. Durch den Vorgang der Deklaration werden Konstantenbezeichner selbst zu einem Objekt von dem durch *Expression* vorgegebenen Typ.

Bsp.38:

```
      ...
REGISTER INT num;
CONST max = 2147483647;
      a_number = rand(0,100);
      is_greater = a_number > 50;
VAR CELL a_cellvar;
CONST initialized_cell = a_cellvar;                    (={0})
      crypt = "<*@_"+ itos(a_number);
      ...
```

3.2 Der Anweisungsteil

Halten wir kurz fest, was wir durch die Angaben im Vereinbarungsteil des RDL-Programms im wesentlichen erreicht haben: Wir sind nun in der Lage, den Zellularautomaten in einem Raum mit von uns gewünschter Dimension zu lokalisieren, d.h. konkreter: Wir können die Zellen seiner Retina (kurz: Zellen) in eindeutiger Weise mit gewissen Raumkoordinaten identifizieren. Weiterhin haben wir uns bereits auf eine grobe, der Zustandsmenge zugrunde liegende Struktur geeinigt; welche konkreten Zustände die einzelnen Zellen vor Simulationsbeginn haben sollen, haben wir bisher allerdings noch nicht festgelegt.

Um die Bildung einer solchen Startkonfiguration geht es nun im Anweisungsteil des RDL-Programms.

3.2.1 Initialisierung der Retina

Genau wie Variablen (und Konstantenbezeichner) werden auch alle Zellen von SCARLET initialisiert, d.h. alle numerischen Register erhalten den Wert `0`, alle String-Register den leeren String `""`. Mit anderen Worten wird also von SCARLET eine Standard-Startkonfiguration vorgegeben, die aufgrund ihrer großen Homogenität allerdings kein besonders aufregendes Verhalten des Zellularautomaten nach sich ziehen dürfte. Es wird demnach unser Ziel sein, einzelnen Zellen oder ganzen Mengen von Zellen andere Zustände zuzuweisen.

Zu diesem Zweck klären wir als erstes, um welche Mengen es sich dabei genau handelt, und auf welche Art wir solche Objekte syntaktisch korrekt charakterisieren können:

3.2.2 Zellen *Cell*

Wie schon gesagt stellen wir uns alle für den Zellularautomaten relevanten Zellen in einem endlichen Ausschnitt des $\mathbb{Z}^n$ liegend vor. Dieser Bereich (die Retina) wurde im Vereinbarungsteil bereits festgelegt. Und genau wie dort sprechen wir auch hier einzelne Zellen daraus an, indem wir ihre Position im Raum (Koordinate) als n-dimensionalen Vektor in der folgenden Art angeben:

Syntax:

→ `[` → *IntExpressionList* → `]` →

Selbstverständlich muß eine solche Positionsangabe innerhalb der Retina liegen. Insbesondere muß *IntExpressionList* aus n Elementen *IntExpression* bestehen.

Bsp.39: Wir betrachten für die 3-dimensionale Retina, festgelegt durch `RANGE [-5,0,7]..[9,2,-1];` folgende Beispiele für Zellen:

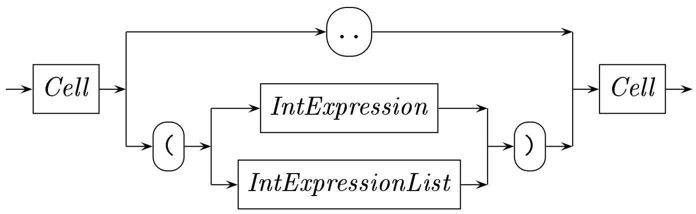
$\surd$	`[-5,0,7]`	(richtig)
$\times$	`[-5,0,8]`	(falsch)
$\surd$	`[abs(-7),0,0]`	(richtig)

3.2.3 Das Zellgitter

Loop

Prinzipiell könnten wir die gesamte Startkonfiguration nun bilden, indem wir allen Zellen einzeln den von uns gewünschten Zuständen entsprechende Zellausdrücke zuweisen. In der Praxis wird es aber oft so sein, daß ganze Gruppen von Zellen zumindest in Bezug auf einzelne Register die gleiche Behandlung erfahren sollen. SCARLET verfügt daher über eine Möglichkeit, das obige Verfahren zu verkürzen und Schreibarbeit zu sparen, indem „relativ einfach“ gebaute Mengen von Zellen zusammen betrachtet werden können.

Syntax:



Die solchen Mengen zugrundeliegende Form ist der Quader, dessen Ausdehnung hier (wie bei der Retina-Definition) durch Angabe zweier sich gegenüberliegender Ecken *Cell* bestimmt wird (siehe S. 69). Er muß vollständig im Bereich der Retina enthalten sein.

Bemerkung:
Dazu reicht es aus, darauf zu achten, daß die angegebenen Eckkoordinaten innerhalb der Retina liegen.

Schrittweite

Wir werden von SCARLET jedoch nicht dazu gezwungen, alle Zellen des Quaders in unsere Menge aufzunehmen, sondern wir können uns auf solche beschränken, die zusätzlich auf den Schnittpunkten eines vorgegebenen Gitters liegen. Ein solches Gitter erzeugen wir durch die Angabe von konstanten Schrittweiten in den einzelnen Koordinatenrichtungen. Im allgemeinsten Fall lautet die Syntax für ein solches Objekt also folgendermaßen:

$$\mathit{Cell}_1\ (\mathit{IntExpressionList})\ \mathit{Cell}_2$$

Hierbei stehen Cell_1 und Cell_2 für zwei geeignete Eckkoordinaten, die gemeinsam den Quader erzeugen, während das Tupel (*IntExpressionList*) die Angaben für die einzelnen Schrittweiten enthält.
Dabei bezieht sich die erste Angabe auf die erste Koordinatenrichtung, die zweite auf die zweite usw. . Wird ein n-dimensionaler Raum betrachtet, so müssen zwingend n Schrittweitenangaben

$$IntExpression_1\,,\; IntExpression_2\,,\; \cdots\;,\; IntExpression_n$$

gemacht werden. Alle Angaben sind nichtnegativ und `0` ist nur dann ein gültiger Wert, falls der Quader in dieser Koordinatenrichtung die Ausdehnung eins besitzt, d.h. falls $Cell_1$ und $Cell_2$ in der betreffenden Vektorkomponente denselben Wert haben.

Es ist darauf zu achten, daß SCARLET die Gittererzeugung bei $Cell_1$ beginnt: Bezeichnen wir die Menge aller Koordinaten des Quaders mit Q, so liegt eine Zelle formal in der durch die obige Syntax bezeichneten Menge, falls es für deren Koordinate $Cell\; m_1, m_2, \cdots, m_n \in \mathbb{Z}$ gibt mit

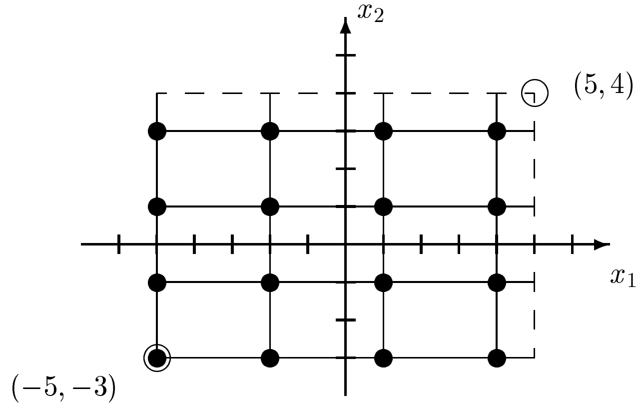
$$Cell = Cell_1 + \begin{bmatrix} m_1 \cdot IntExpression_1 \\ m_2 \cdot IntExpression_2 \\ \vdots \\ m_n \cdot IntExpression_n \end{bmatrix} \in Q$$

Alle in *Loop* auftretenden Ausdrücke werden nur einmal evaluiert. Insbesondere ist die Schrittweite tatsächlich konstant. !

Bsp.40: Für einen 2-dimensionalen Raum erzeugt

```
[-5,-3] (3,2) [5,4]
```

folgende (durch schwarze Punkte gekennzeichnete) Menge:



Für folgende Spezialfälle akzeptiert SCARLET auch Kurzschreibweisen: **Kurzschreibweisen**

- Soll für jede Koordinatenrichtung dieselbe Schrittweite *IntExpression* gelten, so können Sie dies zum Ausdruck bringen durch

$$\mathit{Cell}_1\ (\mathit{IntExpression})\ \mathit{Cell}_2$$

- Ist in diesem Fall noch spezieller $\mathit{IntExpression} = 1$ (ist also der ganze Quader gemeint), so können Sie stattdessen auch schreiben

  $$\mathit{Cell}_1\ ..\ \mathit{Cell}_2$$

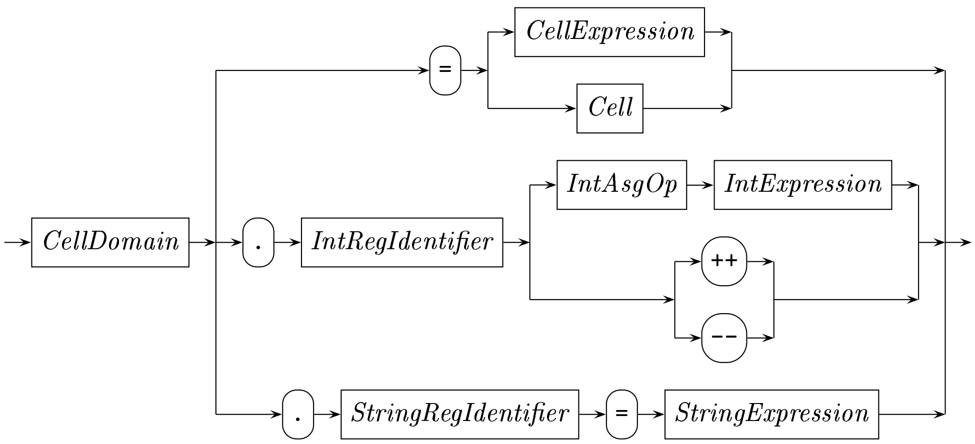
  Hier entfallen die Klammern.

