SCARLET 0·85

A Programming Environment for Cellular Automata

—

A Manual

Institute of Informatics
Justus-Liebig-University, Giessen

May 1998

Contents

Introduction

"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)

When A. Turing in 1937 published his mathematical model of a computation universal machine, he gave birth to a rapidly growing theory: Other calculs as powerful as the so-called Turing machine were developed soon. About ten years later J. v. Neumann inspired by S. Ulam presented the world his theory about "selfreproducting automata". It was his aim to show a more practical way of constructing such computation universal machines. Because of this suggestion he became the official inventer of cellular automata (CA):

Roughly speaking he started out from a finite system which is discrete in time and space and represented by finite state variables, called cells. The cells are located in a finite and connected subset of a suitable Euclidean space, for instance $\mathbb{Z}^n$. Then the (global) evolution of the system is determined by the (local) behaviour of its cells. Furthermore he assumed the system to be homogeneous in space, that means that every two cells behave the same way under the same conditions, which are given by the states of certain cells in the "neighbourhood". The structure of the "neighbourhood" is supposed to be identical for all cells as well. Like that, the whole system can be described by a local transition function, which is the same for all cells.

In spite of this promising idea, there was paid little attention to CA for quite a long time. Because of a lack of communication they were invented several times under different names, and attempts to learn more about their properties went unnoticed to other researchers, who sometimes did the same work again.
The situation changed only in 1970 by coincidence, when J. Conway attracted the interest of many nonscientists, too, in a very special CA: the "Game of Life". At the same time, he focused once more on an important application

of CA: the simulation; a trend, which was also supported by the developement of more powerful computer systems.

From then on up to now, some old as well as important theoretical questions on CA could finally be answered, for instance, concerning their inversibility, or their properties on language recognition, and so on.

But there were progresses made concerning simulation, too. While it had been the main issue to investigate the global behaviour of CA in order to classify them in the beginning, the inverse problem became more and more of interest, especially for the natural sciences. They tried to find out which of those CA models could be used to describe a piece of reality, for example a physical system, which is controled by certain laws of nature, which can usually be described by partial differential equations.
Here the developement of models for particle movements in gases (lattice gases) could be mentioned. Inspite of these and other successes, a lot of important connections between CA and their continuous counterparts are still lying in the dark.

Throughout the years, several modifications and generalizations of v. Neumann's standard CA model were created to serve special needs: the modeling of fluids or gases even lead to the construction of *probabilistic CA*, which can exhibit distinct behaviours with certain probabilities. Moreover we can find CA that are not simultaneously updated (*asynchronous CA*), or that are *inhomogeneous* concerning different aspects. Analoguous to other concepts of computational theory *nondeterministic CA* were investigated, even *hierarchical CA* are used, just to name some types.

Although the abstract model on which CA are based is very simple, the behaviour they are capable to show can be so complex that it became almost enevitable to employ simulation systems to deal with both theoretical and practical issues. Except for the cellular automata machines (CAM) developed by T. Toffoli and N. Margolus to increase efficiency, most of those commercial or noncommercial systems are just supported by software and designed for different kinds of computers and operating systems. At the same time their intentions vary: Some of them are restricted to special forms of the standard CA (e.g., by just allowing certain neighbourhoods or space dimensions). Others are mainly meant to model generalizations; there can be found a whole lot of software packages that support probabilistic rules or spatial inhomogeneity; and there exist even more creative concepts, but especially those are often not intended for higher space dimensions, though.

SCARLET at its present version 0.85 allows the simulation of standard CA with almost no restriction at all. Besides it permits the modeling of generalizations in a "natural manner" (using additional states and the built-in random function).

This manual is intended to be a guide for everybody who has somehow heard about SCARLET and who would like to learn how to use it in order to solve serious scientific problems or simply because of a personal interest in CA.

So, we hope you enjoy SCARLET and wish you a lot of success!

Chapter 1

Cellular Automata

1.1 Formal Definitions

In order to give readers who are not very acquainted to CA a chance to come to remarkable results in SCARLET after a short time, we are going to introduce CA by some basical definitions and to explain it with some examples before looking at SCARLET itself. Readers who are already used to CA this chapter may serve to make sure of the notation used here. Most of the definitions given here are taken from the report "Parallele Automaten" of Martin Kutrib[5].

We approach the definition of CA by examining the definition of cellular spaces first:

A 5-tuple (S, d, N, σ, q_0) is called ***cellular space****, if it holds:*

1. *S is a finite state set, which is not empty.*
2. *$d \in \mathbb{N}_0$ is the space dimension.*
3. *N is a n-tuple of pairwise different elements of $\mathbb{Z}^d$, $n \in \mathbb{N}$, called* ***d-dimensional neighbourhood of order n****.*
4. *$\sigma : S^n \longrightarrow S$ is the local rule function.*
5. *$q_0 \in S$ is the quiescent state, which suffices: $\sigma(q_0, \ldots, q_0) = q_0$.*

Are you already in the mood to give up? We hope not, because although precise definitions are sometimes a bit abstract and difficult to see through, the idea of CA hidden somewhere between the lines, is amazingly concrete:

First of all we consider a single Moore-automaton, which has got

- a finite and non-null state set S - exactly like the cellular space,
- a mapping β that calculates an output just depending on the state the automaton has got at that moment,
- a mapping λ that computes the state of the automaton in the next time step having the current state and an input as arguments.

(Furthermore Moore-automata have got initial and final state sets, too, but we will ignore that, since it does not play an important part in the following explanations.)

"Could you please repeat at which point exactly I should have got the idea of it???" Really, did it not become clear? Whenever we speak of an automaton in daily life, we most of the time actually mean an object that can theoretically be described by a Moore-automaton!

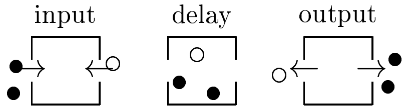
You cannot believe that? Perhaps the following example will convince you:

Example 1: The "'Input/Output Automaton"'(I/O-A)
Consider a little machine in a box with two holes, each of them on opposite sides, such that the machine can fulfil only one (boring) task: to eject balls that have been inserted on one side, one by one on the other side after a small delay, where no more balls will be accepted. The machine permits to use both holes as slots.

Now we are going to show that we can model the I/O-A as a Moore-automaton. First of all we need a suitable state set. Observe that a state is, for example, conveniently specified by the number and origin of balls within the automaton during the time of delay ("no ball"; "one ball from the right, two from the left", and so on). Like that the set of all possible states S is not empty, but finite for there can only be a finite amount of all balls within the automaton at a time.
Further, states and outputs are related to each other in a unique way (β exists and is bijective):

It is easy to comprehend that there exists the same relationship between the state set and the set of inputs; in particular we have the mapping λ as postulated above. (In this simple example it is just depending on the input, not on the current state.)

"Ok, ok; but what have Moore-automata to do with cellular spaces?" That will be the next step; in fact a cellular space consists of an infinite number "two balls from the right, one from the left" $\longrightarrow$ "eject two balls to the left and one to the right"



"no ball" $\longrightarrow$ "eject no ball";

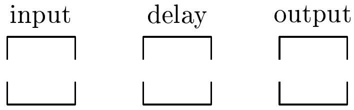


Fig. 1: Different possible states of I/O-A

of automata of the same type, which are communicating with each other. - But let us return to our example:

Example 2: One-dimensional I/O-A cellular space:
Assume now an infinite amount of I/O-A lined up such that the slots of each two automata standing next to each other touch. Now their positions could be labeled with elements of $\mathbb{Z}$; in other terms: The I/O-A are ordered in the one-dimensional space $\mathbb{Z}$.
Besides we consider all I/O-A to be in the same rhythm of time, that means the periods of input, output, and delay happen for all of them to be at the same time. By consequence of this construction, each automaton has got a left and a right neighbour, whose outputs will determine its input. Conversely its own output will influence the inputs of other I/O-A. Because of the fact that β is bijective, we can identify the output of an I/O-A with the state it has at that moment and could, therefore, say:

> *The state of an I/O-A in the next time step is completely determined by the current states of its neighbours.*

(Since cellular spaces are always constructed of Moore-automata with bijective output function β, it is not longer necessary to speak of input and output. It is convenient to care about the states instead.)

Arriving at that point we have already found a local rule function $\sigma : S^n \longrightarrow S$ that seems to be convincing: Starting from two states it computes a new state in such a way that this is the state of an I/O-A, if the arguments have been the preceding states of its left and right neighbour. Hence the neighbourhood is of order two and the quiescent state q_0 finally is the state "no ball", for an I/O-A will not have a ball, if its neighbours have not had one in the previous time step.

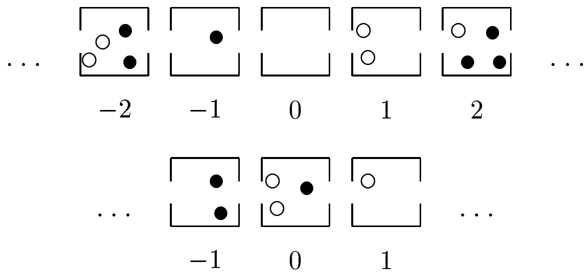


Fig. 2: Two timesteps in the evolution of an I/O-A cellular space.

"But who tells us we've chosen the right function σ? The definition doesn't tell anything about that! Anyway, what about this strange n-tuple that is said to be the neighbourhood? It seems that it isn't useful at all, otherwise it would have been already mentioned?!"

Just a moment! These points are still unclear, because we have always emphazised that cellular spaces are accumulations of single automata with a high degree of independence. But in fact one focuses on the cellular space as a whole, with a homogeneous behaviour and regards the automata only as its components. (That is the reason why they are also called ***cells*** (or ***sites***).) The following definitions allow the use of such a point of view that looks at a cellular space in its totality, and, therefore, complete our explanations:

A mapping $c_t : \mathbb{Z}^d \longrightarrow S$ is called ***configuration*** *of the cellular space at time step $t \in \mathbb{N}_0$. c_0 is also labeled* ***initial configuration****.*
$c_t(i)$, $i \in \mathbb{Z}^d$ is called ***state*** *of the automaton i under the configuration c_t.*

Suppose that A denotes an alphabet, $N = (a_1, \ldots, a_d)$ a d-dimensional neighbourhood, and C_d the set of all configurations $c : \mathbb{Z}^d \longrightarrow A$. A mapping $\mathcal{T} : C_d \longrightarrow C_d$ is called ***global mapping****, if it is induced by a local rule function σ as follows:*
For all $c \in C_d$ is valid:

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

The most important tool to look at a cellular space as a whole seems to be the configuration: Each configuration presents the states of all automata at a certain time step. Consequently now we are looking for a (global) mapping $\mathcal{T}$ that will convert in a suitable way whole configurations. We have found the right one, when its global actions can be reduced on the local effects of σ on every cell. - Obviously this is only possible, if the elements of the cellular space, the automata, are homogeneous!

Example 2: Resumption:
If we choose the 2-tuple $N = (-1, 1)$ as the neighbourhood, we obtain for all $i \in \mathbb{Z}$ and $t \in \mathbb{N}_0$:

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

Taken in account that σ applied to the states of the left and right neighbour yields the state of the I/O-A in i in the following time step, thus $c_{t+1}(i)$ with c_{t+1} the successor configuration of c_t will lead us to the conclusion that the global mapping $\mathcal{T}$ exists. At the same time, the example should have helped to make the connection between $\mathcal{T}$, σ, and N a bit clearer.

In the example given above, those cells that had an influence on the successor state of another cell were also its spatial neighbours. Because that is a general (though not necessary) phenomenom, one speaks just of "neighbours with regard to a certain neighbourhood", when those cells which are the arguments of σ for a certain site, are meant, or in more formal terms:

Suppose that (S, d, N, σ, q_0) is a cellular space. Let $i, j \in \mathbb{Z}^d$. Then, a cell i is called ***neighbour*** *of cell j* ***with regard to N****, if there exists a component k of N such that $i = j + k$.*

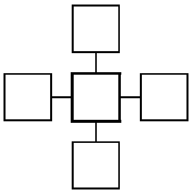
In other words, the neighbourhood indicates the relative positions of an arbitrary cell to its neighbours.
In particular a cell may be its own neighbour, if N contains the component $(0, \dots, 0) \in \mathbb{Z}^d$.