Bsp.41: Die folgenden drei Beispiele beschreiben dasselbe Zellgitter:

```
[0,1,2,3] (1,1,1,1) [4,5,6,7]
[0,1,2,3] (1) [4,5,6,7]
[0,1,2,3]..[4,5,6,7]
```

3.2.4 Wertzuweisungen an Zellen *CellAssignment* ;

Syntax:



Betrachten wir nun die Bestandteile der obigen Syntax im einzelnen: Nachdem wir nun die in SCARLET gebräuchliche Syntax sowohl für einzelne Zellen (*Cell*), als auch für gewisse Mengen von Zellen (*Loop*) kennengelernt haben, ist die uns eigentlich interessierende Zuweisung von (zellwertigen) Ausdrücken an solche Objekte kein Kunststück mehr, da wir sie (nur in Bezug darauf!) syntaktisch wie normale Zellvariablen behandeln.
Fassen wir die Mengen *Cell* und *Loop* zu einer neuen Klasse *CellDomain*
CellDomain zusammen, so lautet die Syntax einfach[1]

[1] Auch hier ist das Semikolon kein eigentlicher Bestandteil von *CellAssignment* (vgl. *VarAssignment*)

CellDomain `=` *CellExpression* ;

Wollen Sie *CellDomain* denselben Zustand zuweisen, den bereits eine andere Zelle *Cell* besitzt, so erreichen Sie dies mit

Wertübertragung

CellDomain `=` *Cell* ;

Beachten Sie, daß alle Ausdrücke auf der rechten Seite einer Zuweisung an eine Menge von Zellen *Loop* für jede Zelle neu ausgewertet werden. Dasselbe gilt auch für den Fall, daß, wie im folgenden beschrieben, nur ein bestimmtes Register der Zellen von *Loop* betrachtet werden soll.

Einzelne Register

Wie schon gesagt ist es auch möglich, lediglich Zuweisungen an einzelne Register der Zellen von *CellDomain* vorzunehmen: Bezeichnen *IntRegIdentifier* bzw. *StringRegIdentifier* vereinbarte Namen für Register vom Typ `INT` bzw. `STRING`, so erhalten Sie wie im Abschnitt 'Ansprechen einzelner Register', S. 50, beschrieben eine Projektion aller Zellen von *CellDomain* auf das betreffende Register durch

CellDomain`.`*IntRegIdentifier*

bzw. durch

CellDomain`.`*StringRegIdentifier*

Im Rahmen von Zuweisungen sind solche Objekte nun syntaktisch wie gewöhnliche Variablen vom Typ `INT` oder `STRING` zu behandeln. Genaueres hierzu finden Sie im Abschnitt 'Wertzuweisungen an Variablen' auf Seite 62.

Beachten Sie, daß es für Projektionen kein Analogon zu

CellDomain `=` *Cell* ;

gibt.

$\boxed{!}$

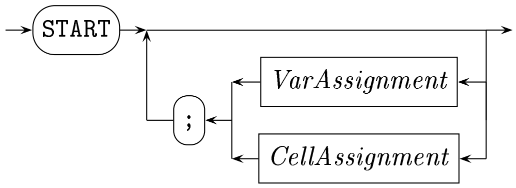
Weiterhin ist es prinzipiell nicht möglich, den Inhalt einer Zelle an eine Variable zu übergeben. Was sich auf den ersten Blick vielleicht wie eine Einschränkung ausnimmt, ist tatsächlich aber keine: Sie können eine solche Zuweisung ganz einfach umgehen, indem Sie alle Zuweisungen an die betreffende Zelle doppelt vornehmen: einmal an die Zelle und parallel dazu an die gewünschte Variable!

Daß dies überhaupt erlaubt ist im Rahmen eines RDL-Programms, zeigt der nächste Abschnitt:

3.2.5 Aufbau des Anweisungsteils

StatementPart

Syntax:



Der Beginn dieses zweiten und letzten Abschnitts eines RDL-Programms, dessen Intention und Spezialitäten wir in den vorangegangenen Abschnitten bereits vorgestellt haben, wird dem Compiler durch das Schlüsselwort

`START`

signalisiert. Es schließen sich beliebig viele Zuweisungen an Variablen

VarAssignment ;

oder Zellen

CellAssignment ;

an.

Abarbeitungsfolge

Alle Zuweisungen werden sequentiell bei `START` beginnend und mit sofortiger Wirkung auf Variablen- und Zellinhalte durch den Compiler abgearbeitet. (Es mag überflüssig erscheinen, dies extra zu erwähnen, denn es handelt sich hierbei um die übliche Abarbeitungsfolge; jedoch werden wir sehen, daß SDL-Programme hier eine Ausnahme bilden...)

Bsp.42: Bei dem folgenden Beispiel handelt es sich um die 2-dimensionale triviale Retina: Alle Zellen enthalten die Initialisierungswerte ihrer Register, hier also {`0,""`}. Das Schlüsselwort `START` muß jedoch trotzdem vorhanden sein.

```
     RETINA initial;
     RANGE [0,0]..[5,5];
     REGISTER INT num;
              STRING str;
     VAR /* könnte auch entfallen */
     CONST /* - */

START
```

Bsp.43: Eine beliebige weitere Retina.

Der letzte Befehl bewirkt, daß jede Zelle in einem bestimmten variablen Gebiet den Wert einer jeweils anderen, zufällig ermittelten Zelle erhält.

```
RETINA modern_art;
RANGE [0,0]..[20,15];
REGISTER STRING point;
VAR
CONST yellow= "*";
      red   = "/";
      green = "+";
      blue  = "-";
START
  [0,0]..[20,10] = {blue};
  [1,5] (3) [20,15] = {green};
  [7,5] (2,1) [15,10].point = red;
  [2,3]..[17,7] = {yellow};
  [10+rand(-2,2),1]..[19,5] = [10+rand(-5,5),7+rand(-3,3)];
```

Kapitel 4

SDL - Sigma Definition Language

Wir wenden uns in diesem Kapitel nun der zweiten Sprachkomponente von Scarlet zu: der sogenannten Sigma Definition Language (SDL). Sie befaßt sich mit der Festlegung der „charakteristischen Verhaltensweise" eines Zellularautomaten, d.h. mit der Definition von (lokalen) Überführungsfunktionen (auch Sigma-Überführungsfunktionen genannt - daher der Name der Sprache - bzw. kurz: σ).
Hierfür ist es zunächst vollkommmen ausreichend, Angaben über

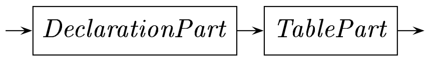
- die Dimension des Raumes,
- die Zustandsmenge (d.h. Art und Anzahl der zu verwendenden Register)
- die Nachbarschaftsbeziehungen der Zellen untereinander
- den gemeinsamen Zustand aller Zellen der Grenzschicht

zu machen; die genaue Ausdehnung der Retina im Raum, oder gar deren Konfiguration vor Simulationsbeginn sind in diesem Zusammenhang irrelevant; Vereinbarungen dieser Art werden in Scarlet daher (wie im letzten Kapitel beschrieben) in einer separaten Datei als RDL-Programm festgehalten.
Erst sehr viel später entsteht ein kompletter Zellularautomat durch Verknüpfung je eines kompilierten RDL- und SDL-Programms, vorausgesetzt, daß diese bestimmten Kompatibilitätsbedingungen genügen, auf die wir dann im letzten Abschnitt dieses Kapitels noch zu sprechen kommen.

Doch vorerst betrachten wir den Aufbau eines SDL-Programms genauer. Formal untergliedert sich auch dieses in einen Deklarations- und einen Anweisungsteil:

Sigma

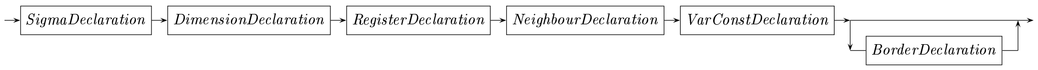
Sigma:



4.1 Der Vereinbarungsteil

DeclarationPart

Syntax:

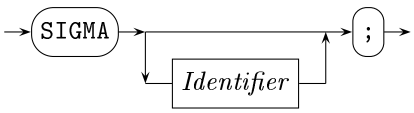


Der Vereinbarungsteil besteht aus einer Folge verschiedener Angaben (zu Raumdimension, Registern, Nachbarn, Variablen und Konstantenbezeichnern sowie Randwerten), die im wesentlichen in der hier angegebenen Reihenfolge im Programm erscheinen müssen. Ausnahmen bilden die Konstantenbezeichner- und Variablen-Blöcke in *VarConstDeclaration* sowie die Randwertbehandlung. Beides sind optionale Bestandteile. Die Variablen- und Konstantenbezeichner-Deklarationen können darüber hinaus untereinander in beliebiger Reihenfolge angeordnet werden.

4.1.1 Die Programmbeginn

SigmaDeclaration

Syntax:



Ein SDL-Programm beginnt mit

`SIGMA` *Identifier* ;

oder einfach nur

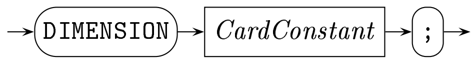
`SIGMA ;`

da die Angabe eines Programm-Namens nicht zwingend erforderlich ist (vgl. RDL, S. 67).

4.1.2 Die Raumdimension *DimensionDeclaration*

Zwingend erforderlich ist nun als erstes die Spezifikation des Grundraumes. Was sich hinter dieser hochgestochenen Formulierung verbirgt, ist jedoch nichts anderes, als die Angabe der Dimension für den $\mathbb{Z}^n$, $n \in \mathbb{N}$, in dem das Modell des (Standard)-Zellularautomaten per definitionem angesiedelt ist!

Syntax:



Dabei muß *CardConstant* als (positive!) Dimension ein vorzeichenloses Element ungleich Null aus *IntConstant* sein.

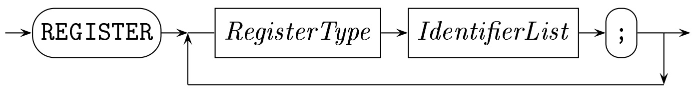
Dim. = 0

Bemerkung:
Aus verschiedenen Gründen wird Dimension 0 hier nicht zugelassen. Trotzdem kann der „punktförmige“ Zellularautomat, der nur aus einer einzigen Zelle umgeben von Grenzzellen besteht, durch räumliche Einschränkung eines höherdimensionalen Zellularautomaten auf diese eine Zelle erzeugt werden.

4.1.3 Register-Deklaration *RegisterDeclaration*

Ebenfalls unumgänglich ist nun die Vereinbarung einer für alle Zellen des Automaten (und damit auch für sonstige Zellobjekte) verbindlichen Registerstruktur. Die Vorgehensweise hierfür unterscheidet sich nicht von der für RDL-Programme.