Exp.3: A quite common example of a d-dimensional neighbourhood is the so-called "general von Neumann neighbourhood"

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

For $d = 2$ and $k = 1$ we obtain the following structure:

H_1^2 :



Unlike the Moore-automata mentioned in the beginning, the main purpose of cellular spaces is not the construction of real existing machines, as well as software, which is e.g., capable of pattern recognition and so on, but the discrete simulation of continuous systems, as known, for instance, from the natural sciences, for both a better comprehension of the theory of dynamical systems, and practical calculations.
Therefore, not seldom cells are intended to model discrete points of a system in $I\!R^3$; states contain discrete physical quantities or they just tell, if there is a particle located in that point; and the local rule is meant to imitate some laws of nature.

Because all real existing systems are subject to certain material restrictions - probably this is less obvious for real systems than for simulation systems - we are usually more interested in a finite form of cellular spaces: the cellular automaton (CA).

Suppose that $M = (S, d, N, \sigma, q_0)$ is a cellular space. M is called ***cellular automaton*** *if the following is true:*

1. *There exists $\# \in S$ such that for all $t \in I\!N_0$ and for all $i \in Z\!\!Z$:*

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

   $\#$ is called ***boundary value****.*

2. *Suppose*
   $B := \{l \in Z\!\!Z^d \backslash R :$ *There exists $i \in R$, such that l is a neighbour of i* $\}$
   A finite, not empty domain $R \subset Z\!\!Z^d$, called ***retina*** *(or* ***lattice****) has the following properties:*

   - *For all $i \in R$: $c_0(i) \neq \#$*
   - *For all $k \in B$: $c_0(k) = \#$*

- *For all* $k \notin B \cup R$: $c_0(k) = q_0$
- *For all* $i, j \in R$: *i is neighbour of j with regard to* H_1^d *- directly or with mediator neighbours in between - or the other way around.*

In contrast to cellular spaces the area of interest of a CA lies within a fixed subset of $\mathbb{Z}^d$, the so-called retina. The last condition of 2., which demands that the cells are organized in undisrupted lines along the coordinate axes, guarantees that the set is a connected domain.
It is bounded by the set B, that means that B contains all cells which are neighbours of cells within the range of the retina, but which are not lying in the interior themselves. Therefore, B is called ***boundary*** or ***border***. All cells on the boundary - and only those - remain in a particular state # all the time; so their influence on the retina is always the same.
Furthermore those cells lying behind the boundary are - because of the definition of B - neighbours to no cell in retina and, therefore, do not have any influence on it.
If we are only interested in the behaviour of a cellular space within a certain area, we do not need to hesitate to forget the world behind the boundary. We could assume, for instance, that all cells there have got the quiescent state at $t = 0$.

1.2 Cellular Automata in SCARLET, Examples

1.2.1 General Features of SCARLET

The software package SCARLET 0.85, which is intended for simulations of cellular automata, may be used on all computers where the operating system SUN Solaris, version 2.0 or bigger, is installed.
From a X11 window interface the user can load a CA and controle the simulation. During the simulation the user can choose a two-dimensional sector of the retina that will be displayed by SCARLET. It contains information about positions and states of the shown cells.

To specify the CA which should be simulated, SCARLET - like many other software packages of this kind - offers an own programming language consisting of two components (RDL and SDL), whose syntax is similar to the one of C. While SDL is chiefly used to design a table for the local rule function σ and to specify a proper neighbourhood, RDL is used to describe initial configurations. Only later SCARLET will take two matching RDL and SDL programs stored in different files to create a complete CA.

The software package SCARLET permits the user to model any standard CA with any cube shaped retina he could think of, if it does not exceed the available ressources of the computer system. Moreover it provides a built-in function to gain random numbers, which may be a powerful tool to create probabilistic CA as well (see example 17, pg. 44).

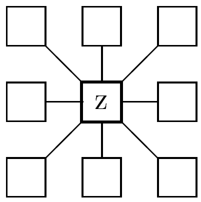
Other generalizations of the standard CA are not directly supported. But nevertheless it is often possible to reduce them to a standard CA using additional states or neighbours. At that point, SCARLET helps the user providing a possibility to use compounded data structures as state set (compare with 33).
Like that CA which are *inhomogeneous in time* or *space* could be designed adding a component that contains information about the time or the cell position (see example 20, pg. 46) to the state structure, and modifying the local rule table.
Enlarging the neighbourhood will also allow to simulate simple kinds of *variable boundary conditions* (compare to example 23, pg. 48).

1.2.2 Examples of Programs

In conclusion of the first chapter we would like to give the reader a first impression of SCARLET'S features. Therefore, the reader is certainly not expected to understand every detail of the programs shown in this paragraph. It seems to be more important to get a general idea of their structure, a main connecting thread, which may guide through the manual.

Example 1 ("Life")

Let us start with a famous example which helped a lot to make the theory of CA almost popular: The "Game of Life" invented by J. H. Conway in 1970. It was his intention to investigate the evolution of an idealized population of living beings. It does not matter much that we cannot implement "Life" in a two-dimensional cellular space, as it was suggested by Conway; we will take a retina that is finite, though big enough instead. Furthermore we use two states `0` and `1`: A cell will represent a living organism, if its state is `1`, and a dead one, if the state is `0`. The neighbourhood will include the cell z refered to (generic cell) itself, together with the eight surrounding cells.



The evolution of the population obeys the following rules:

- if exactly three cells in the neighbourhood are "alive", then the generic cell will become "alive" (state 1), too.
- If exactly four neighbours are "alive", the generic cell will keep its old state.
- In all other cases the generic cell will be "dead" (state 0).

Though these are very simple rules to model phenomenoms like the deadly consequences of a too high or too low population density, or the birth of new life under good conditions, the behaviour of such a system is already so complex that it could hardly be overseen. One reason for that surely is that the chosen rules are very balanced.

```
SIGMA life;
DIMENSION 2;
REGISTER INT state;
NEIGHBOUR center       :  (-1,-1);
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; :  /* no statement */ :

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

The program given here can be recognized as a SDL program by the keyword `SIGMA` in the headline. It desribes what the local rule function will do to a single but arbitrary cell of the retina during one step of the simulation. The program consists of two parts: The first part reachs until `TABLE` and includes several declarations:

- The space dimension is set on `2`.
- In SCARLET the state set is subdivided into smaller units, the registers. We distinguish between two types which have nearly the same ranges as string and signed integer variables in the programming language C. To model the states `0` and `1`, we use an integer register and label it `state`.

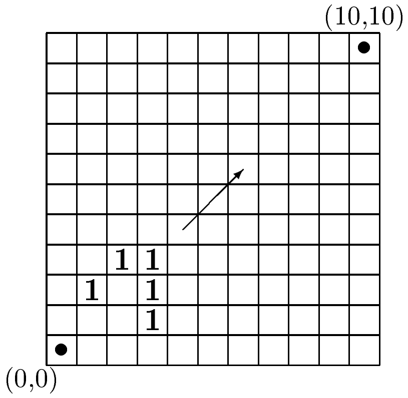
- Like in the abstract model, neighbours are indicated by relative positions and can be accessed by symbolic names, which are chosen by the user.

- An integer variable `sum` is defined,

- and finally the device `BORDER {0};` will cause all cells on the boundary to remain in state `0` during the whole simulation (what seems to be a proper assumption to our model). Besides, boundary cells could in principle be treated like cells in the interior.

The second part shows the rules of evolution as kind of some (here: three) nested IF-THEN-ELSE statements (entries). Notice that all IF, THEN and ELSE parts are separated from each other by colons : . Everywhere in the statement part, assignments to variables are allowed; Here, the total number of neighbours with state `1` is stored first in `sum`. Later `sum` is used to choose a proper action.
Observe that a non existent condition - like in entry three - equals `TRUE` and that the generic cell is left invariant, if there is no statement in the consequence part - like in entry two.

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



The RDL program shown above describes an initial configuration that matches the SDL program `Life`. Again we find a subdivision into a declaration and a statement part:

- The chosen retina, which must always be a cube in SCARLET, is located in $\mathbb{Z}^2$ and has got the corners $(0, 0)$, $(0, 10)$, $(10, 0)$, and $(10, 10)$. (Just two diametral ones must be indicated.)

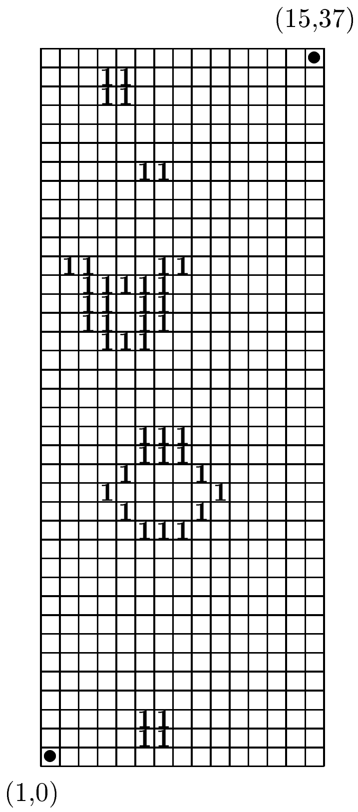
- Corresponding to `SIGMA Life` there must appear an integer register in this program as well, which has got the same name `state` by coincidence.

- The keyword `START` signals the beginning of the statement part, where `1` is assigned to several cells accessed by their position. As shown here it is possible to assign the whole state surrounded by {} (first and third line) as well as to store values to single registers (second line). The construction in the third line causes all cells within the rectangle, with corners given there, to be treated together.
  The remaining cells are already in the desired state `0` because of the initialization of the retina, which is automatically done by SCARLET.

Applying the local rule `Life` to this initial configuration, the figure consisting of state-`1`-cells will appear again after five time steps, but shifted up and to the right one cell each.

By the way, the following strange-looking initial configuration answered the question raised by Conway, if there exist populations whose size will approach infinity in time, assuming that there is enough space. (Hint: The solution is close connected with the first figure.)