Syntax:

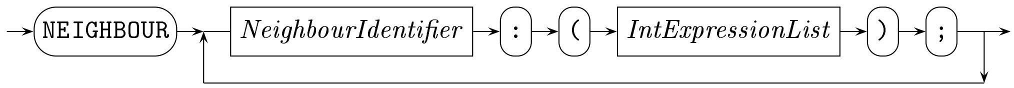


Wir verzichten deshalb an dieser Stelle auf eine Erläuterung des Syntaxdiagramms und verweisen stattdessen nur auf Abschnitt 3.1.4, S. 71.

4.1.4 Der Nachbarschaftsindex *NeighbourDeclaration*

Ganz zu Anfang, im ersten Kapitel, haben wir bereits gesehen, daß sich Zellularautomaten grundsätzlich aus einer Menge gleichartiger (Moore-) Automaten zusammensetzen, den Zellen. Im Unterschied zu gewöhnlichen Automaten ist deren Überführungsfunktion jedoch nicht von einer Eingabe „von außen“ abhängig, sondern vielmehr von den Zuständen gewisser Nachbarzellen. Um welche Zellen es sich dabei handelt, wird durch den Nachbarschaftsindex geregelt: Da wir uns alle Zellen der Retina als gleichartig vorstellen wollen, ist es naheliegend, auch eine für alle Zellen einheitliche „Umweltstruktur“ zu fordern. Folglich werden deren Nachbarn im Nachbarschaftsindex auf allgemeine Art durch die Angabe ihrer relativen Positionen zu ihrer jeweiligen Bezugszelle festgelegt (vgl. Kapitel 1).

Syntax:



Die Syntax dieses unbedingt erforderlichen Programmbestandteils wird eingeleitet mit

`NEIGHBOUR`

Es folgt mindestens eine Nachbarschafts-Eintragung der folgenden Gestalt

NeighbourIdentifier `:` `(` *IntExpressionList* `)` `;`

Ist n die Raumdimension (S. 84), so müssen für *IntExpressionList* genau n Ausdrücke *IntExpression* vorhanden sein. Da die Deklaration von Variablen und Konstantenbezeichnern erst im Anschluß hieran erfolgt, dürfen Sie für *IntExpression* keine Objekte aus *IntVar*, *IntConstantIdentifier* und *IntReg* verwenden.

Während *NeighbourIdentifier* also einen frei wählbaren Namen für diese Art von Nachbarschaft darstellt, gibt `(` *IntExpressionList* `)` die relative Position zur Bezugszelle an, oder präziser ausgedrückt:
Ist K die Koordinate einer beliebigen Zelle, so erhält man die Koordinate N ihres Nachbarn bezüglich der Nachbarschaftsart *NeighbourIdentifier* durch die Beziehung [1]

$$N = K + \begin{pmatrix} \mathit{IntExpression}_1 \\ \mathit{IntExpression}_2 \\ \vdots \\ \mathit{IntExpression}_n \end{pmatrix}$$

[1] Die Definition zeigt, daß es eigentlich sinnvoller wäre, statt von „Nachbar“ von „Beeinflusser“ zu sprechen, da wir uns üblicherweise unter Nachbarschaft eine symmetrische Beziehung vorstellen, die hier nicht gegeben ist.

Bsp.44: Für das 2-dimensionale verallgemeinerte von-Neumann-Raster H_1^2 aus Beispiel 3 erhalten wir

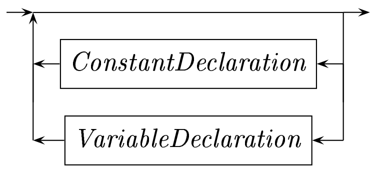
```
NEIGHBOUR  center : ( 0, 0);
           left   : (-1, 0);
           right  : ( 1, 0);
           upper  : ( 0, 1);
           lower  : ( 0,-1);
```

4.1.5 Konstantenbezeichner- und Variablen-Deklaration

VarConstDeclaration

An dieser Stelle des Programms können nun bei Bedarf Konstantenbezeichner sowie Variablen deklariert werden.

Syntax:

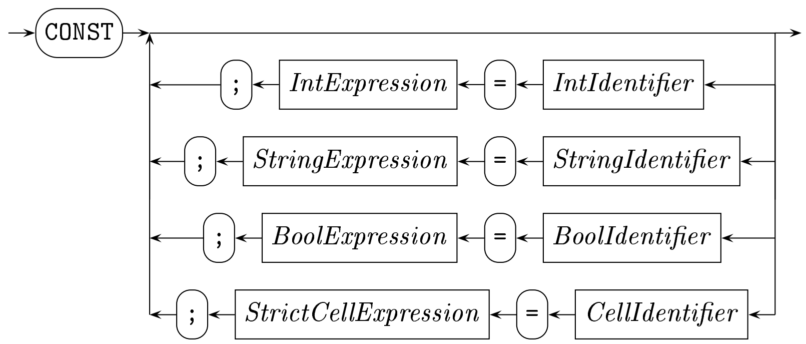


mit

VariableDeclaration:

und

ConstantDeclaration:



Wir haben beide Konzepte bereits im Zusammenhang mit RDL erläutert und verweisen deshalb hinsichtlich einer Erklärung der Syntax auf die einschlägigen Stellen in RDL, S. 71 ff., sowie auf die Abschnitte über Variablen und Konstantenbezeichner, S. 36, in Kapitel 3.

Es bleibt zu erwähnen, daß die Inhalte von Konstantenbezeichnern durch die gesamte Simulation hinweg erhalten bleiben, während Variablen nach jedem Wechsel der Bezugszelle neu initialisiert werden. Somit macht die Verwendung von Nachbarn in Konstantenbezeichner-Deklarationen auch in SDL keinen Sinn und ist daher untersagt.

Keine Nachbarn

4.1.6 Randwert-Festlegung *BorderDeclaration*

Das formale Zellularautomatenmodell sieht für alle Zellen in der die Retina umschließenden Grenzschicht einen einheitlichen Zustand vor, der sich von jenen, die Zellen innerhalb der Retina annehmen können, unterscheidet.
SCARLET trägt diesem Umstand indirekt dadurch Rechnung, daß es in SDL die Funktion `isborder()` zur Verfügung stellt, die darüber Auskunft gibt, ob eine bestimmte Zelle in der Grenzschicht liegt, oder nicht. Auf diese Art wird also eine zusätzliche Zustandsinformation geschaffen, die es ermöglicht, Grenzzellen ansonsten beinahe wie Retinazellen zu behandeln. Das heißt konkret: Grenzzellen besitzen (genau wie Retinazellen) einen in Register aufgeteilten Zustand, der durch diesen Programmteil *BorderDeclaration* vom Benutzer in beliebiger Weise festgelegt werden kann. Dieser Zustand ist dann der gemeinsame Zustand aller Grenzzellen, der während der gesamten Simulation unverändert erhalten bleibt. Diese Methode erweist sich beim Anlegen der Überführungstafel als vorteilhaft insofern, als auf eine gesonderte Randwertbehandlung ganz oder in Teilen verzichtet werden kann, falls dies im Sinne des Benutzers ist. (Die hierdurch gesparte Schreibarbeit sollte nicht unterschätzt werden!)

Standard-Wert

Standardmäßig enthalten alle Int-Register einer Grenzzelle in Scarlet den Wert `35` und alle String-Register das Zeichen `#`. Wollen Sie diese (willkürlich gewählten) Vorgaben ändern, so tun Sie dies durch

Syntax:

wobei *StrictCellExpression* für den gewünschten einheitlichen Zustand aller Grenzzellen steht. Die Auswertung von *StrictCellExpression* erfolgt nur ein einziges Mal (vor Simulationsbeginn) um die Konstanz des Wertes zu gewährleisten.

`VOID`

Bemerkung:
Trotz der etwas anderen Syntax haben wir es hier mit einer Zuweisung zu tun. `VOID` als Registerinhalt bewirkt demzufolge, daß für dieses Register die Standardvorgabe übernommen wird.

Bsp.45: Für ein Int- und ein String-Register sind z.B. folgende Grenzzustandsvereinbarungen möglich:

```
BORDER {0,""};
BORDER {42, VOID}; ⇝ {42,"#"}
```

4.2 Der Anweisungsteil

Wir kommen nun zu dem zentralen Zweck eines SDL-Programms: der Definition einer (oder mehrerer) lokaler Überführungsfunktion(en) σ durch Überführungstafeln. Durch sie wird der eigentliche Ablauf der Simulation geregelt. Innerhalb eines diskreten Simulationsschrittes werden dabei zeitlich parallel[2] die Zustände aller Zellen der Retina in Abhängigkeit gewisser Eintragungen in der Funktionstafel geändert. (Diese nehmen üblicherweise Bezug auf die Zustände der Nachbarzellen, die diese infolge des vorangegangenen Simulationsschrittes angenommen haben.)
Spätestens hier zeigt sich nun der entscheidende Vorteil Scarlet's gegenüber einer eigenen Implementation: Da in Scarlet bereits alle nötigen Routinen zur Bewältigung und Steuerung des Simulationsablaufes fest integriert sind

[2]Natürlich können die Veränderungen an der Retina durch Scarlet nur dann zeitgleich vorgenommen werden, wenn die Ausführung auf einem Parallelrechner erfolgt. Ansonsten wird dieser Vorgang durch einen äquivalenten seriellen simuliert.

(d.h., Scarlet die Überführungstafel selbständig auf alle Zellen anwendet, passende Datenstrukturen sowohl für die eigentliche Retina als auch zur Speicherung von Zwischenergebnissen anlegt, die Endergebnisse wieder in die Retina überträgt usw.), reduziert sich unsere Aufgabe auf die Festlegung einer (oder) mehrerer Überführungstafel(n), deren Angabe nun in fast vollständig abstrakter Form erfolgen kann.
Trotzdem werden wir einige Worte darüber verlieren müssen, auf welche Art Scarlet eine Funktionstafel interpretiert, zumindest soweit es einen Einfluß auf unsere Vorgehensweise bei der Konstruktion hat.

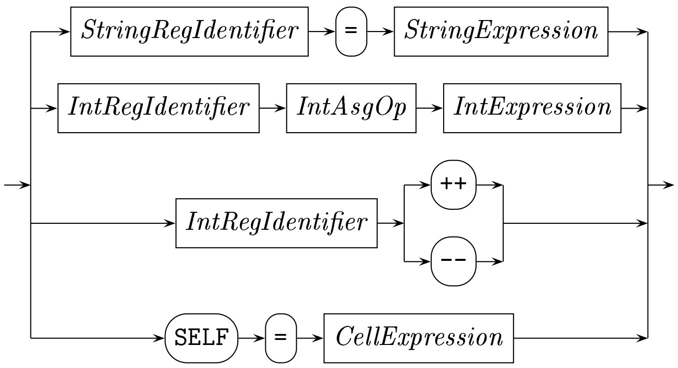
Zunächst ist es aber an der Zeit, eine zum Aufbau der Überführungstafel benötigte spezielle Syntax für Zuweisungen an Retina-Zellen, auch einfach Zellen genannt, vorzustellen:

4.2.1 Wertzuweisungen an Zellen in SDL-Programmen

SelfAssignment ;

Wie sich der Leser vielleicht erinnern wird, haben wir bereits in Kapitel 4 über RDL eine Syntax vorgestellt, die gezielte Wertzuweisungen an Zellen durch Angabe ihrer Raumkoordinate ermöglichte. Wozu das ganze also noch einmal? Dort (in RDL) ging es uns um die (einmalige) Gestaltung einer Startkonfiguration. Ist sie abgeschlossen, so dürfen Änderungen nur noch durch Anwendung der globalen Überführungsfunktion auf die Konfiguration, d.h. durch gleichzeitige Anwendung der lokalen Überführungsfunktion auf alle Zellen erfolgen. Das hohe Abstraktionsniveau in Scarlet gestattet es uns hier, eine solche Anweisung im Rahmen einer Funktionstafel nur einmal, stellvertretend für alle Zellen zu formulieren.

Syntax:



Bezugszelle Die virtuelle Zelle, an deren Stelle wir uns jede Zelle der Retina denken können, haben wir auch bei früheren Gelegenheiten schon „Bezugszelle“ genannt. Syntaktisch wird sie mit `SELF` bezeichnet. Und „Änderungen“ an der Konfiguration sind letztlich eine Folge von Zuweisungen von zellwertigen Objekten an die Zellen der Retina - bzw. in der neuen Notation: an die Bezugszelle `SELF`, so daß wir insgesamt folgende Syntax erhalten:[3]

`SELF` = *CellExpression* ;

Nachbarn Beachten Sie, daß mit *CellExpression* in SDL auch Nachbarn (*Neighbour-Identifier*) gemeint sein können. In diesem Fall ist *CellExpression* im Verlauf der gesamten Simulation i. a. kein konstanter Ausdruck, sondern veränderlich mit wechselnder Bezugszelle.

`VOID` `VOID` als Registerinhalt in *CellExpression* bewirkt hier, daß der entsprechende Registerinhalt von `SELF` nicht verändert wird.

Bsp.46: Für eine 2-dimensionale Retina, drei Nachbarn `left`, `right`, `center` (die Bezugszelle) und eine Zellstruktur bestehend aus drei numerischen Registern `num1`, `num2`, `num3` sind beispielsweise folgende Bildungen zulässig:

```
SELF = left;
SELF = {center.num1+1,VOID,VOID};
SELF = {VOID,right.num3,left.num2};
```

Einzelne Register Auch hier haben wir wieder die Möglichkeit, Zuweisungen nur an einzelne Register vorzunehmen. Dafür müßten wir, wie für den allgemeinen Fall im Abschnitt „Ansprechen einzelner Register“ beschrieben, als erstes durch Anhängen des entsprechenden Registernamens an die Zellbezeichnung die Projektion auf das betreffende Register bilden, also z.B. `SELF`.*StringRegIdentifier* für ein Stringregister. Da `SELF` über keinen Informationsgehalt verfügt, wird es in der Syntax generell weggelassen. Wir schreiben stattdessen einfach

StringRegIdentifier

für Stringregister und

IntRegIdentifier

für numerische Register.
Die restliche Syntax präsentiert sich vollkommen analog zu der für numerische und Stringvariablen, so daß wir insgesamt erhalten

[3] Auch hier ist das Semikolon kein eigentlicher Bestandteil von *SelfAssignment* (vgl. *VarAssignment*)

StringRegIdentifier `=` *StringExpression* ;

bzw.

IntRegIdentifier `=` *IntExpression* ;

Für numerische Register stehen ebenso die Kurzformen mit gleicher Bedeutung wie für Variablen zur Verfügung:

Kurzschreibweisen

IntRegIdentifier IntAsgOp IntExpression ;

IntRegIdentifier `++` ;	(Dekrement)
IntRegIdentifier `--` ;	(Inkrement)

Genaueres hierzu finden Sie in Kapitel 3, Abschnitt 2.9, S. 62.

Bsp.47: In Beispiel 46 statt `SELF = {center.num1+1,VOID,VOID}`:

```
num1++;
```

4.2.2 Die Einträge der Überführungstafel

Entry

Bevor wir uns eine Überführungstafel als Ganzes anschauen, wollen wir zunächst ihre einzelnen Bestandteile etwas näher unter die Lupe nehmen:
Eine Funktionstafel konsistiert grundsätzlich aus einer beliebig langen Folge[4] von Einträgen der folgenden Form:

Syntax:

→ *ConditionPart* → : → *Consequence* → : →

Wie durch die Namen der beiden Hauptbestandteile bereits angedeutet wird (und wie vielleicht schon zu vermuten war), handelt es sich bei den einzelnen Einträgen also um eine Art bedingter Anweisung, deren Aufbau allerdings nicht ganz so streng ist, wie etwa der eines IF-THEN-ELSE-Konstrukts. Dennoch gilt: Der Bedingungsteil enthält stets explizit (oder implizit) einen booleschen Ausdruck, bei dessen Auswertung zu TRUE *Consequence* bearbeitet wird, und andernfalls übergangen wird. In diesem Fall wird der sich an *Consequence* anschließende Programmteil bearbeitet.

---

[4] Die Formulierung mag vielleicht den Eindruck erwecken, daß die Reihenfolge innerhalb der Tafel irrelevant sei. In der Tat ist dies nicht der Fall. Wir gehen hierauf in Abschnitt 4.2.3, S. 96 näher ein.

Bsp.48: Das nachstehende einfache Beispiel soll den Aufbau einer Überführungstafel und ihrer Einträge *Entry* demonstrieren.
Dazu betrachten wir eine 2-dimensionale Retina, Zellen mit einem numerischen Register `state`, sowie einen Nachbarschaftsindex bestehend aus der der Bezugszelle selbst, sowie den acht sie umgebenden Zellen. Das Modell soll die **Erregung von Nervenzellen**, ausgelöst durch benachbarte Nervenzellen, nachstellen. Dabei kann eine Zelle drei mögliche Zustände annehmen:

- schußbereit $\leftrightarrow$ `0`
- aktiv $\leftrightarrow$ `1`
- regenerierend $\leftrightarrow$ `2`

Zusätzlich gelten folgende Regeln:

- eine schußbereite Zelle wird aktiv, wenn mindestens drei der sie umgebenden Zellen aktiv sind.
- eine aktive Zelle fällt danach in den regenerativen Zustand
- eine regenerierende Zelle ist im nächsten Schritt wieder schußbereit.

Durch die geeignete Wahl der Zustände (`0, 1, 2`) können alle Zustandswechsel durch eine Addition von 1 mod3 realisiert werden. Die Überführungstafel besteht aus zwei Einträgen: Der erste überpüft, ob die Zelle schußbereit ist; wenn nicht (`center.state > 0`), wird in dem zugehörigen Konsequenzteil die Addition ausgeführt. (Die nachfolgenden Einträge werden ignoriert.) War die Zelle doch schußbereit, so wird *Consequence* übergangen und der zweite Eintrag bearbeitet: `center.state == 0` müßte hier nicht mehr geprüft werden (ist nun bestimmt erfüllt). Ist außerdem die Summe der `1`-Zustände der umgebenden Zellen mindestens drei (werden bitweise mit 1 verglichen), so geht die Zelle durch *Consequence* in den aktiven Zustand über, andernfalls behält sie ihren alten Zustand bei:

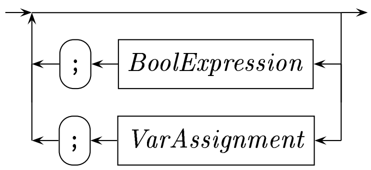
```
SIGMA firing_neurons;
DIMENSION 2;
REGISTER INT state;
NEIGHBOUR center     : ( 0, 0);
          left       : (-1, 0);     left_lower : (-1,-1);
          right      : ( 1, 0);     right_lower: ( 1,-1);
          lower      : ( 0,-1);     left_upper : (-1, 1);
          upper      : ( 0, 1);     right_upper: ( 1, 1);
```

```
VAR INT sum;
BORDER {0};

TABLE
/* 1. Eintrag */
  center.state > 0;     /* ConditionPart */
        : state++; state %= 3; :     /* Consequence */

/* 2. Eintrag */
  center.state==0;     /* ConditionPart */
  sum=(left.state&1) + (right.state&1)
      + (lower.state&1) + (upper.state&1)
      + (left_lower.state&1) + (right_lower.state&1)
      + (left_upper.state&1) + (right_lower.state&1);
  sum>=3;
   : SELF = {1}; :     /* Consequence */
```

Der Bedingungsteil *ConditionPart*

Die oben angesprochene Verallgemeinerung von Überführungstafel-Einträgen gegenüber IF-THEN-ELSE-Bildungen ist im wesentlichen im Bedingungsteil angesiedelt:

Syntax:



Dieser erlaubt nämlich die Angabe beliebig vieler boolescher Ausdrücke

BoolExpression ;

sowie zusätzlich auch beliebig viele Variablen-Zuweisungen,

VarAssignment ;

in beliebiger Reihenfolge stehend und jeweils durch Semikolon abgeschlossen. Dabei gelten folgende Regeln:

- Der Wahrheitswert des Bedingungsteils entspricht dem Wahrheitswert der Konjunktion[5] aller auftretenden booleschen Ausdrücke. Jener entscheidet über die Ausführung (bei TRUE) oder Nichtausführung (bei FALSE) des Konsequenzteils.

  Bsp.49: Beachte: `bool1 || bool2; bool3;` ist dasselbe wie `(bool1 || bool2) && bool3;`

- Ist kein boolescher Ausdruck vorhanden, so ist der Wahrheitswert des Bedingungsteils TRUE.

- Alle durch Semikolon getrennt stehenden Einheiten werden in der Reihenfolge ihres Auftretens ausgewertet. Die Wirkung von Variablen-Zuweisungen tritt sofort ein.

nichtstrikte Auswertung

Die Bearbeitung erfolgt <u>nichtstrikt</u>, d.h., sie bricht ab, sobald ein boolescher Ausdruck zu FALSE evaluiert wurde. (Zu diesem Zeitpunkt steht bereits fest, daß *Consequence* nicht bearbeitet wird.)