```
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};
```



Remark:
The device `[5,15] (4) [9,15] = {1};` assigns `1` to a couple of cells within the rectangle `[5,15] .. [9,15]`: exactly to every fourth cells in each coordinate direction starting with the cell in `[5,15]`.

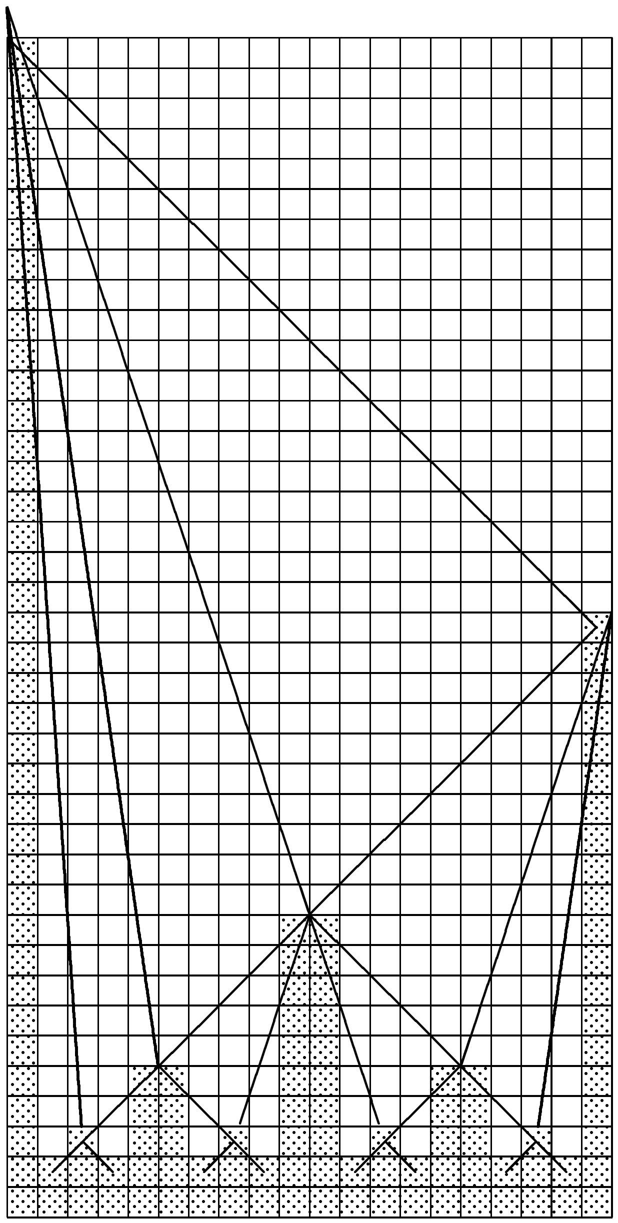
Example 2 ("FSSP")

The second example we are going to see is as well famous, though a bit more complicated, and dates from the early years of CA. It is about the problem, if it is possible (and in case of yes, how) to synchronize a number of lined up automata, when each of them is only able to communicate with its adjacent left and right neighbours, and the synchronization signal is given by one of the automata on the boundary. (Not very seriously meant, the same question could be posed with a company commander with hoarse voice and his soldiers (helpfully transmitting in a whisper); that is the reason why the issue is called **F**iring **S**quad **S**ynchronization **P**roblem.)

The method suggested here goes back to a time optimal solution invented by A. Waksman in 1966 (compare with [5]). The underlying idea is to bisect the retina generating new generals at the same time again and again, until all cells are generals and know by that fact that they have to fire in the next time step. The bisection is done and controlled by signals at distinct speeds; To gain these signals will actually be the challenge of FSSP.

First of all, the general on, say, the left boundary sends out a main signal, which moves to the opposite side at speed 1 (= one cell per time step) and will be reflected there producing a second general on the right boundary.
Additional signals are to be sent out by the first general such that they will happen to meet the main signal on its way back, entering the first half (the first quarter, the first eighths, ...) of the retina.
If the length of the piece of retina they divide in two is an odd number, this will produce one general in the middle, and in case it is even-numbered, two. (In the first case we call the main signal, which is carrying the information about that, 1G-signal, otherwise 2G-signal.)
The new generals are recursively responsible for their (smaller) domains.

Exp.4: If we watch the evolution in time of a one-dimensional retina consisting of 20 cells, we will see the following: (Each line of fields represents the appearance of the retina in a certain time step. The fields shaded stand for the generals.)



Considering the example raises the following questions:

1. How do we generate control signals that have the properties we demand?

2. How do we achieve that main and control signals work together well, knowing that the reflection of a main signal at one general will take less time than at two generals? (Remember that the solution should be optimal concerning time!)

In case the main signal is reflected at two generals, the n-th control signal has to move 1 field, while the main signal will cover $2^{n+1}-1$ fields $(1:(2+1)$, $1:(4+3)$, $1:(8+7)$, ...). However the control signal must reach the middle at latest at the same time as the main signal. Considering this will lead us to the method of Waksman:

Every second time step the main signals send out a control signal in the opposite direction, while the generals send out a constant signal. When a control signal arrives at a constant signal, it will shift it one field away from the general, who will refresh the constant signal in the next step. Besides, a constant signal lets every second control signal pass so that it can influence constant signals coming behind.

The answer on the second question is actually given by the picture: If there will be only one general (thus the main signal will move one field less) it is pretended that the control signal was sent out one step earlier; in other terms, the control signals are sent out one time step earlier by 1G-signals than by 2G-signals.

Exp.4: (Resumption)
The next figure shows again the time evolution of the 20 fields retina, but now in a realization with 10 states.

We use different generals `"G"` or `"g"` depending on if they should send out 1G- or 2G-signals. Soldiers are represented by the state `""`, control signals by `">"` or `"<"` respectively, constant signals change their state from `"1"` (permeable) to `"2"` (impermeable) and back. The main signals are symbolized by `"L"` and `"R"`. They "come in two lines" in order to make clear, if they are 1G- or 2G-signals. At the same time the second line carrys the information whether or not the distance between two generals was even-numbered. Finally `"f"` denotes the firing state.

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

Now it is time to present the complete SDL program belonging to the solution outlined above. In spite of its size no difficult programming structures are used.
The different states are stored in a string register `state`. To improve the clarity of the code, the register contents of the three neighbours involved are written in variables, first.

A typical entry of the table provides an assignment to the generic cell for single neighbour configurations (e.g., `l=="G"; c=="g"; r=="G";`) or for certain sets of configurations (e.g., `c=="g"; l=="G";`) together. In cases that are not mentioned by the table, the state of the generic cell does not change. This time we do not use the notation with `SELF` known from "Life", which requires a cell expression, but we directly assign the values to the register `state`.

Dealing with bigger local rule tables like this, special care must be taken about the order of the entries. However, entries with more special neighbour configurations must precede more general ones if they include the special case. Otherwise, the special ones will have no effect because of the IF-THEN-ELSE structure of the table.

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

TABLE
/* Assignment to variables */
     /* (always done for it is the first part of the first condition) */
l=left.state; c=center.state; r=right.state;
/* fire */
l=="g"|| l=="G"; c=="g"|| c=="G"; r=="g"|| r=="G"; :
     state="f"; :

/* the step before the last one */
l=="g"|| l=="G"; r=="g"|| r=="G"; :
     state="G"; :

/* creation of a 1G-signal (conversion of the general) */
     /* by a 2G-signal (2 generals) */
c=="g"; l=="G"|| r=="G"; :  state="G"; :  /* not 3 generals */
     /* by a 1G-signal (1 general) */
l==""; c=="g"; r==""; :  state="g"; :
l==""; c=="g"; r=="L"; :  state="G"; :
l=="R"; c=="g"; r==""; :  state="G"; :

/* producing a general in the middle */
     /* the first general always */
```

```
c=="1"|| c=="2"; l=="R"|| r=="L"; :  state="g"; :
     /* the second general only if there arrives a 2G-signal */
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"; :  /* length==4 */
l=="g"; c=="R"; r=="1"|| r=="2"; :  state="g"; :
     /* otherwise:  first general receives the information */
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"; :  /* length==5 */
l=="G"; c=="R"; r=="1"|| r=="2"; :  state="1"; :

/* creation of a general on the boundary */
l=="R"; c==""; isborder(right); :  state="g"; :
isborder(left); c==""; r=="L"; :  state="g"; :
/* conversion of the general on the boundary for a 1G-signal */
l=="R"; c=="g"; isborder(right); :  state="G"; :
isborder(left); c=="g"; r=="L"; :  state="G"; :

/* controling of 1G- and 2G-signals */
     /* creation, reflection at the boundary */
l=="g"; (c=="2"||c=="L"||c==""); :  state="R"; :
(c=="2"||c=="R"||c==""); r=="g"; :  state="L"; :
     /* movement */
l=="R"; c==""; :  state="R"; :  /* isborder(right)==FALSE ! */
c==""; r=="L"; :  state="L"; :

/* More special:  1G-signals */
l=="G"; c==""; :  state="R"; :  /* creat.:  1G-signal */
c==""; r=="G"; :  state="L"; :
l=="G"; c=="R"; r==""; :  state="1"; :  /* creat.:  1. const.  signal */
l==""; c=="L"; r=="G"; :  state="1"; :
l=="1"|| l=="R"; c=="R"; r==""; :  state=""; :  /* movement 1G */
l==""; c=="L"; r=="1"|| r=="L"; :  state=""; :
c==""; r=="R"; :  state="<"; :  /* creat.:  control signal */
l=="L"; c==""; :  state=">"; :
l=="<"; c=="R"; r=="R"; :  state=""; :  /* recogn.:  1G-signals */
l=="L"; c=="L"; r==">"; :  state=""; :
l==""; c=="R"|| c=="L"; r==""; :  state=c; :
```

```
/* more special:  2G-signals */
l=="g"; c=="R"; r==""; :  state="<"; :  /* creat.:  1.  control signal */
l==""; c=="L"; r=="g"; :  state=">"; :
l==">"; c=="R"; r==""; :  state="<"; :  /* creat.:  other control sign.  */
l==""; c=="L"; r=="<"; :  state=">"; :
l=="<"; c=="R"; r==""; :  state=">"; :  /* recogn.:  2G signals */
l==""; c=="L"; r==">"; :  state="<"; :

/* control signals */
l=="G"|| l=="g"; c=="<"; :  state="1"; :  /* creat.:  const.  signals */
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="<"; :  /* passes 1-signals */
l==">"; c=="1"; :  state=">"; :
c=="2"; r=="<"; :  state=""; :  /* stopped by 2-signals */
l==">"; c=="2"; :  state=""; :
c=="2"; l==">"|| r=="<"; :  state=""; :
l==">"; c==""; :  state=">"; :  /* movement */
c==""; r=="<"; :  state="<"; :
c==">"|| c=="<"; :  state=""; :  /* does not meet 1,2,g,G */
```

Exp.4: (Resumption)
A RDL program that models a retina with 20 cells and a proper initial configuration (19 soldiers and the general) is simply given by

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

Chapter 2

Common Basis of RDL and SDL

Within the next three chapters, we will introduce the reader to the two most important tools for writing programs which will simulate a given cellular automaton (CA). First of all we will recall the essential parts of a CA:

- retina (finite subset of $\mathbb{Z}^n$)
- finite state set
- initial configuration
- neighbourhood
- local rule function σ

There are several reasons for separating the declaration of the retina and the initial configuration on the one hand, from that of the neighbourhood and the local rule on the other hand, in SCARLET. This proceeding allows us, for instance, to combine different "matching" automaton parts to get a complete CA, later. "Matching" means that the dimension of the retina and the state set must be the same in both definitons to permit a combination. We are going to discuss this later.

For the design of a CA, SCARLET provides two special programming languages: RDL (= **R**etina **D**efinition **L**anguage) and SDL (= **S**igma **D**efinition **L**anguage) - it is probably easy to guess which language belongs to which task?!
Now you will perhaps say: "Am I really forced to learn two languages to create a simple CA??" That is right, indeed, but the reader will soon detect

that they are not very different at all, on the contrary: Most of the basic concepts are used in both languages RDL and SDL. We go even further as to simply speak of SCARLET, if we mean both, RDL and SDL. In addition, readers, well acquainted with C, will soon notice that there exist similar or even identical concepts in SCARLET. Nevertheless we will give a (more or less detailed) description of all of them. Finally SCARLET is, because of its special task, not as large as other general purpose languages and, therefore, easier to learn!

2.1 Arrangements

In order to describe the components of the SCARLET programming language we only use such names for non-terminal symbols which are suggested by the formal grammar. We also use them, wheneverwe mean the whole class of non-terminals and not only a single element of it. Whenever such a name appears in the headline of a paragraph, the syntactical objects mentioned there are usually elements of the belonging class.

Var, Expression, Constant-Identifier

Sometimes it makes things easier to use names for more general classes than given by the grammatical rules:
The union of *IntExpression*, *StringExpression*, *CellExpression* and *BoolExpression*, for example, we will call *Expression*. Analogously we get the classes *Var* and *ConstantIdentifier*.

Listings

Instead of one or more objects of an arbitrary class *Class* separated by commas, we will sometimes write *ClassList*.

Syntax Diagrams

As soon as the syntax of a class consists of more than one terminal or non-terminal symbol, we give a syntax diagram to clarify the structure: Terminal symbols appear in ovals, while non-terminals are framed by rectangles. (In addition, terminals are always printed in typewriter font unlike the non-terminals which appear in the normal but italic font.)
A path through the graph which leads to valid grammatical objects starts with the arrow at the very left side, and follows always the direction pointed out by the arrows, until it ends at the point of the arrow on the right. We get an element of the described class by writing the terminal and non-terminal symbols one behind each other in right the order we passed them on a path through the diagram. The single symbols should usually be separated by blanks (compare to page 29).

Sometimes there will be phrases like "There can be an arbitrary amount of objects...". Then we mean, of course, that the number is only restricted by the size of the available saving medium.

2.2 Source Code Components

2.2.1 Character Set

A correct RDL or SDL source code consists of the following characters:

Small and capital letters (*Letter*) : a, b, .. , z, A, B, .. , Z
(without ä, ö, ü, Ä, Ö, Ü)
Digits (*Digit*) : 0, 1, .. , 9
Special characters : +, -, *, % , /, & , |, =, >, <, !, (,), [,], { , } , ., :,
,, ;, _ , ',
ä, ö, ü, Ä, Ö, Ü, ^, °, ", §, $, ?, \, `, ´, @, ~, #, ␣
(just in strings)

Separation signs: ␣, TAB, newline

Letters, digits and special characters together form the class *Character*.

Separation Signs

Separation signs have to be used whenever identifiers or keywords are standing after one another. In all other cases they can be omitted. In general, it does not matter which separation sign is used, but it certainly is of advantage to organize the source code in lines and paragraphs, for it is easier to keep control of it. It is especially recommened to design the local rule table in a way which allows the recognition of its entries (see pg. 93). Blanks, too, should be used to make it easier to read the code. But it is certainly necessary to keep tokens like identifiers, keywords and compounded assign operators (<=, ++ , ...) together, otherwise the compiler would not understand.
Also if we want a string to go on in the following line, we are not permitted to put separation signs between the backslash and the carriage return, see pg. 32.