Bsp.50: Würde in Beispiel 48 die Summe `sum` statt im zweiten schon (wie hier) im ersten Eintrag nach `center.state > 0;` ermittelt werden, so würde dies zu Fehlern führen: Im Fall, daß `center.state == 0;` gilt, wäre `center.state > 0;` stets falsch, so daß die Bearbeitung des ersten Eintrags sofort abgebrochen und `sum` nicht mehr berechnet würde:

```
center.state > 0; sum = left.state&1 + ... :
          state++; state%=3; :
center.state == 0; sum >= 3; :
          SELF = {1}; :
```

! Beachte: Im Rahmen des Bedingungsteils ist es lediglich gestattet, die Inhalte von Variablen zu verändern; an den Zuständen der Zellen kann hier nicht gerüttelt werden.

Es sei an dieser Stelle bereits erwähnt, daß die durch ein SDL-Programm erstellte Überführungstafel (und somit auch ihre Einträge) während des Simulationsvorgangs separat auf jede Zelle der Retina angewandt wird, zu dem

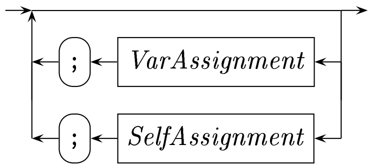
[5] Konjunktion = Verknüpfung mit logisch UND

Zweck, ihr einen neuen Zustand zuzuweisen. Werden dabei die booleschen Ausdrücke in *ConditionPart* im Sinne der Überführungsfunktion σ genutzt, so enthalten sie in erster Linie Aussagen über die Zustände ihrer Nachbarzellen. In Abhängigkeit davon werden dann Folgezustandszuweisungen vorgenommen. Wem diese Bemerkung nun nicht unmittelbar einleuchtet, der möge sich noch bis zum Abschnitt über die Überführungstafel, S. 96, gedulden; dort werden diese Vorgänge noch ein wenig genauer beleuchtet.

Der Konsequenzteil *Consequence*

In den Konsequenzteilen der Einträge verbergen sich u.a. die eigentlichen, für den Zellularautomaten wichtigen Aktionen, die eine Konfigurationsänderung herbeiführen können. Sie werden jedoch nur ausgeführt, falls *ConditionPart* zuvor bearbeitet wurde (vgl. voriger Abschnitt) und all seine booleschen Ausdrücke zu TRUE ausgewertet wurden.

Syntax:



Neben den bereits bekannten Variablen-Zuweisungen

VarAssignment ;

finden wir in *Consequence* auch die weiter oben besprochenen Zuweisungen an Retina-Zellen

SelfAssignment ;

(Das Semikolon ist als syntaktisches Element einer solchen Zuweisungseinheit innerhalb von *Consequence* zu verstehen.) Beide Zuweisungsarten können beliebig oft und in beliebiger Reihenfolge in *Consequence* auftauchen.

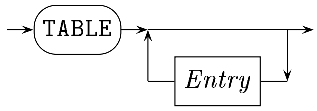
Abarbeitungsfolge

Auch hier gilt:
Die Evaluation der einzelnen Zuweisungseinheiten in *Consequence* geschieht in der Reihenfolge ihres Auftretens.
Während die Änderung von Variableninhalten sofort erfolgt, tritt die Wirkung von *SelfAssignment* erst nach Abschluß eines vollständigen Simulationsschrittes ein. Dadurch wird der Forderung nach einer parallelen Anwendung der Überführungsfunktion auf alle Zellen Rechnung getragen.

4.2.3 Die Überführungstafel

Table

Syntax:



Der Beginn einer Überführungstafel wird SCARLET durch das Schlüsselwort

`TABLE`

bekanntgegeben. (Das Ende der Tafel wird entweder gekennzeichnet durch den Beginn einer weiteren Überführungstafel, oder aber es ist gleichzeitig auch das Ende des SDL-Programms.) Auf `TABLE` folgen die einzelnen Einträge *Entry* der Form

$$ConditionPart \; : \; Consequence \; :$$

Eine Überführungstafel kann beliebig viele Einträge enthalten.

Bemerkung:
Layout Da sowohl *ConditionPart* als auch *Consequence* mit Doppelpunkt abgeschlossen werden, ist es bei umfangreicheren Tafeln sicherlich von Vorteil, eine Notation zu verwenden, durch die beide Teile deutlich voneinander abgehoben werden, z.B. durch Einrücken der Doppelpunkte.

$$\begin{array}{l} ConditionPart_1 \\ \qquad\quad : \\ Consequence_1 \\ : \\ ConditionPart_2 \\ \qquad\quad : \\ Consequence_2 \\ : \\ \qquad\quad \vdots \end{array}$$

Interpretation Soweit die Syntax. Um zu verstehen, wie eine solche Tafel von SCARLET gelesen wird, erinnern wir noch einmal an die formale σ-Funktion des Zellularautomaten-Modells, kurz σ. Durch sie wird eine Funktion beschrieben, die in jedem Simulationsschritt zeitlich parallel jeder Zelle in Abhängigkeit der Zustände ihrer jeweiligen Nachbarn einen Folgezustand zuweist. Bei σ handelt es sich also um eine Funktion von S^d nach S, falls S die Zustandsmenge ist und der Nachbarschaftsindex vom Grade d ist.

Im Vergleich dazu betrachten wir nun den Aufbau einer Überführungstafel im ganzen. Da die Eigenschaften der einzelnen *ConditionPart*- und *Consequence*-Teile einen entscheidenden Einfluß auf ihre Funktionsweise haben, werden sie bei dieser Gelegenheit noch einmal aufgeschlüsselt. Seien $Entry_1, \ldots, Entry_m$, $m \in I\!N$ die Einträge einer Überführungstafel, $c_1^{(i)}, \ldots, c_{j_i}^{(i)}$, $j_i \in I\!N_0$, $i \in \{1, \ldots, m\}$, alle Variablenzuweisungen aus $ConditionPart_i$ in ihrer richtigen Reihenfolge bis auf die, die infolge der nichtstrikten Behandlungsweise übergangen werden (S. 94). $b_i, i \in \{1, \ldots, m\}$, sei der boolesche Wert von $ConditionPart_i$ (siehe S. 93). Dann impliziert ein SDL-Programm mit dieser Tafel etwa folgenden, vereinfacht dargestellten Algorithmus:

$$
\begin{array}{l}
c_1^{(1)};\ c_2^{(1)};\ \ldots\ ;\ c_{j_1}^{(1)};\\
\text{IF } b_1 == \text{TRUE}\\
\quad \text{THEN } Consequence_1\\
\quad \text{ELSE}\\
\qquad \{c_1^{(2)};\ c_2^{(2)};\ \ldots\ ;\ c_{j_2}^{(2)};\\
\qquad\ \text{IF } b_2 == \text{TRUE}\\
\qquad\quad \text{THEN } Consequence_2\\
\qquad\quad \text{ELSE}\\
\qquad\qquad \{c_1^{(3)};\ c_2^{(3)};\ \ldots;\ c_{j_3}^{(3)};\\
\qquad\qquad\ \ \vdots\\
\qquad\qquad \}\\
\qquad \}
\end{array}
$$

Er beschreibt die Vorgänge bei Anwendung der Überführungstafel auf eine beliebige Zelle z. Verfügt die Retina über k Zellen, so erfordert die Ausführung eines Simulationsschrittes dementsprechend den k-maligen Aufruf des SDL-Programms. Dies geschieht jedoch selbständig und ohne Zutun des Benutzers.

Beachten Sie weiterhin, daß

- alle **Variablen** vor jedem Aufruf der Funktionstafel initialisiert werden,

- **Symbolische Konstanen** nur vor vollen Simulationsschritt und

- die **Zellen** der Retina schließlich nur nach abgeschlossenen Simulationsschritten aktualisiert werden.

Die Bearbeitung für Zelle z vollzieht sich beim ersten Eintrag der Tafel beginnend nach folgendem Prinzip: Führe im Bedingungsteil des Eintrages i alle Anweisungen in der Reihenfolge ihres Auftretens aus und überprüfe zwischendurch alle booleschen Ausdrücke. Wurden sie alle zu TRUE ausgewertet, so bearbeite den zugehörigen Konsequenzteil und führe eventuell eine damit verbundene Zuweisung an z aus; den Rest der Tafel ignoriere. Der Vorgang für Zelle z ist somit abgeschlossen.
Andernfalls beende die Bearbeitung von $ConditionPart_i$, sobald ein boolescher Ausdruck zu FALSE ausgewertet wurde. Übergehe $Consequence_i$ und setze die Bearbeitung mit dem (i+1)-ten Bedingungsteil fort.

$\boxed{!}$ Während einer Ausführung des SDL-Programms wird also maximal ein Konsequenzteil pro Überführungstafel[6] bearbeitet, und zwar der erste, für den die Bedingungen in *ConditionPart* erfüllt sind. Dies ist deshalb bemerkenswert, weil lediglich durch diesen einen Konsequenzteil der Zustand der gerade bearbeiteten Zelle beeinflußt werden könnte, oder anders formuliert: ein Funktionswert im Sinne von σ vergeben werden kann.

Bsp.51: Wir bemühen noch einmal Beispiel 48, um die Einträge seiner Überführungstafel in die algorithmische Form zu übersetzen:

```
IF center.state > 0; == TRUE
   THEN
      state++;
      state %= 3;
   ELSE
      IF center.state==0; == TRUE
         THEN
             sum = left.state&1 + ... ;
             IF (center.state==0; && sum <=3) == TRUE
                THEN SELF = {1};
```

Beachte: `sum` wird nur ermittelt, falls `center.state == 0` ist, daher die zusätzliche Abfrage vor `sum = ...` .

[6] In Abschnitt 4.2.4, S. 101 wird klar, daß ein SDL-Programm mehrere Überführungstafeln enthalten kann.

Reihenfolge der Einträge

Welche Rolle hierbei die Anordnung der Einträge innerhalb der Tafel spielt, soll folgendes, einfaches Beispiel verdeutlichen:

Bsp.52:

```
SIGMA signal;

DIMENSION 1;
REGISTER INT state;
NEIGHBOUR left : (-1);

TABLE
left.state == 1; : SELF = {1}; :
TRUE; : SELF = {0}; :
```

erzeugt, angewandt auf die folgende eindimensionale Retina der Länge l

1	0	0	0	…	0	0

ein Signal, das mit „Geschwindigkeit 1“ von links nach rechts wandert und bei Auftreffen auf den rechten Rand verschwindet:

k=0:	**1**	0	0	0	…	0	0
k=1:	0	**1**	0	0	…	0	0
k=2:	0	0	**1**	0	…	0	0
					…		
k=l-1:	0	0	0	0	…	0	**1**
k=l:	0	0	0	0	…	0	0

Werden die beiden Einträge vertauscht, so ändert sich das Verhalten radikal: Bereits nach einem Schritt ist das Signal verschwunden.
Der Grund hierfür ist offensichtlich: Die Menge aller Kombinationen von Nachbar-Konfigurationen, für die Bedingung eins erfüllt ist, ist in der, für die Bedingung zwei erfüllt ist, enthalten.
Wäre Bedingung drei so formuliert worden, daß sich keine Überschneidungen ergeben hätten (`left.state != 1;`), so hätte ein Vertauschung der beiden Einträge keine Auswirkungen auf das Verhalten des Zellularautomaten gehabt.

Um also auch bei komplexeren Überführungstafeln den Überblick zu behalten, empfiehlt es sich, bei ihrem Aufbau die hier noch einmal allgemein formulierte Strategie zu beachten:

Eintrags-Reihenfolge

Seien dazu $E_1, \ldots, E_m, m \geq 2$, die Einträge einer Überführungstafel, $z \in \mathbb{Z}^n$ die Koordinate der Bezugszelle, $N = (n_1, \ldots, n_d)$ n-dimensionaler Nachbarschaftsindex vom Grade d. Dann bezeichnen wir in diesem Abschnitt mit M_i die Menge aller Tupel $(s_1, \ldots, s_d) \in S^d$, für die die Bedingung des Eintrages E_i erfüllt ist, wenn $s_1 = c\,(z+n_1), \ \ldots, \ s_d = c\,(z+n_d)$ für eine Konfiguration $c : \mathbb{Z}^n \longrightarrow S$ ist. Damit erhalten wir die folgenden Bedingungen:

1. Falls $M_i \cap M_j = \emptyset, i \neq j$, so können die Einträge E_i und E_j in beliebiger Reihenfolge in der Überführungstafel angeordnet werden.
2. Falls $M_i \subseteq M_j, i \neq j$, so ist der Eintrag E_i mit der spezielleren Bedingung dem Eintrag E_j voranzustellen.
3. Alle anderen Fälle führe man durch Mengenzerlegung in geeigneter Weise auf 1. und 2. zurück.

Implizite Zuweisungen

Besondere Beachtung erfordert das Verhalten der Überführungstafel in Fällen, in denen die Bezugszelle ihren Folgezustand nicht durch eine *SelfAssignment*-Anweisung erhält. Eine solche Situation tritt auf, wenn

1. ihre Nachbar-Konfiguration keine der Eintragsbedingungen erfüllt,
2. zwar ein Konsequenzteil erreicht wurde, dort aber
   (a) keine *SelfAssignment*-Anweisung zu finden ist, oder
   (b) nur eine partielle Zuweisung vorgenommen wird (d.h. eine *SelfAssignment*-Anweisung, die nur bestimmte Register umfaßt (S. 90).

In 1. und 2.(a) behält die Zelle ihren alten Zustand bei. Im Fall 2.(b) bleiben alle Register unverändert, die durch die partielle Zuweisung nicht erfaßt wurden.
Allgemein besteht in jedem der Fälle eine Abhängigkeit des Folgezustandes der Bezugszelle von ihrem Zustand im vorangegangenen Simulationsschritt.

Ist die Bezugszelle selbst ein Nachbar (enthält der Nachbarschaftsindex also die Eintragung $(0, \ldots, 0) \in \mathbb{Z}^n$), so sind Abhängigkeiten dieser Art erlaubt, und die Überführungstafel entspricht stets den Anforderungen an eine σ-Überführungsfunktion im formalen Modell.

$\boxed{!}$ Kritischer wird es, falls die Bezugszelle <u>nicht</u> zu den Nachbarn gehören soll. In diesem Fall muß der Benutzer selbst dafür Sorge tragen, daß die Überführungstafel keine der oben beschriebenen Situationen erlaubt, wenn sie eine σ-

Überführungsfunktion repräsentieren soll.

Unvollständige Tafeln

Während diese Beschränkung für 2. noch überschaubar ist (da zur Überprüfung die Betrachtung einzelner Einträge für sich genommen ausreichend ist), ist in manchen Fällen das einzige hinreichende Kriterium zur Vermeidung von 1., das auch hinreichend einfach überprüfbar ist, die Vollständigkeit der Überführungstafel ($M_1 \cup M_2 \cup, \ldots, \cup M_m = S^d$; für die Definition der M_i siehe Seite 100).

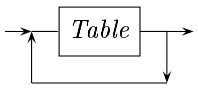
Bei unvollständigen Tafeln muß gewährleistet sein, daß alle nicht berücksichtigten Nachbar-Konfigurationen tatsächlich nicht angenommen werden.

!

Beachte: In diese Überlegungen muß auch das Aussehen der Startkonfiguration mit einbezogen werden!

4.2.4 Aufbau des Anweisungsteils *TablePart*

Syntax:



Mehrere Tafeln

Im Anweisungsteil stehen lediglich eine oder mehrere Überführungstafeln *Table*. Die Funktionsweise einer einzelnen Tafel haben wir bereits im vorangegangenen Abschnitt kennengelernt. Folgen mehrere Überführungstafeln aufeinander, so wird zunächst die oberste Tafel in der bekannten Weise auf die behandelte Zelle angewandt, dann mit der nächsten Tafel ebenso verfahren usw. . Wir haben es dann mit einer Hintereinanderausführung mehrerer Funktionen zu tun.

4.3 Kompatibilität von RDL- und SDL-Programmen

Durch die vorangegangenen Kapitel ist der Leser nun (hoffentlich!) in die Lage versetzt worden, gültige RDL- und SDL-Programme zu verfassen. Diese liegen allerdings in getrennten Dateien vor und werden erst später zu einem vollständigen Zellularautomaten zusammengesetzt (und selbst dann nur für die Dauer der Simulation).
Vorteil dieser Methode ist, daß auf relativ einfache und platzsparende Weise

unterschiedliche Startkonfigurationen mit verschiedenen Überführungstafeln kombiniert werden können.
Dabei versteht es sich von selbst, daß nicht jede denkbare Retina mit jeder Überführungstafel harmoniert. Erst folgende Bedingungen an RDL- und SDL-Programme gewährleisten die Kompatibilität:

1. Die Raumdimensionen müssen identisch sein.
   (In RDL wird sie im Rahmen des `RANGE`-Befehls definiert, in SDL direkt durch `DIMENSION`...)

2. Die Zustandsmenge muß jeweils dieselbe sein. Das heißt konkret:
   In beiden Programmen müssen dieselben Registertypen in derselben Reihenfolge deklariert werden.
   (Die einzelnen Register müssen allerdings nicht gleich benannt sein.)

Variablen, Konstantenbezeichner

Sollten Sie in Ihren Programmen Variablen oder Konstantenbezeichner verwenden, so brauchen Sie nicht für eine eindeutige Benennung innerhalb beider Programme zu sorgen, da deren Gültigkeitsbereich auf das Programm beschränkt ist, in dem sie deklariert wurden.

Bsp.53: Ausschnitte zweier zueinander kompatibler RDL- und SDL-Programme:

```
RETINA ;                               SIGMA;

RANGE [0,-12,4,2]..[2,4,7,-1];         DIMENSION 4;

REGISTER                               REGISTER
        INT     r1;                            INT     reg1;
        STRING  r2;                            STRING  reg2;
        INT     r3;                            INT     reg3;
        INT     r4;                            INT     reg4;
     ...                                    ...
```

Bemerkung:
Außer den oben genannten Punkten existieren keine weiteren formalen Beschränkungen für die Kombination von RDL- und SDL-Programmen.

Semantische Einschränkungen

Allerdings ist es für deren sinnvolle Zusammenarbeit für den Anwender unerläßlich, auch die semantischen Eigenschaften beider Komponenten aufeinander abzustimmen. (Was auf den ersten Blick vielleicht wie eine trivialerweise erfüllte Bedingung anmutet, macht vor allem dann Schwierigkeiten, wenn nach längerer Ruhezeit an bereits bestehende Arbeiten angeknüpft werden soll...!)

Setzen wir voraus, daß beiden Programmen derselbe formale (also von der konkreten Umsetzung unabhängige) Zellularautomat zugrunde liegt, so spielt im wesentlichen nur noch die Modellierung der Zustandsmenge eine Rolle, d.h.

- die Aufteilung in Register mit ihrer jeweiligen Bedeutung; (bei mehreren Registern gleichen Typs ist auch deren Reihenfolge relevant!)
- die Bedeutung der verwendeten Registerinhalte, und damit eng zusammenhängend
- der durch die Überführungstafel abgedeckte Wertebereich der einzelnen Register (in dem Fall, daß die Bezugszelle kein Nachbar ist, vgl. S. 100).

Kapitel 5

Benutzeroberfläche

Last but not least soll es nun noch kurz um die Benutzerschnittstelle SCARLET'S gehen. Es handelt sich dabei um eine unter Motif erstellte Windows-Oberfläche, von der aus die Vorgänge der Simulation gesteuert und beobachtet werden können.

Im Gegensatz zu verschiedenen Software-Paketen beschränkt sich SCARLET nicht auf die Verwendung einiger weniger Zustände, die dann oft als farbige Punkte auf dem Bildschirm dargestellt werden, sondern erlaubt komplexe Datenstrukturen und Zustände. Dem muß auch die Darstellungsweise gerecht werden, und dies geschieht durch die Möglichkeit zur vollständigen Anzeige aller Zustandsinformationen, wie sie im Programm verwendet wurden.

Da wir nicht automatisch davon ausgehen können (und wollen), daß alle SCARLET-Nutzer über dieselbe Erfahrung im Umgang mit Fenstersystemen verfügen, bietet dieses Kapitel neben einer schematischen Übersicht über die wichtigsten Aktionen eine ausführlichere Beschreibung der einzelnen Programm-Elemente. Wir möchten jedoch ausdrücklich dazu ermutigen, die Benutzeroberfläche SCARLET'S einfach durch Ausprobieren kennenzulernen und diese Beschreibung lediglich im Falle auftretender Ungereimtheiten zu konsultieren.