2.2.2 Keywords

The following words are reserved in SCARLET:

`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`

We cannot use them as identifiers, e.g., for variables, symbolic constants, registers, or neighbours at all. Furthermore, it will be the same to the compiler, if we use small or capital letters.

2.2.3 Comments

In SCARLET it is possible to insert comments into the source code at nearly every position. This is done by

`/*` *comment* `*/`

It will cause the compiler to ignore all characters except the combinations `/*` and `*/` from the beginning of the comment up to the next end of comment. Therefore, comments may be nested, and since separation signs are ignored, comments can cover several lines.

Remark:

Strings

We cannot place a comment within a string, because `/* */` will simply be interpreted as part of the string.

2.3 Identifiers

Identifier

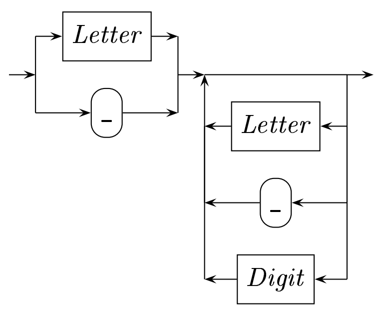
Whenever we want to label a new object, we are in need of an identifier. Such objects are usually symbolic constants or variables, but in SCARLET also registers, and neighbours, and finally the retina, and the local rule function themselves. In order to make clear which kind of object they shall label in further syntax diagrams, identifiers are often not simply called *Identifier* but somehow different (i.e., *IntConstantIdentifier*, *StringRegIdentifier*, ...)

Below we give a summary of all nonterminal symbols that are actually just identifiers and what they label.

Identifier for		**Nonterminal Symbol**
Constants:	Integer	*IntIdentifier*, *IntConstantIdentifier*
	String	*StringIdentifier*, *StringConstantIdentifier*
	Boolean	*BoolIdentifier*, *BoolConstantIdentifier*
	Cell	*CellIdentifier*, *CellConstantIdentifier*
Variables:	Integer	*IntVar, Identifier*
	String	*StringVar, Identifier*
	Boolean	*BoolVar, Identifier*
	Cell	*CellVar, Identifier*
Registers:	Integer	*IntRegIdentifier, Identifier*
	String	*StringRegIdentifier, Identifier*
Neighbours:		*NeighbourIdentifier*
Retinas:		*Identifier*
Local Rule Tables:		*Identifier*

Ensure that the same identifier is not used twice for different objects within the same program.

Syntax:

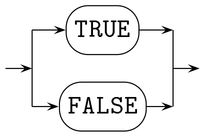


Legal identifiers consist of (at least one) capital or small letters (*Letter*), dashes, or digits (*Digit*). It is not allowed to use a digit at the beginning. Here, too, the compiler will not distinguish between capital and small letters.

2.4 Constants

2.4.1 Boolean Constants *BoolConstant*

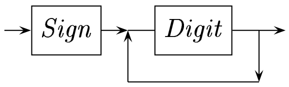
Syntax:



In SCARLET the boolean values are represented by the keywords `TRUE`, if it is meant TRUE and `FALSE` in case of FALSE. Boolean values are of data type `BOOL`.

2.4.2 Integer Constants *IntConstant*

Syntax:

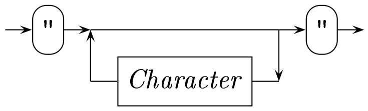


Integers in SCARLET are within a range from −2.147.483.648 up to +2.147.483.647. (This corresponds to an internal representation as a 32-bit word, where the first bit is occupied by the sign.) As usual, integer constants are written in decimal format with an optional `+` or `-` (*Sign*), which have the meaning of a positive or negative sign. An integer without sign is positive. Integers are of data type `INT`.

Remark:
Numbers are chiefly used to indicate absolute or relative cell positions within the retina as well as to name states. That is the reason why there is no need for rational numbers in SCARLET.

2.4.3 String Constants *StringConstant*

Syntax:



In SCARLET strings are words consisting of at most 255 characters. They are surrounded by quotation marks `" "` in order not to be confused with identifiers.
Strings are of data type `STRING`.

Long Strings

Sometimes it may have advantages to continue a string in the following line (e.g., because it does not fit the screen). In this case the first part of the string has to be finished with a backslash `\` at the desired end of line; the rest of the string must follow immediately at the beginning of the next line. Note that it is not permitted to place any character between the backslash and the line end symbol, neither blanks. Blanks at the beginning of the next line belong to the string.

Exp.5:
```
"This string doesn't come to an end in this line, \
but it goes on in the second line."
```

2.4.4 Cell Constants

[1]

We have already mentioned the importance of registers to form a finite state set, which is easy to handle. Instead of keeping all state information in one large variable, what will soon lead us to complex and confusing state indications, SCARLET provides a possibility to split them up into smaller units, called registers. There are two different kinds, according to SCARLET'S `INT` and `STRING` types: The first one contains integer values, described on page 32, the second kind is intended for strings (see page 32). (We call them integer and string registers respectively.)
At the beginning of each program, number and types of all used registers are fixed uniquely for all cells of the automaton. In addition they are put in order by their declaration, and each register is given a (symbolic) name.

[1]For more or less formal reasons, cell constants are not an independent part of SCARLET'S grammar, but a subset of all cell type objects *CellExpression*. (And there it belongs to the construction {*RegExpressionList*}.) Nevertheless we will give a description here, because it enables us to familiarize the reader already now with the concept of partioning the state set.

(See also paragraph 3.1.3 on page 68.)

Then the entire state of a cell is given by the cross product of the ordered tuple of its register contents. Because of the finite range of all registers the cross product - as the resulting state set - is finite, too.

Exp.6: Let us have a look at the following state set:

$$S = \{s_0, \cdots, s_n\}\,, \quad n \in I\!N$$

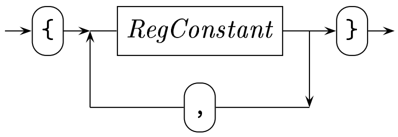
Certainly it can be modeled with a single string register. But we may also use an additional integer register and form the state set like that:

$$\{\, \{''s'', 0\}\,, \{''s'', 1\}\,, \cdots, \{''s'', n\}\}\;,$$

Now it is quite easy to change a state: We simply have to transform the integer value by an arithmetical operation.

The structure of a cell induced by the registers must, of course, be evident in the syntax of cell constants, too: The (constant) contents of the different registers stand in sequence (as fixed before by the declaration), surrounded by braces and separated from each other by commas:

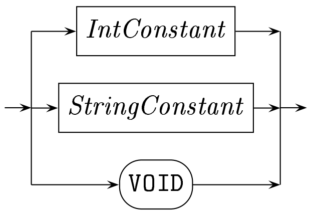
Syntax:



RegConstant

There has to be values for all declared registers according to their data types, which means: Normally *RegConstant* is an element of *IntConstant* if the corresponding register type is `INT`, otherwise it is an element of *StringConstant*. In both cases *RegConstant* may also be `VOID`.

RegConstant:



VOID

`VOID` does not belong to a special register type, since it is actually kind of a device. If we assign a cell constant with `VOID` in some of its registers to an arbitrary cell type object, the contents of all these registers to which `VOID` is assigned remain unchanged (see page 62). Cell constants are of data type `CELL`.

Exp.7: Now we think of a cell structure created of three registers with types `INT`, `STRING`, `INT` in right this order and some examples of cell constants. Which of them concur with the given structure?

$\times$	`{"word", -567}`	(wrong)
$\surd$	`{-123, "state", VOID}`	(right)
$\times$	`{0, , 0}`	(wrong)
$\surd$	`{0, "", 0}`	(right)

2.5 Symbolic Constants

ConstantIdentifier

Sometimes there are certain constants that would appear very often within the source code. Because it can be rather boring to type them again and again, SCARLET offers - as most of the high-level programming languages - a possibility to define symbolic names, which can be used as abbreviations. Here the name must be an element of the class *Identifier*. Constants which are represented by a symbolic name are called symbolic constants. Agreements of this type must occur in the declaration part of a (RDL or SDL) program. In consequence of its declaration, a symbolic constant automatically becomes an object of one of the types `INT`, `STRING`, `BOOL`, or `CELL`. Depending on this, we also speak of elements of the classes

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

Since their meaning is directly connected to the programming environment they are used in, we will discuss them in detail in the chapter about RDL on page 70.

2.6 Variables *Var*

SCARLET becomes a real programming language by putting variables at our disposal. They always have to be denoted by elements of *Identifier*. In contrast to many other programming languages (e.g., C), all variables have to be declared at the beginning of the program where they will be used in. For more details on this topic, please refer to page 70. The intention of a preceding definition is to classify each variable as an object of one of the data types known by SCARLET, namely `INT`, `STRING`, `BOOL`, or `CELL` (and connected to this to allocate the memory size needed). Once the type is fixed, it cannot be changed during a run of the program. That is why we have the right to speak more precisely of elements of

IntVar	(for variables of type `INT`)
StringVar	(-"- `STRING`)
BoolVar	(-"- `BOOL`)
CellVar	(-"- `CELL`)

instead of *Var*.

Single Registers

If we form the projection of a cell variable on a certain register by attaching the register name (compare with pg. 49), we get an object of *IntVar* again, when the register type is `INT`,

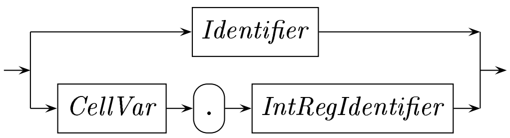
CellVar . IntRegIdentifier

and an element of *StringVar*, when it is of type `STRING`.

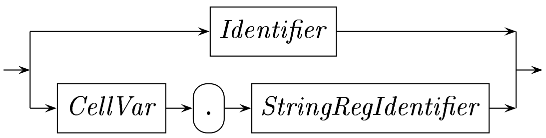
CellVar . StringRegIdentifier

To conclude this paragraph we give a little overview over all possibilities to obtain integer or string type variables:

IntVar:



StringVar:



2.7 Operators

Within the next few paragraphs we are going to see the operators used in SCARLET. Depending on how they work they are classified as relational, logical, binary, and arithmetical operators. Some of them work similarely, so they are designated by the same symbol, they will be discussed together, too.
Only some special operators, such as `[ ]` to extract single characters from string expressions or `.` to select single registers of cell expressions as well as the assign operators, are talked about later.

Binary Operators

All operators dealed with here, except the logical negation sign `!` and the signs `+` and `-`, are used with two operands in infix notation. Therefore, the syntax for this type is in general:

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

Here *Op* means the chosen operator, while $\textit{Expression}_1$ and $\textit{Expression}_2$ are elements of one certain expression class. The result of such an action will again be an expression. (Further information can be found in 2.10, pg. 53.)

Unary Operators

Unlike this, operators with only one operand (unary operators) are directly - and without blank - placed in front of the operand:

$$\textit{Op Expression}$$

The statements about the membership of operand and result are also valid in this situation.

2.7.1 Relational Operators *RelOp*

SCARLET provides all the common relational operators on integer values *IntExpression* in just the form we are used to in daily life. (Compare with pg. 53.) In detail we are talking about

Relop:

`==`	equal to
`<`	less than
`>`	greater than

`!=`	not equal to
`>=`	greater than or equal to
`<=`	less than or equal to

Integer Expressions

They compare two operands at a time and return a boolean value corresponding to the resulting logical value.

String Expressions

But relational operators may also be applied to string expressions (short: strings) (see pg. 55). In this case, the normal comparison for numbers is replaced by the lexical one, that means, two strings are compared character by character, starting at the left and obeying the following rules:

- Two strings are equal to each other, if both have the same length, and all characters are identical, otherwise they are not equal.
- A string S_1 with length n, $n \in I\!N$, is greater than a second string S_2, if the first $i-1$, $i \leq n$, characters are identical and either the ASCII number of the i-th character of S_1 is greater than the i-th character of S_2, or S_2 has only got $i-1$ characters.

If the exceptional features of the internal order of the ASCII code are not taken in account, the lexical comparison may sometimes lead to slightly surprising results:

Exp.8:

$\surd$ `"bigger"> "big"`$\leadsto$ `TRUE`
$\surd$ `"mouse"> "MAMMOTHE"`$\leadsto$ `TRUE`
$\times$ `"99"> "One"`$\leadsto$ `FALSE`

ASCII-Code

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

Cell Expressions

Because cell expressions consist of one or more registers with the same priority (see pg. 56), it does not normally make sense to ask whether one cell expression is greater (or less) than the other; but some of the relational operators can successfully be applied to them, too, namely `==` (equal) and `!=` (not equal). Here a comparison can be done by comparing all corresponding registers with each other. Hence two cell expressions are equal, if the contents of each corresponding register pair are equal; otherwise they are not equal. Doing this, we take for granted that both cell expressions have the same register structure, otherwise they would not be comparable[2].

If it is the issue, whether two register contents are equal while one of them is `VOID`, the result will always be TRUE; (in case of "unequal" it will be FALSE.) Indeed `VOID` enables partial comparisons of cell expressions. **Partial Comparisons**

Exp.9: $\surd$ `{"s", 25, VOID} == {"s", 25, "a"}` $\leadsto$ `TRUE`
$\times$ `{"s", 25, VOID} != {"s", 25, "a"}` $\leadsto$ `FALSE`

2.7.2 Logical Operators *LogOp*

In Scarlet the three fundamental logical operators AND, OR, and NOT are represented by `!`, `&&` and `||`:

	`!`	logical negation
LogOp:		
	`&&`	logical AND
	`||`	logical inclusive OR

They operate within the class of boolean expressions (see pg. 58).
`!`, contrarily to `&&` and `||`, is a unary operator [3] and, therefore, is put directly (and without blank) in front of the operand.

2.7.3 Binary Operators *ShiftOp, BitOp*

Furthermore there also exist binary operators in Scarlet, which process integer expressions (see pg. 53) thus elements of $\mathbb{Z}$. Therefore they strictly make use of the value's internal binary representation as a 32-bit number. (For further explanation it will be useful to think of these binary numbers as written down in a scheme with 32 fields, in which the one at the left end is used for the sign (`+` corresponds to 0, `-` to 1) and the remaining 31 bits to show the number itself.) We distinguish between shift and logical operators

ShiftOp:

[2]This can not happen, since the structure is uniformly fixed for all cell expressions within one program.
[3]That is the reason, why it cannot belong to the syntactical class *LogOp*.

`<<`	Shift to the left
`>>`	Shift to the right

BitOp:

`&`	bitwise AND
`|`	bitwise inclusive OR

Shift Operators

The shift operators in SCARLET work like the corresponding operators in C for signed variables.

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

means that the bit pattern of $IntExpression_1$ is $IntExpression_2$ positions shifted to the left. In fact all bits "dropping out" of the left side of the scheme are lost, while the scheme is filled up with zeros behind the shifted number. $IntExpression_2$ must always be positive and less than the width of $IntExpression_1$ regarded as a binary number.
For the shift to the right the situation is very similar:

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

$IntExpression_1$ is shifted $IntExpression_2$ positions to the right, but here the sign of $IntExpression_1$ decides which bit is used for the fill-up: 0 if the sign is positive and 1 if it is negative.

Exp.10: `11 << 4` $\leadsto 00\cdots0000001011$
$\leadsto 00\cdots0010110000 \leadsto$ `176`

`-14 >> 1` $\leadsto 1000\cdots001101$
$\leadsto 1100\cdots000110 \leadsto$ `-1.073.741.830`

Logical Operators

Looking at `&` and `|`, the binary representation is interpreted as a sequence of logical values; every 1 is taken as TRUE, every 0 as FALSE, so both expressions are bitwise combined with the logical AND or OR respectively, and the results of all single comparisons are returned as 0 or 1 again.

Exp.11: `13 & -7` $\leadsto$

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

The results of all binary operations are again integer expressions.

2.7.4 Arithmetical Operators *AddOp, MulOp*

Of course, SCARLET knows the four basic operators of arithmetic.

MulOp:

`*`	Multiplication
`%`	Modulus
`/`	Division

AddOp:

`+`	Addition
`-`	Subtraction

Their use does in no way differ from what we would expect; as usually they operate on the set of integer expressions (see pg. 53) and have integer expressions as results, too. Furthermore it should not be surprising that divisions by zero are not allowed.
Please notice that all operations happen on a subset of $\mathbb{Z}$; therefore, the normal division has to be replaced with the integer division `/`, which returns the integer part of the division, and the modulus `%`, which returns the remainder. **Modulus**

Exp.12:
```
7 % 3 ⇝ 1
7 / 3 ⇝ 2
7 % 3 + 7 / 3 ⇝ 3
```

`+` and `-` have a second meaning as unary operators: in this case they are used as signs and stand right in front of the operand. **Signs**

On top of everything, `+` works as a concatenation operator, if it is applied to two string expressions (see pg. 55). **Concatenation**

$$StringExpression_1 \texttt{ + } StringExpression_2$$

denotes the string expression that will emerge, if $StringExpression_2$ is attached to $StringExpression_1$. Certainly the maximum string length of 255 characters must not be exceeded in process.

Exp.13: `"thousand "+ "five"⇝ "thousand five"`

2.7.5 Operator Hierarchy

In this paragraph, all operators already known together with the assign operators (`=`, `+=`, `-=`, ..., `++`, `--`) as well as `[ ]` and `.`, which will be discussed later, are listed in a table.
To avoid confusions we already mention at this point that the brackets used as operators `[ ]` are intended to get a character at a certain position of a string. (Brackets are also used to mark cell positions; but there they do not function as operators.)
By the selection operator `.` single registersof cell expressions are accessed (see pg. 49).

The operators are listed in rows by descending evaluation order. Besides, for operators with the same priority it is indicated whether they are evaluated from the left to the right (L) or the other way around (R).

Concerning evaluation order and direction Scarlet totally agrees with the programming language C.

Operators	Direction
`( )`, `[ ]`, `.`	L
`++`, `--`, `!`, `+` (*Sign*), `-` (*Sign*)	R
`*`, `/`, `%`	L
`+`, `-`	L
`<<`, `>>`	L
`<`, `<=`, `>`, `>=`	L
`==`, `!=`	L
`&`	L
`|`	L
`&&`	L
`||`	L
`=`, `+=`, `-=`, `*=`, `%=`, `/=`, `<<=`, `>>=`, `&=`, `|=`	R

Exp.14: Sometimes it becomes necessary to use parentheses to obtain evaluation priority:

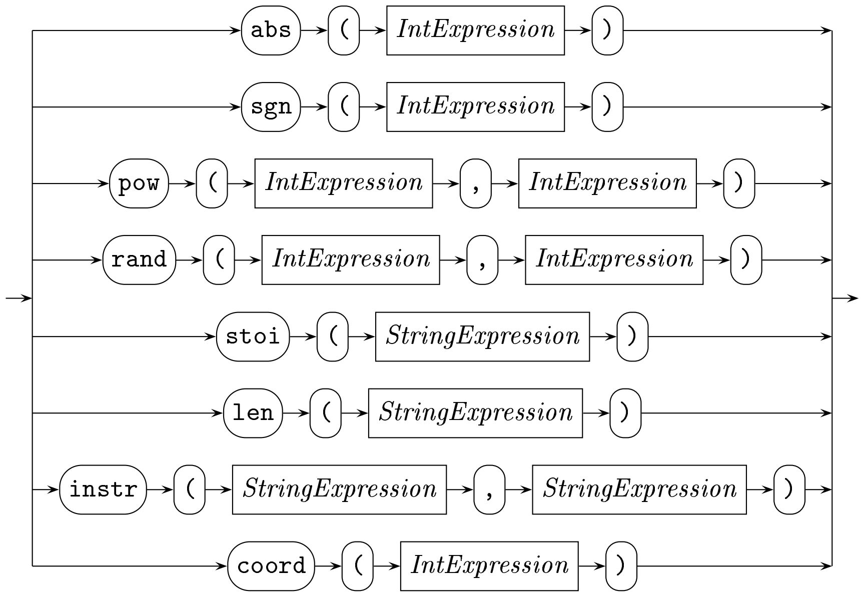
√ `!TRUE || TRUE` ⇝ `TRUE`
× `!(TRUE || TRUE)` ⇝ `FALSE`
√ `1000 < 1024 && !("today"=="tomorrow")` ⇝ `TRUE`
√ `16 >> 1 == 2 << 2` ⇝ `TRUE`

2.8 Functions

Besides the basical operators, SCARLET offers some built-in functions, which may simplify the work with strings (pg. 55) and integer expressions (pg. 53). To classify them we make a distinction between functions that have an integer result and such that return a string.

2.8.1 Integer Functions *IntFunctionCall*

Syntax:



`abs(` *IntExpression* `)`

calculates the absolute value of *IntExpression*.

`sgn(` *IntExpression* `)`

returns in principle the sign of its argument. Since the result should be an integer, it is `1` if *IntExpression* is positive, `-1` if it is negative, and `0` if *IntExpression* is itself zero.

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

calculates the power of $IntExpression_1$ with exponent $IntExpression_2$. In order not to exceed $\mathbb{Z}$, negative values for the exponent are not permitted.

Remember furthermore that the result in absolute value must be less than $2^{31} - 1$.

Exp.15:

```
pow(10,5) ⤳ 100 000
pow(34,0) ⤳ 1
pow(-2,3) ⤳ -8
```

`rand(` i_1 `,` i_2 `)`

where i_1 and i_2 are elements of *IntExpression*, determines a random number of $\mathbb{Z}$ within the interval $[i_1, i_2]$ if $i_1 \leq i_2$, and within $[i_2, i_1]$ if $i_2 < i_1$.

Exp.16:

`rand(1,49)`	random number between 1 and 49.
`rand(49,1)`	-"-
`rand(9,9)`	returns always 9.

Exp.17: The random function allows the construction of **probabilistic CA** using the following method:

First, for each site a random number between 1 and $n \in \mathbb{N}$ is determined and stored in a variable `number`. Then for only one single event <*Neighbour configuration*> a special rule table can - just depending on `number` - provoke different actions <*Consequence 1*>, <*Consequence 2*>, and so on, with probabilities $\frac{n_1}{n}$, $\frac{n_2-n_1}{n}$, ..., $\frac{n-n_m}{n}$, $0 < n_1 < \ldots < n_m \leq n$.

If the generic cell is itself a member of the neighbourhood as well, then $n_m < n$ is legal; in this case it will keep its old state with probability $\frac{n-n_m}{n}$. Like this, it is automatically guaranteed that the sum of all probabilities is 1.

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

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

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

For further information concerning the structure of SDL programs please refer to 4.2.2 and 4.2.3.

`stoi(` *StringExpression* `)`

converts *StringExpression* as far as possible into an integer, starting at position 0 and obeying the rules below:

- The first character that does not match the integer type ends the conversion; all characters behind will be ignored. In particular, `stoi()` returns `0`, if the conversion is abandoned before it has found a digit.
- Leading blanks - but <u>only those</u> - are ignored.
- The function is not defined for such strings whose conversion would lead to a result that is bigger than $2^{32} - 1$.

Exp.18:

```
stoi("     -431332"); ⤳ -431332
stoi("120.982.566"); ⤳ 120
stoi("state 42"); ⤳ 0
```

`len(` *StringExpression* `)`

determines how many characters *StringExpression* consists of.

Exp.19: Suppose `thirteen` to be a variable of type `STRING` with contents `"13"`.

```
len("ten letters") ⤳ 11
len(thirteen) ⤳ 2
```

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

passes through *StringExpression*$_1$ from the left to the right in order to find out if *StringExpression*$_2$ is a substring. Whether this is true, `instr()` returns the position of the first corresponding character, otherwise `-1`. (We start counting with 0.) If *StringExpression*$_2$ is found more than once in *StringExpression*$_1$, the first agreement will be taken.

`coord(` *IntExpression* `)`

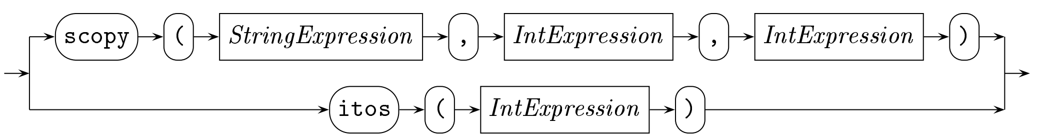
may only be used in RDL within the structure *CellAssignment* in places where integer expressions are allowed. Suppose that d is the space dimension, then *IntExpression* must satisfy $1 \leq IntExpression \leq d$.
If *CellDomain* only consists of one site *Cell*, `coord(`i`)` returns the i-th component of the coordinate of *Cell*.
If *CellDomain* denotes a set of cells *Loop*, `coord(`i`)` will internally be applied to all single cells of the set. (This means, each cell will be assigned the i-th component of its own coordinate.)
More about *CellAssignment*, *Cell*, and *Loop* can be found in chapter 3 about RDL, starting at page 72.