5.1 Kurzanleitung

1. **Erstellen** einer Überführungstafel in Form eines **SDL-Programmes** sowie einer passenden Retina inklusive Startkonfiguration als **RDL-Programm** mit Hilfe eines beliebigen Texteditors und **Speichern** dieser in getrennten Dateien (vorgeschlagene Endungen: `sdl` bzw. `rdl`).

2. **Starten** von SCARLET durch Aufruf der Programmdatei `scarlet`:

   `./scarlet` <Return>

3. **Laden** einer bereits kompilierten Überführungstafel oder — Sigma

   Laden eines **SDL-Programms** und anschließendes **Kompilieren**. — Compile

4. **Laden** einer kompatiblen Retina und Startkonfiguration (**RDL-Programm**). — Retina

5. *Auswahl der gewünschten* ***Anzeige- und Simulations-Optionen***

   - *Retina-Ausschnitt*
   - *Zustands- und Positionsangaben*
   - *Koordinatensystem*
   - *(Bei 1-dimensionaler Retina: Bildschirm-Umbruch)*
   - *Schrittweite*

6. **Ausführen** der durch Schrittweite angegebenen **Simulationsschritte**. — Run

7. *Bei Bedarf:* ***Wechseln*** *von*

   - *Retina oder Überführungstafel (falls nicht kompatibel, wählen Sie* `Drop old` *in der erscheinenden Meldung.)*
   - *Anzeige- oder Simulations-Optionen*

   Neuladen *der Startkonfiguration. (**Achtung:** Die bestehende Konfiguration auf dem Bildschirm wird überschrieben!)* — Reload

8. **Beenden** von SCARLET. — Done

5.2 Detaillierte Beschreibung

5.2.1 Scarlet starten / beenden

Um Scarlet nutzen zu können, benötigen Sie ein Computersystem mit Betriebssystem SUN Solaris ab Version 2.0 sowie der integrierten Windows-Oberfläche X11. Nach erfolgreicher Installation eröffnen sich Ihnen unter X11 gleich mehrere Möglichkeiten, Scarlet zu starten:
Verwenden Sie ein Terminal- oder Konsole-Fenster, so wechseln Sie in das Verzeichnis, in dem Scarlet gespeichert ist und geben Sie dort anschließend

`./scarlet` <Return>

ein. (Falls dieses Verzeichnis bereits in Ihrem Pfad enthalten ist, kann der Verzeichnis-Wechsel natürlich entfallen.)
Selbstverständlich können Sie aber auch den Datei-Manager verwenden und nach dem Wechsel in das Scarlet-Verzeichnis einfach die entsprechende ausführbare Datei `scarlet` durch doppelten Mausklick aktivieren.

Bemerkung:
Denken Sie daran, daß die Benutzeroberfläche lediglich dem Ausführen von Simulationen dient. Die dazu nötigen RDL- und SDL-Programme müssen zuvor bereits mit einem beliebigen Texteditor (emacs, vi, ...) erstellt und gespeichert worden sein.

Nach dem Start sollte sich Ihnen die Scarlet-Oberfläche in einem Fenster präsentieren.

Ein großer Bereich im Fensters, das **Simulationsfeld**, ist zu diesem Zeitpunkt leer; dort wird später die Konfiguration der Retina dargestellt werden.
Die **Meldungszeile** am unteren Rand des Programmfensters informiert über die Reaktionen Scarlet's auf durchgeführte Aktionen.
Die **Statusanzeige** in der rechten oberen Ecke zeigt an, daß momentan weder eine Retina, noch eine Überführungsfunktion aktiv ist, und somit folglich auch keine Nachbarschafts-Struktur; ferner, daß bisher kein zu kompilierendes SDL-Programm benannt wurde.

Damit eine Simulation zustande kommen kann, muß Scarlet zunächst mit einer Überführungstafel und einer Startkonfiguration versorgt werden. Die Reihenfolge, in der die zugehörigen Dateien geladen werden, spielt dabei keine Rolle.

Um SCARLET zu beenden, klicken sie auf `Done` (oder verwenden Sie das Fenster-Schließen-Symbol $\boxed{-}$).
Beachten Sie, daß Simulationsergebnisse damit automatisch verloren gehen. Done

5.2.2 Umgang mit dem X11-Fenstersystem

Da es zuweilen den Anschein hat, daß sich die hartnäckigsten Fehler und Probleme besonders gerne in allzu selbstverständlichen oder trivialen Bereichen aufhalten, seien an dieser Stelle wenigsten ein paar Worte über die allerwichtigsten Handgriffe im Umgang mit X11-Fenstern im allgemeinen verloren:

- **Aktivieren**: Klicken Sie mit der Maus in das Fenster.
- **Schließen**: Doppelklick auf $\boxed{-}$.
- **Verschieben**: Klicken Sie in die Kopfzeile und ziehen Sie das Fenster bei gedrückter Maustaste an die gewünschte Stelle.
- **Vergrößern (Verkleinern)**: Klicken Sie einen Rand oder eine Ecke des Fensters an (Maussymbol muß sich vorher verändert haben) und ziehen Sie sie bei gedrückter Maustaste an die neue Position.
- Als **Symbol** ablegen: Klick auf $\boxed{\cdot}$.
- **Symbol reaktivieren**: Doppelklick auf die Ikone.
- **Vollbild**: Klick auf $\boxed{\square}$.
- **Vollbild deaktivieren**: Weiterer Klick auf $\boxed{\square}$.

Übrigens können die hier vorgestellten Funktionen (ausgenommen „Aktivieren") auch durch einfachen Klick auf $\boxed{-}$ aus einem Pull-Down-Menü ausgewählt oder per Tastenkombination abgerufen werden.

5.2.3 Laden einer Retina

Um eine Retina - und damit verbunden - eine Startkonfiguration zu laden, klicken sie auf den Button mit der Aufschrift `Retina`. Daraufhin erscheint ein weiteres Fenster , das Ihnen die Möglichkeit gibt, eine beliebige Datei aus irgendeinem Verzeichnis auszuwählen und dem Retina-Interpreter SCARLET'S zuzuführen. SCARLET akzeptiert dabei jedes gültige RDL-Programm. Andere Dateien werden mit einer Fehlermeldung des Compilers (etwa, `parse error...` sowie näheren Angaben zum aufgetretenen Fehler) abgewiesen. Retina

Ist die Kompilierung erfolgreich verlaufen, so ändert sich nun die Statusanzeige: Es erscheint hinter `Retina:` der Name der Retina, der in der Kopfzeile des zugehörigen RDL-Programms vergeben wurde (`RETINA` <*Name*>`;`), zusammen mit ihrer Retina-Ausdehnung in den einzelnen Koordinatenrichtungen. Viel wichtiger ist allerdings, daß die geladene Retina nun mit ihrer derzeitigen (Start-)Konfiguration im Simulationsfeld dargestellt wird. Mehr dazu in 5.2.6.

Reload

Soll nach dem Durchführen einiger Simulationsschritte die Startkonfiguration wiederhergestellt werden, so müssen Sie die RDL-Datei nicht erneut laden; es genügt, den `Reload`-Button zu betätigen.

Das Retina-Datei-Öffnen-Fenster

Der **Filter** im oberen Bereich des Fensters bestimmt, welche Dateien im Datei-Auswahlfeld angezeigt werden sollen. Nach dem Programmstart handelt es sich hier standardmäßig um die Dateien mit Endung `rdl` im SCARLET-Verzeichnis (hier: `scarlet.085`):

```
.../scarlet.085/*.rdl
```

SCARLET suggeriert also - schon der Übersichtlichkeit halber - die Verwendung von Dateien mit dieser Endung für RDL-Programme; tatsächlich aber kann jede Datei diese Aufgabe übernehmen. Klicken Sie auf das Filter-Feld, um die Filter-Einstellungen abzuändern (z.B. `.../scarlet.085/*`, um alle Dateien im Verzeichnis anzuzeigen).

Wie alle Änderungen in Fenstern dieser Art bleiben auch diese bis zum Beenden des Programms erhalten.

Wählen Sie weiter in der **Verzeichnis-Auswahlliste** unter Verwendung der Pfeiltasten oder der Bildlaufleisten das Verzeichnis, in dem sich die gewünschte Datei befindet, und bestätigen Sie Ihre Wahl durch Return oder doppelten Mausklick auf das Verzeichnis.
(Insbesondere steht `/.` für das aktuelle, `/..` für das jeweils übergeordnete Verzeichnis; letzteres erlaubt somit einen beliebigen Wechsel in der Verzeichnisstruktur.)
Verzeichniswechsel werden automatisch in die Filterangaben sowie das Namensfeld übernommen.

Die **Datei-Auswahlliste** zeigt alle Dateien an, die der obigen Filterangabe entsprechen und sich im gewählten Verzeichnis befinden. Öffnen Sie die zu ladende Datei nach dem Wechsel in das richtige Verzeichnis durch Auswahl und Doppelklick in dieser Liste, oder geben Sie den vollständigen Namen und Pfad der Datei explizit im **Namensfeld** an und bestätigen Sie anschließend mit Return oder klicken Sie auf `Load retina` .

Falls Sie nach Programmstart noch Änderungen an der Datei-Auswahlliste vorgenommen haben, (d.h. eventuell Dateien gelöscht oder hinzugefügt haben), könnte es nötig sein, die Dateiliste zu aktualisieren. Sie tun dies, indem Sie den Button `Filter` betätigen.
Mit `cancel` schließen Sie das Fenster ohne eine Datei zu aktivieren.

5.2.4 Laden einer Überführungstafel

Im Gegensatz zu RDL-Programmen, die lediglich an einen Interpreter geleitet werden, erweist es sich (u.a. aus Laufzeitgründen) als sinnvoll, SCARLET SDL-Programme in kompilierter Form zur Verfügung zu stellen. Der (separate) Vorgang der **Kompilierung** wird dabei normalerweise ebenfalls von SCARLET'S Programm-Oberfläche aus vorgenommen werden (siehe unten). SCARLET speichert das Ergebnis dann automatisch in einer neuen Datei. Gleichzeitig zeigt SCARLET den Namen der gewählten Programmcode-Datei in der Statusanzeige als `Compile:` <Dateiname> an. Beachte: Durch die Kompilierung wird die Funktionstafel nicht aktiviert!

Soll eine bereits kompilierte Funktionstafel nun tatsächlich zum Zweck einer Simulation aktiviert werden, so muß sie in einem zweiten Schritt ausgewählt und geladen werden.
Nach dem erfolgreichen Ladevorgang wird dann die Statusanzeige den Namen der Überführungstafel (vergeben in der Kopfzeile des SDL-Programms) sowie den damit verbundenen Nachbarschaftsindex melden.