Exp.20: In $\mathbb{Z}^2$ we are going to look at sites with two integer registers `x1` and `x2`. We choose a retina such that all cells of the square with diametral corners $(0,0)$ and $(10,10)$ are elements of it. By the following devices placed within the statement part of the belonging RDL program, we achieve that the registers `x1` and `x2` of those cells are initialized with their coordinates:

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

2.8.2 String Functions *StringFunctionCall*

Syntax:



`scopy(` *StringExpression* `,` $\mathit{IntExpression}_1$ `,` $\mathit{IntExpression}_2$ `)`

returns a substring of *StringExpression* starting at the position given by $\mathit{IntExpression}_1$. (Note that we find the first character in a string at position 0.)
$\mathit{IntExpression}_2 \geq 0$ represents the length of the substring. Ensure that it is still completely part of the string; more precise,

$$\mathit{IntExpression}_1 + \mathit{IntExpression}_2 \leq \texttt{len(}\ \mathit{StringExpression}\ \texttt{)}$$

must be satisfied.

Exp.21: `scopy("state 13",6,2)` $\leadsto$ `13`
`scopy("1234567890",3,1)` $\leadsto$ `4`

`itos(` *IntExpression* `)`

converts *IntExpression* into a string constant. Leading zeros in *IntExpression* are not allowed.

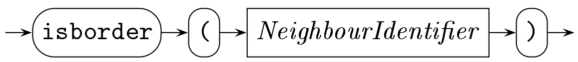
Exp.22: `itos(-10203)` $\leadsto$ `"-10203"`

Suppose that `str` is a string variable with contents `"3 happy corners of a triangle"`.

`itos(stoi(str)-1)+"corners?!"`$\leadsto$ `''2 corners?!"`

2.8.3 Boolean Functions *BoolFunctionCall*

Syntax:



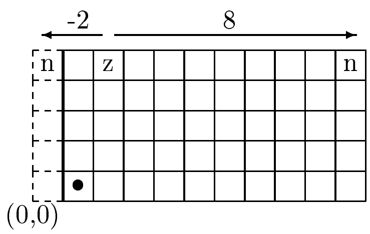
`isborder(` *NeighbourIdentifier* `)`

This function is only available within the statement part of SDL programs. There it finds out whether the neighbour of the current site labeled by *NeighbourIdentifier* is within the range of the retina. If not, it is an element of the border and the function returns TRUE; otherwise `isborder()` is FALSE. Compare with page 67 and page 92.

Exp.23: Having the `isborder()` function it is easy to construct lattices with **inhomogeneous boundary conditions**, for example tori. All we have to do is to enlarge the neighbourhood and to manipulate the local rule a bit: Suppose that the retina will be stuck together in the i-th coordinate direction, and suppose that i_j is the retina's extension in this direction. Then for each neighbour with non-zero i-th component a new one must be added to the neighbourhood, where the i-th component is exchangend with the smallest equivalent element with inverse sign of the cyclical group $(Z_{i_j}, +)$; or to express it in unformal terms: each neighbour which will be "on the other side of the gap" for some sites must be described twice, "clockwise" and "counter-clockwise".
Besides, the critical passages of the local rule table are supplied with entries that check whether the neighbour lies on the border and then activate the substitute.

We are now going to have a look at a **torus** that developed from a two-dimensional retina `[0,0]..[9,4]` by sticking the x_1-coordinates together, and a rule which shifts the whole pattern "clockwise" in x_1 direction, two sites further each time step:

```
SIGMA torus;
DIMENSION 2;     /* 2-dimensional retina */
REGISTER INT state;     /* integer register state */
NEIGHBOUR l_left :  (-2,0);     /* neighbour */
          l_left_behind_border :  (8,0); /* substitute */

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

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



Two possibilities to describe the neighbour n of z on the torus.

2.9 Accessing Single Registers

As we have already seen before, the complete "state variable" of cell objects is subdivided into smaller units: the registers, which can be either of type `INT` or of type `STRING`; in other words: the `CELL` type of SCARLET is similar to the compounded data types `struct` of C or `RECORD` of Pascal. Hence it would be, of course, very nice to have a possibility to access the single components in order to get (or to store) their contents as strings or integers. And - how could it be otherwise - SCARLET comes up to our expectations: defining a state structure in the declaration part of a RDL or SDL program (see pg. 68) each register gets a name, which can be used to access this component of any cell object whatever, from then on. To do this, like in C or Pascal, the names of the object and the register are simply linked together separated by the selection operator `.` :

<objectname> . <registername>

The only exception to the rule is the *SelfAssignment* environment of SDL where it is sufficient to type the name of the register if the "generic cell"[4] is meant.

The new object has got the same type as the register involved thus `INT` or `STRING`.

Note that the syntax will not change, if the cell structure only consists of one register. Even in this case the contents has got type `CELL`- not type `INT` or `STRING`, as one might possibly expect. Also here the register must be accessed to get the value in integer or string format.

Because there are some remarks to be made, we are going to distinguish between four different kinds of access:

a) Accessing Symbolic Constants and Variables

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

[4]The "generic cell" syntactically stands in for each cell of the retina the local rule is applied to.

Objects like this, which have their origin in variables or symbolic constants of type `CELL`, could be treated like normal variables or symbolic constants of type `INT` or `STRING`. They exist in RDL as well as in SDL. Of course, it is strictly forbidden to declare them separately as isolated integer or string objects, they can only be defined as cell objects.

Exp.24:

```
     ...
REGISTER INT num;
         STRING str;
VAR CELL a_variable;
CONST a_constant={5,"nothing special"};

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

b) Accessing Sites by their Position

by *CellDomain.IntRegIdentifier*
CellDomain.StringRegIdentifier

is only possible in RDL in order to design an initial configuration. Here, *CellDomain* represents one (or more) existing coordinate(s) within the range of the retina, while *IntRegIdentifier* and *StringRegIdentifier* represent the labels of an integer or string register. (Compare with 3.2.4, pg. 76.)
Since it is not necessary to read the initial configuration just created, we only have a writing access in RDL.

Exp.25: This example of a retina (comp. to [6]) should represent the snapshot of a `0`-sky with `1`-snow flurry. Therefore, the random function is used to fill all registers of the two-dimensional retina with corners $(0,0)$ and $(5,10)$ with `1` or `0`. The site with coordinate $(2,4)$ should contain a snowflake, at any rate.

```
RETINA falling_snow;
RANGE [0,0]..[5,10];     /* rectangular retina */
REGISTER INT state;     /* integer register state */
```

```
START
  [0,0]..[5,10].state = rand(0,1);     /* rand. 0/1 pattern */
  [2,4].state = 1;      /* a fixed snowflake */
```

c) Accessing Neighbours

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

is only possible in SDL. By *NeighbourIdentifier* - as an element of *Identifier* - a name for an existing so-called neighbour is given, while *IntRegIdentifier* and *StringRegIdentifier* represent the names of integer and string registers. (They are determined by the neighbourhood; see 4.1.4, pg. 81.) Because neighbours do not play a part in constructing the initial configuration, they are not defined in RDL.
Registers of such objects can only be accessed reading. (Usually the states of the neighbours have an influence on the state of the "generic cell", but they themselves must not be changed at that moment.)

Exp.26: (Resumption of example 25, compare to [6])
Now we present a local rule table, which fits the retina of the last example; it has got two entries and is intended to simulate the falling down 1-snow and its piling up on the ground. These are the rules put in words:

1. If the generic cell is still free, and there is a snowflake above, it will have a snowflake in the next step. (The second rule will take care that the flake above will be deleted, at the same time.)
2. If the generic cell has already got a snowflake, and there is still space in the cell beneath, it will drop it. (The first rule ensures that it is caught by the cell below.)

The border cells whose state is {35} form a boundary to this rule.

```
SIGMA falling_snow;
DIMENSION 2;
REGISTER INT state;
NEIGHBOUR upper:  (0,1);     /* neighbour above */
          lower:  (0,-1);     /* neighbour beneath */
          center: (0,0);     /* generic cell */
```

```
TABLE
  /* 1.  entry:  */
  upper.state == 1;     /* neighbour configuration */
  center.state == 0; :       /* -"- */
  state = 1; :      /* assignment to generic cell */

  /* 2.  entry:  */
  lower.state == 0;     /* analogous */
  center.state == 1; :
  state = 0; :
```

d) Accessing the Generic Cell

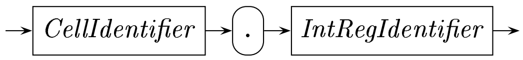
by *IntRegIdentifier*
StringRegIdentifier

- where by *IntRegIdentifier*and *StringRegIdentifier* valid names for integer and string registers are meant - is done without using the selection operator `.` , because from the context it follows that they are registers of the generic cell `SELF`, anyway. This kind of (writing) access is only used in SDL in order to update the generic cell (comp. to exp. 26).

Remarks on the Syntax

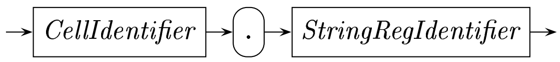
Because of the classification of language elements according to the functions they have, it became rather difficult to get a general idea of the syntax of the different language classes introduced here. That is why we give at least the whole syntax of the most important structures *IntReg* and *StringReg* in the end. There must be made a distinction between RDL and SDL:

IntReg (RDL):



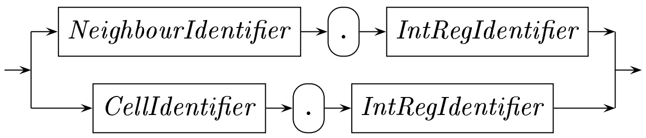
SDL↔RDL

StringReg (RDL):

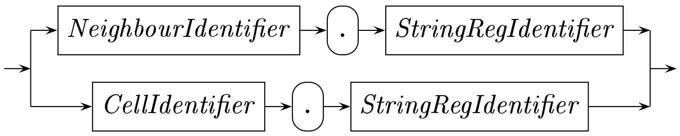


SDL extends this syntax to neighbours:

IntReg (SDL):



StringReg (SDL):

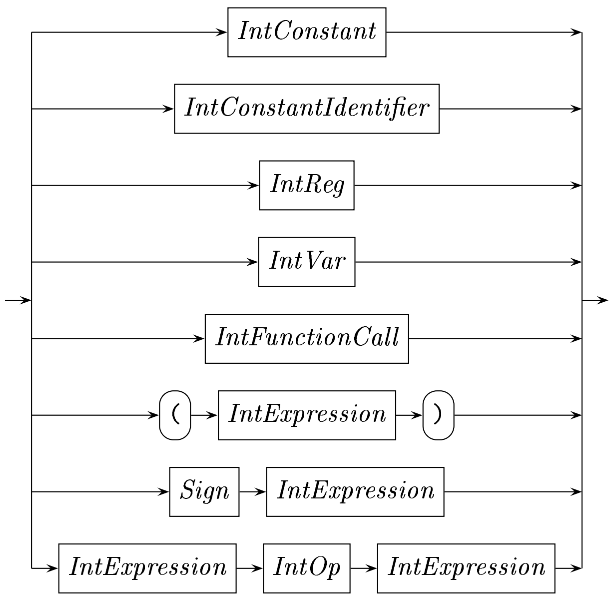


2.10 Expressions

2.10.1 Integer Expressions

IntExpression

Syntax:



In principle, every object of type `INT` or better every arithmetical expression in parentheses is called an integer expression. Now you will surely have a rather good idea of what could be meant, and, of course, this is totally legitimate! But we should not forget that we try to communicate with a compiler, which does really not need to understand all the subtile human thoughts and intentions... That is why we - in spite of our intuition - are going to precise the definition:

Elementary Expressions

First we already know some elementary integer expressions, which are described above in the same chapter, and which all the other integer expressions are assembled out of. These modules are

- integer constants (*IntConstant*)
- symbolic integer constants (*IntConstantIdentifier*)
- integer variables (*IntVar*)
- return values of integer functions (*IntFunctionCall*)
- integer registers (*IntReg*)

Take specially care about restrictions and differences between *IntReg* objects in RDL and SDL (see page 52).

Compounded Expressions

Besides that, we have the possibility to come to more complex expressions using binary operators with operands from *IntExpression* (*IntOp*) introduced in 2.7 (`*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `|`)

$$IntExpression_1 \; IntOp \; IntExpression_2$$

Operator Hierarchy

Like this, it is no problem to design expressions with more than one operator. In this case, the order of their evaluation is given by a fixed hierarchy. Operators with higher priority are evaluated first. (E.g., `*`, `/` and `%` have an higher evaluation priority than `+` and `-` according to the well-known arithmetical rule.) A complete listing for all operators can be found on page 42. Whenever there are two operators which have the same priority within the same expression, the left one is evaluated first. (The only exception to the rule are the signs, which have not been mentioned yet.)

Parentheses

Furthermore an expression put in parentheses is again an expression:

`(` *IntExpression* `)`

As usual the parentheses give the evaluation of the surrounded expression priority treatment; - of course, only in relation to the normal order, or in other words, the expression which is surrounded by the greatest number of parentheses is not automatically evaluated first. The example beneath should make it a bit clearer:

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

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

In particular the existing hierarchy may be canceled, that means, operators with a lower priority can be evaluated before others with an higher priority.

Signs

In principle it is not forbidden by SCARLET'S grammar to transform <u>every</u> expression by placing a sign `+` or `-` directly in front of it into a new one with the same or negative value respectively; but we will not allow such expressions where more than one `+` or `-` are following each other. It could be avoided using parentheses.

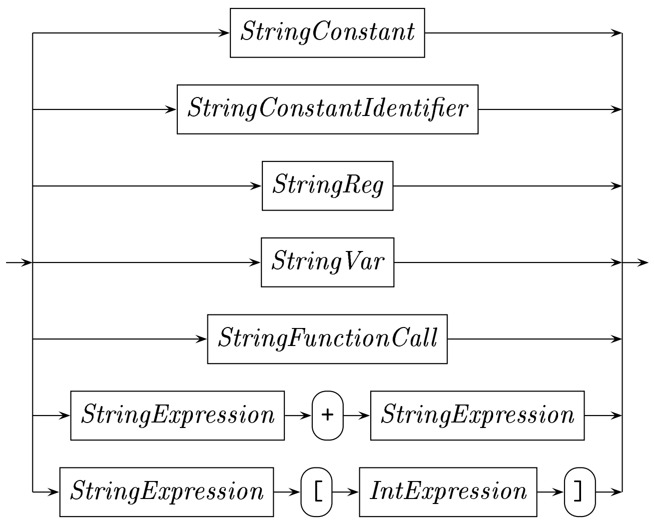
Exp.28: Finally there are some examples of integer expressions.
Let us suppose that `int_var` is an integer variable with contents `-15`.

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

2.10.2 String Expressions

StringExpression

Syntax:



Compared with integer expressions the structure of string expressions (strings) is easier. The reason for this is very simple: there exists only one operator that converts two strings into a new string, namely the concatenation operator `+` , which simply attaches one string to another.

The basic modules of *StringExpression* are

- string constants (*StringConstant*)
- symbolic string constants (*StringConstantIdentifier*)
- string variables (*StringVar*)
- string registers (*StringReg*)
- return values of string functions (*StringFunctionCall*)

All these objects have already been discussed in this chapter. Note that - like *IntReg* - not all elements of *StringReg* are available in RDL (see page 52).

Substrings

Besides the general `scopy` function there exists a second, special way to get to substrings:

StringExpression `[` i `]`

returns the i-th character of *StringExpression*, where i is of *IntExpression*, $0 \leq i \leq$ `len`(*StringExpression*) $- 1$. (As usual we start counting from 0.)

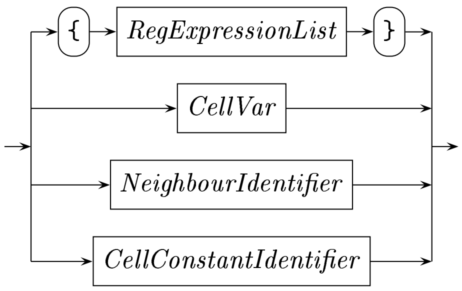
Exp.29: Suppose `language` to be a variable of type `STRING` with contents `"Retina Definition Language"`.

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

2.10.3 Cell Expressions

CellExpression

CellExpression (SDL):



Even less complicated than the definition of strings is the one of the so-called cell expressions. There does not exist a way to make up more complex expressions of simple ones at all! In consequence, all cell expressions we could ever think of are captured by the list below:

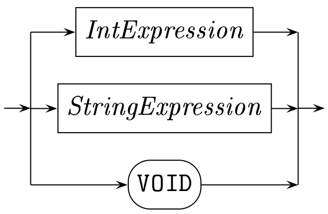
- cell constants (*CellConstant*)
- symbolic cell constants (*CellConstantIdentifier*)
- cell variables (*CellVar*)
- constructions of the form { *RegExpressionList* }
- neighbours (*NeighbourIdentifier*)

We will have a closer look to the kind of cell expressions given by

{ *RegExpressionList* }

Here the contents of all registers *RegExpression* are indicated in an explicitely way. There must be as many *RegExpression* as registers declared before. Depending on the precise, predetermined state structure (number, order, and kind of registers) they can be integer expressions or string expressions respectively or `VOID`:

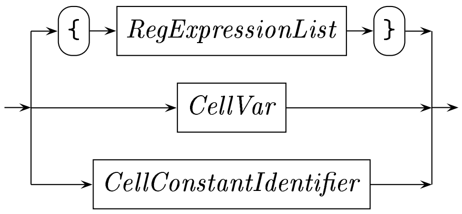
RegExpression:



For further information please refer to page 33 as well as page 68.

Note that the use of neighbours is only allowed in SDL, but not in RDL. For this reason *CellExpression* is used in RDL with a slightly different definition, **SDL↔RDL**
without *NeighbourIdentifier*:

CellExpression (RDL):

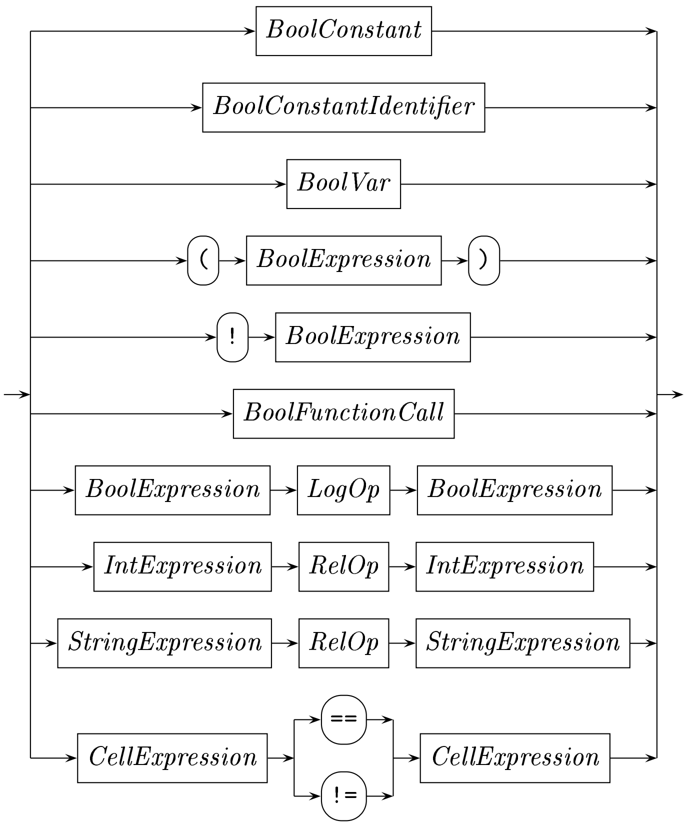


StrictCell-Expression

From time to time it makes sense to forbid neighbours as cell expressions in SDL as well; therefore, we consider there the class *StrictCellExpression* with exactly the same meaning as *CellExpression* in RDL.

2.10.4 Boolean Expressions *BoolExpression*

Syntax:



Elementary Expressions

Analogous to the other expression classes, roughly speaking, all objects that represent boolean values belong to the class of boolean expressions. To be at bit more sophisticated, we distinguish again between elementary and compounded expressions. To the first group belong

- boolean constants (*BoolConstant*)
- symbolic boolean constants (*BoolConstantIdentifier*)
- boolean variables (*BoolVar*)

We have already had a look at them.
Furthermore elementary boolean expressions are also obtained as results of comparisons between expressions of other types. Although they have been introduced earlier (see 2.7), their full syntax is shown here again:

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

In case of strings and integers, *RelOp* may be any relational operator `==`, `!=`, `<`, `<=`, `>`, or `>=` whatever.
Since it makes no sense to apply `<` or `>` to cell expressions, the only remaining cases here are

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

Compounded Expressions

Going over to compounded boolean expressions, we observe that a parenthezised boolean expression is again a boolean expression:

$$(\ BoolExpression\ )$$

Like in the context of integer expressions, the parentheses do the same here. Their high evaluation prioritiy causes the expression within to be evaluated first.

But what should parentheses be useful for without a chance to gain more complex boolean expressions connecting easy ones? Therefore, we have the logical operators `&&` (AND), `||` (OR), and `!` (NOT) at our disposal:

$$BoolExpression_1 \texttt{ \&\& } BoolExpression_2$$
$$BoolExpression_1 \texttt{ || } BoolExpression_2$$

$$!\,BoolExpression$$

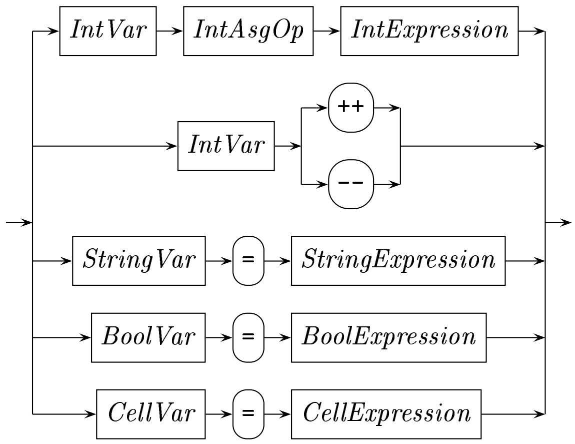
You find a description in 2.7.

Remark:
If (boolean) results of comparisons are connected by `&&` or `||`, parentheses are

not needed to make clear which are the boolean expressions. The evaluation priority of relational operators is higher than the one of AND and OR.
The situation changes as soon as the logical negation `!` is involved: its priority is higher than the members of both *RelOp* and *LogOp*. Especially if the result of a comparison should be denied, the comparison <u>must</u> be collected in parentheses.
Observe furthermore that the priority of `&&` is higher than of `||`. (In case of remaining doubts you may consult the operator table on page 42.)
Some examples of boolean expressions can be found in example 14 on page 42.

2.11 Assignments to Variables *VarAssignment* ;

Syntax:



2.11.1 General Form

Talking of an assignment to a variable we mean the storage of a value in that part of the memory which is allocated to this variable. The value being there before is overwritten by this action.
The most general syntax of an assignment is given by [5]

$$Var = Expression \ ;$$

It means that *Expression* is assigned to a variable with name *Var*. Both the variable and the value may be of type `INT`, `STRING`, `BOOL`, or `CELL`. The only

[5] The semicolon is not a part of *VarAssignment*, but belongs to superordinated parts of the grammar, which are not mentioned explicitely.

thing that is important is that their types must be identical. (Remember that the variable type is already fixed at the beginning of a program. Because of that fact *Var* is replaced by *IntVar*, *StringVar*, ... in the syntax diagram above.)

Exp.30: The following example, which is - admittedly - not very closely connected with CA, should illustrate assignments to the different variable types. The lines shown here could (at least theoretically) be part of a RDL as well as of a SDL program:

```
     ...
VAR INT sum;
    CELL fishes, others, all;
    BOOL ok;
REGISTER STRING species;     /* string register species */
         INT amount;     /* num.  register amount */
     ...
fishes = {"carps", 23};
others = {"water-fleas", 1842103};
sum = fishes.amount;
sum = sum + others.amount;
all = {"inhabitants", sum};
ok = all.amount == 1842126;
     ...
```

2.11.2 Short Forms

Observe that the syntax introduced above and especially the structure of *Expression* allows the variable name to appear on both sides of the equal sign; that means that the new value of *Var* is just a conversion of the old one.
Especially in connection with integer variables, assignments of this kind are often used. In the syntax below *IntVar* denotes twice the same variable.

IntVar = *IntVar IntOp IntExpression* ;

Like C, SCARLET offers abbreviations instead of these long forms. Generally they are given by

IntVar IntOp= *IntExpression* ;

Note that there must not be a blank between *IntOp* (thus `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `|`) and `=`. This class of compounded assign operators (`*=`, `/=`, `%=`, ...) we call *IntAsgOp*. ***IntAsgOp***

Exp.30: (Resumption)
We could have replaced `sum = sum + others.amount;` with

```
sum += others.amount;
```

The situation after a fairly shared meal (for the carps, of course...):

```
all.amount -= (others.amount / 23) * 23;
others.amount %= 23;
```

Increment, Decrement

Besides that, SCARLET provides own operators for increment (`++`) and decrement (`--`) of integer variables:

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

In the first case, `1` is added to the value of *IntVar*; in the second `1` is subtracted from it.