Um eine der beiden oben beschriebenen Aktionen durchzuführen, klicken Sie auf den Button `Sigma`. SCARLET öffnet dann ein weiteres Fenster, das in seiner Funktionsweise einem „doppelten" Retina-Datei-Öffnen-Fenster ähnlich ist. Sigma

Die rechte Hälfte des Fensters dient der Auswahl und der Kompilierung einer SDL-Programmcode-Datei, während die linke Hälfte dazu bestimmt ist, eine kompilierte Funktionstafel zu laden. Durch die Filtereinstellung wird die

Verwendung von Dateinamen mit Endung `sdl` für nichtkompilierte und `sim` für kompilierte Dateien nahegelegt, jedoch nicht erzwungen.
Nach der Auswahl einer geeigneten Datei erfolgt die Kompilierung nun durch Betätigen des `Compile`-Knopfes, und das Laden einer Funktionstafel für die Simulation durch `Load sigma`.

Genaueres hinsichtlich der allgemeinen Bedienung des Datei-Öffnen-Fensters entnehmen Sie bitte dem Abschnitt 5.2.3.
Compile
Die Kompilierung kann auch über den `Compile`-Button im eigentlichen Programm-Fenster ausgelöst werden.

5.2.5 Unverträglichkeits-Meldung

Sollten Sie versuchen, zu einer vorhandenen Retina eine nichtkompatible (siehe 4.3) Überführungsfunktion zu laden, oder umgekehrt - sei es versehentlich, oder weil Sie beabsichtigen, zu einem neuen Zellularautomat überzugehen - wird SCARLET Sie mit einer Meldung darauf aufmerksam machen.

Entscheiden Sie sich dafür, die neue Datei nun doch nicht zu laden, so behalten Sie die alte Konfiguration bei und wählen `Drop new`; klicken Sie dagegen auf `Drop old`, so erreichen Sie, daß SCARLET die vorhandene inkompatible Komponente selbständig deaktiviert.

5.2.6 Anzeige-Optionen

Sofern Sie bereits eine Retina geladen haben, wird SCARLET einen gewissen Teil davon im Simulationsfeld optisch wiedergeben. Dargestellt wird ein 2-dimensionaler Ausschnitt der Retina, so daß sich zunächst die Zelle mit der Koordinate $(0, ..., 0)$ in der linken unteren Ecke befindet. Die Größe des gezeigten Gebietes richtet sich nach den gewählten Anzeige-Optionen und wird gleichzeitig der Größe des SCARLET-Programm-Fensters angepaßt. Beim Vergrößern oder Verkleinern des Fensters bildet die Zelle in der linken unteren Ecke jeweils einen Fixpunkt: Ihre Lage im Simulationsfeld verändert sich nicht.

Jede Zelle der Retina wird durch ein Kästchen symbolisiert, das, solange diese Einstellungen nicht verändert werden, gewisse Informationen über die **Position** der Zelle im Raum (siehe weiter unten), sowie den kompletten **Zustand**, aufgeteilt in die verschiedenen Register, angibt.

Gleichzeitig finden Sie die verwendeten Registertypen noch einmal am linken Rand des SCARLET-Fensters angegeben. Durch Zurücknehmen oder Setzen einzelner Häkchen per Mausklick erreichen Sie, daß die entsprechenden Register in der Retina-Darstellung ausgeblendet oder angezeigt werden.
Die Kopfzeile jeder Zelle enthält deren Position bezüglich der auf dem Bildschirm dargestellten Koordinatenachsen. Dabei bezieht sich die erste Angabe auf die x-Richtung (waagerecht) und die zweite auf die y-Richtung (senkrecht). (`d3:10,d2:4` besagt z.B., daß die Dimensionen 3 und 2 das Bildschirm-Koordinatensystem bilden, bezüglich dessen die betreffende Zelle die Position $(10, 4)$ einnimmt.)
Deaktivieren Sie `Header` (ebenfalls am linken Rand zu finden), so erscheinen die einzelnen Zellen ohne diese Positionsangabe.

Direkt unter diesem Block finden sich zwei weitere Anzeige-Optionen: `Ruler` und `Wrap`.

Entfernen Sie die Markierung bei `Ruler`, wenn das **Lineal**, das die Lage in den ersten beiden Koordinatenrichtungen angibt, nicht im Simulationsfeld erscheinen soll. Umgekehrt leistet es beispielsweise dann wertvolle Dienste, wenn die einzelnen Zellen ohne Positionsangaben gezeigt werden sollen.

Die Funktion `Wrap` steht Ihnen nur zur Verfügung, wenn Sie eine 1-dimensionale Retina verwenden. Sie bewirkt, daß die „einzeilige“ Retina jeweils am Zeilenende umgebrochen wird, so daß der zur Verfügung stehende Platz im Simulationsfeld optimal ausgenutzt wird.

Ist `Wrap` aktiv, so präsentiert sich das Lineal in geringfügig veränderter Funktionsweise: Addieren Sie die Werte der vertikalen und horizontalen Achse, um die Koordinate der betrachteten Zelle zu bekommen.
`Wrap` ist - sofern möglich - beim Programmstart eingeschaltet.

Darüberhinaus befindet sich ebenfalls auf der linken Seite des Fensters eine Steuereinheit, über die geregelt werden kann, welcher Ausschnitt der Retina im Simulationsfeld angezeigt werden soll. Sie sieht zu diesem Zweck für jede einzelne Raumdimension eigene Steuerelemente vor.

Bestimmen Sie den gewünschten Ausschnitt, durch folgende Aktionen:

1. Wählen Sie die beiden (bevorzugten) Dimensionen, die die **Koordinatenachsen** auf dem Bildschirm bilden sollen (`Screen`). - Die Werte der übrigen Koordinatenrichtungen sind für alle dargestellten Zellen konstant (`Fixed`).

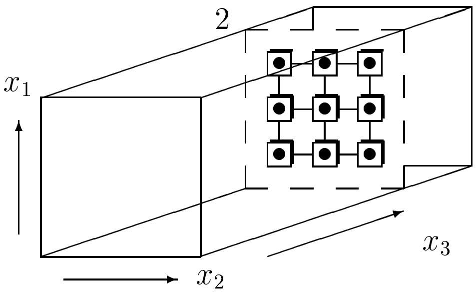


Abb. 3: Wirkung der Ausschnitts-Steuerung: Schnitt durch eine 3x3x3 Retina bei fixierter 2. Komponente (Wert 2).

Standardmäßig wird die erste Dimension in x-Richtung (also waagerecht) und die zweite (falls vorhanden) in y-Richtung (also senkrecht) dargestellt.

Sie können diese Einstellungen ändern, indem Sie das **Bezeichnungsfeld** der Dimension anklicken, die zur neuen x-Achse werden soll. - Die ehemalige x-Achse wird dann automatisch zur neuen y-Achse!

(Wollen Sie also beide Koordinatenachsen neu festlegen, so bestimmen Sie durch Anklicken zunächst die neue y-Richtung und dann erst die x-Richtung.)

2. Legen Sie die Koordinate der Zelle fest, die in der linken unteren Ecke des Simulationsfeldes erscheinen soll.

   Stellen Sie die gewünschten Werte durch direkte Eingabe über die Tastatur im Eingabefeld oder durch Anklicken der $\boxed{+}$ und $\boxed{-}$ Buttons ein. (Verwenden Sie als direkte Eingaben nur ganze Zahlen; eventuelle Vorzeichen sind der Zahl direkt voranzustellen.

5.2.7 Simulation

Run

Nachdem Sie eine Überführungsfunktion geladen und kompiliert sowie eine kompatible Retina hinzugefügt haben und bereits im Vorfeld alle Hürden bei der Programmierung erfolgreich genommen haben, kann es nun endlich losgehen: Sie setzen die Simulation - ganz unspektakulär - durch Anklicken des `Run`-Buttons in Gang! ... bis sie beinahe augenblicklich wieder zum Stillstand kommt, denn jede Betätigung des `Run`-Knopfes führt nur soviele Konfigurations-Transformationen aus, wie durch den Schrittweite-Regler vorgegeben (siehe 5.2.8). Der default-Wert beträgt `1`.

An der Anzeige `Elapsed time` können Sie erkennen, wieviele Schritte bereits durchgeführt wurden. Offiziell ist SCARLET maximal auf etwa $2^{31} - 1 \approx 2,14 \cdot 10^9$ Schritte ausgerichtet.

5.2.8 Simulations-Optionen

Schrittweite verändern

Vielleicht sind Sie aber gar nicht daran interessiert, alle Konfigurations-Übergänge der Retina im einzelnen zu verfolgen. Dann können Sie diese überspringen, indem Sie in das Feld `Time to run:` am oberen Rand die gewünschte Schrittweite (d.h. die Anzahl der Simulationsschritte, nach denen SCARLET die Konfiguration erneut zeichnen soll) eintragen. Erlaubt sind auch hier positive ganze Zahlen bis $2^{31} - 1$. Bestätigen Sie Ihre Eingabe mit Return.

Anstatt die Schrittweite direkt anzugeben, können Sie aber auch den Schrittweite-Regler darunter verwenden: Klicken Sie dazu den **Regler-Balken** an und halten Sie die linke Maustaste solange gedrückt, bis Sie die gewünschte Stellung erreicht haben. Der Regler deckt nach Programmstart zunächst Werte zwischen 1 und 100 ab; durch Eingaben in `Time to run` kann der Maximalwert jedoch verändert werden.

Literaturverzeichnis

[1] Ackermann, Kurt: Programmieren in C, Skriptum, Univ. Giessen, 1995

[2] Banahan, Mike : The C-Book, Addison-Wesley, Wakingham, 1988

[3] Eigen, Manfred; Winkler, Ruthild: Das Spiel, Piper, München, 1975, S.205-244

[4] Gutowitz, Howard (ed.): Cellular Automata, Theory And Experiment, MIT-Press, 1990

[5] Kutrib, Martin: Parallele Automaten, Bericht Nr. 9401, AG Informatik, Univ. Giessen, 1994

[6] Toffoli, Tommaso; Margolus, Norman: Cellular Automata Machines, A New Environment for Modeling, MIT Press, 1987

[7] Worsch, Thomas: Programming Environments for Cellular Automata, technical report 37/96, Dept. of Informatics, Univ. Karlsruhe

Index