Remark:
! Concerning increment and decrement SCARLET is stricter than C: although *IntVar*`++` is a short form, it stays an assignment and can <u>never</u> become part of an *Expression*!

Exp.30: (Resumption)
It seems that one of the carps has eaten too much...:

```
fishes.amount--;
```

2.11.3 The Special Status of `VOID`

The compounded structure of cell objects leaves us an odd little programming element: the word `VOID` (not to be confused with the string `"VOID"`), which can be used as contents of registers of <u>all</u> types, but <u>only there</u>.
But, although we do not restrict its appearance to special objects of cell type, it will only be discovered in constants and symbolic constants; it will not show up in variables or even cells within the retina or on its border. The

explanation for this behaviour is very simple:
whenever two cell expressions are put in some relation to each other by an operator, all couples of corresponding registers where `VOID` can be found as a contents of at least one of them, are excluded from the action. (Note that `=` is an operator, too.) More concretely:

- In case `VOID` is on the right side of an assignment, the value of the corresponding target register remains invariant.

- When it is checked, whether two cell expressions are equal, all registers where `VOID` is involved yield `TRUE`; and `FALSE` in case it is asked if they are not equal.[6]

Now that we know this, it is evident why `VOID` can never appear in registers of sites or cell variables: they always get their contents by an assignment!
But what about the symbolic constants? Are there not values assigned to them, too? No. In this case, the equal sign does not function as an assign operator; it just gives the value a name! And because `VOID` is a constant expression, it will not be affected by the evaluation of the previous expression of the right side. You find more about symbolic constants and how they work in 3.1.4, pg. 70.

[6]If all registers are compared with `VOID`, the general description given above fails here.

Exp.31:

```
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)
```

Note that `VOID` is neither of type `INT` nor of type `STRING`. So when the contents of a register should be interpreted as a string or an integer for further progression, it must be guaranteed first that it is not `VOID`. (But actually this is easy to control, for `VOID` can only be found written down explicitely within the source code or in symbolic constants. And, because it makes no sense to extract single values out of values which are explicitely given, we just have to make sure that there is no illegal access to symbolic constants.)

!

Exp.32: To continue the example above, the following assignments would have been forbidden:

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

$\texttt{intvar = abs(}\underbrace{\texttt{cellconst\_1.r1}}_{=\texttt{VOID}}\texttt{) + 42;}$

Chapter 3

RDL - Retina Definition Language

After having a look at all the basic modules we need to create a RDL or SDL program, we will now try to understand their specific *programming structures*. Therefore, we certainly have to make a distinction between RDL and SDL. In this chapter we will start with RDL programs.

As already mentioned it is the aim of a RDL program to determine the retina of the CA, or in other words, to fix its spatial extension in $\mathbb{Z}^n$ for a certain $n \in \mathbb{N}$, as well as to gain a valid initial configuration. But before doing this, it is also necessary to make some statements about the state set of the CA, in terms of SCARLET to determine the amount, the sequence and the types of the registers used.

No Neighbours

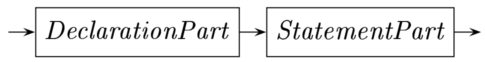
Notice that the neighbourhood is of no importance for the retina and the initial configuration. That is the reason why it simply does not exist in RDL. So never use neighbours or components of them (registers) to form an expression in RDL, more precisely:

- *CellExpression* does not allow the use of *NeighbourIdentifier* (compare with page 56),
- It is forbidden to take *NeighbourIdentifier*.*IntRegIdentifier* as an *IntExpression* as well as
- to use *NeighbourIdentifier*.*StringRegIdentifier* as a *StringExpression* (compare with page 52).

From a formal point of view, a RDL program is divided up into a declaration part and a statement part:

Retina

Retina:

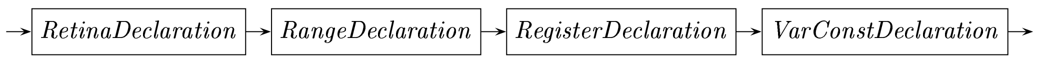


3.1 Declaration Part

DeclarationPart

The declaration part begins with a mark that tells SCARLET that a RDL program starts. Then it contains information about the range of the retina, used registers,variables and symbolic constants. It is essential to put the different parts inright the order which is given by the syntax diagram:

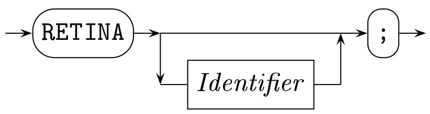
Syntax:



3.1.1 Headline

RetinaDeclaration

Syntax:



The specific keyword which makes SCARLET interpret all the following things as belonging to a RDL program is

`RETINA ;`

If you do not want to give your program a certain name, which might remind you later of some important properties of the retina, then you just finish the line - as shown above - with `;` . Otherwise you insert your favorite name as an element of *Identifier*. By the way, SCARLET will display this name, if this program is activated for a simulation, later.

`RETINA` *Identifier*;

Arriving at this point, an important common rule for both RDL and SDL becomes already visible:

Each device or declaration unit ends with a semicolon !

On the following pages the different pieces of the declaration part will be described in the same order they have to appear within the program.

3.1.2 Range of the Retina *RangeDeclaration*

Beneath the headline, the actual declarations of a RDL program start with the shape of the retina itself. As already mentioned, we think of it as a finite, connected set in $\mathbb{Z}^n$. Because they are much easier to handle, and it grants the set to be connected, we consider in SCARLET only such retinas which form a cube. In each point (coordinate) of the set we imagine a cell of the CA to be located. According to this, we can always refer to a cell by its coordinate:

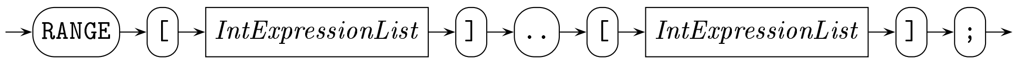
$$[\ \mathit{IntExpression}_1\ ,\ \mathit{IntExpression}_2\ ,\ \ldots\ ,\ \mathit{IntExpression}_n\ ]$$

or shorter

$$[\ \mathit{IntExpressionList}\ ]$$

In fact, this notation is just the usual way to write a n-dimensional vector. Starting from this, we get the syntax of the whole cube easily by indicating two coordinates, which will be interpreted by SCARLET as two diametral corners of the cube.

Syntax:



In more formal terms, if we compare the two coordinates in a certain component, we get the cube's extension in this direction of the coordinate system. The corners themselves belong to the retina. Because of this the retina's extension cannot be zero in no direction, therefore, the set cannot be empty, it always consists of at least one cell. It need not be underlined that $\mathit{IntExpressionList}_1$ and $\mathit{IntExpressionList}_2$ must have the same length. In fact, the dimension of the retina is given by the length, at the same time. **Dimension**

Because this declaration is located in the very beginning of the program, where variables and symbolic constants are still unknown to SCARLET, we **!**

cannot use them here.

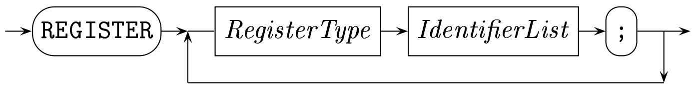
Exp.33: `RANGE [20]..[0];` defines a one-dimensional retina consisting of all the cells with coordinates between 0 and 20. - The first corner is not meant to have always the smaller values for its coordinate.

`RANGE [-3,2,5]..[3,4,5];` defines a retina within the three-dimensional space. Because of the `5` in the third place of both of the coordinates, the retina is only "one cell thick"; actually it has got the properties of a two-dimensional retina.

3.1.3 Registers *RegisterDeclaration*

In the second place, we have to introduce SCARLET to the registers which it should the CA equip with.

Syntax:



This part starts with the keyword

`REGISTER`

A single register is completely determined by giving its name *Identifier* and type *RegisterType*, which can be `INT` for an integer one and `STRING` for a string register:

$$\textit{RegisterType Identifier} \;;$$

At least one register has to be declared. In case of having more than one, the same must be done for each register. We automatically get a list like this:

$$\begin{gathered}\textit{RegisterType Identifier}_1 \;; \\ \textit{RegisterType Identifier}_2 \;; \\ \dots \\ \textit{RegisterType Identifier}_N \;;\end{gathered}$$

Order

At the same time, we have induced an order on the registers. First that might seem to be of no interest, because size and properties of the state set do not change if we choose another order. In fact, it is only a convention, which allows us not to repeat all the register names, each time we want to give the contents of a cell: it is sufficient to put the contents of the single registers in the given order, starting at the left and separated by commas.

Concluding this paragraph, we point out a possibility to simplify the style of the syntax. Instead of

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

for registers which are to be of the same type, we may type ***Register-List***

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

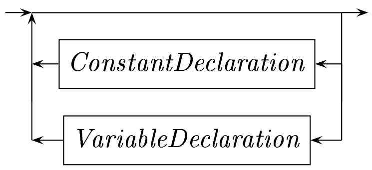
Exp.34: The constant {`1, "1. string", 2 ,3`} matches the following register declaration:

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

3.1.4 Symbolic Constants and Variables *VarConstDeclaration*

All the rest of the declaration part is provided for variables and symbolic constants.

Syntax:



You may skip this part in your program, if you have no need of these structures. Furthermore you are rather free in how to order the single devices. But take care that all the objects used to declare a symbolic constant are already defined.

Variables

VariableDeclaration

Once we have decided to use variables in our program, we have to declare them first:

Syntax:

Each variable declaration unit starts with the keyword

`VAR`

and is followed by entries of the form

VariableType IdentifierList ;

Legal variable types are `INT`, `STRING`, `BOOL`, and `CELL`, while *IdentifierList* contains the names of variables, which should be of the chosen type. There can be as many entries as we want.

Initialization

All defined variables are initialized by SCARLET. Integer variables get the value `0`, boolean the logical value `TRUE` and string variables the empty string `""`. Variables of type `CELL` are initialized by handling the single registers as string or integer variables respectively.

Exp.35:

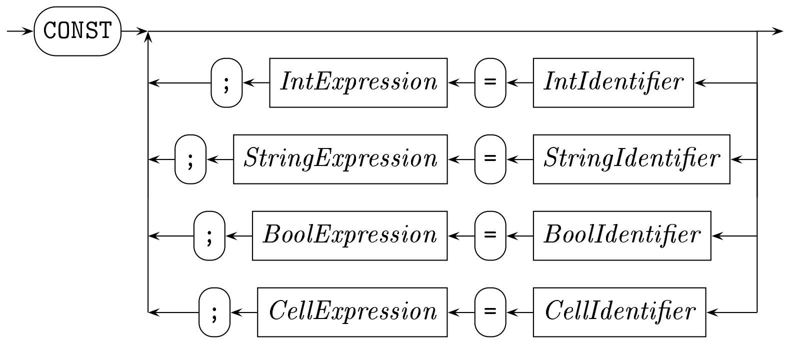
```
        ...
VAR INT counter;
    STRING name1, name2;
    INT result;
VAR BOOL is_enough;
    CELL a_state;
        ...
```

Because of the construction of *VarConstDeclaration* as shown in the syntax diagram, it is of no importance, how often we put `VAR`.

Symbolic Constants

ConstantDeclaration

Syntax:



Now we arrive at the principle of symbolic constants, already introduced in chapter 2. Like the declaration part for variables, the one for symbolic constants begins with a special keyword

`CONST`

It is followed by an arbitrary amount of entries of the form

Identifier `=` *Expression* ;

where *Identifier* means the name of the constant to be declared and *Expression* the value it should have. (Remember that *IntIdentifier*, *StringIdentifier*, ... are just *Identifiers*.) Depending on what we choose for *Expression* (*IntExpression*, *StringExpression*, *BoolExpression*, or *CellExpression*), the constant itself will be of one of the types `INT`, `STRING`, `BOOL`, or `CELL`.

It is allowed to use variables as well as other symbolic constants to form *Expression*, but only if they are already declared above.

Evaluation

Running the program later, SCARLET will first - according to their order within the source code - evaluate all the expressions on the right side of each equal sign to get constant expressions, and will finally store them internally. SCARLET does this only once a runtime. Such an evaluation leaves constant expressions unchanged, symbolic constants are replaced by their (already established) value, variables are replaced by their present value (which is nothing else than their initialization value). Expressions with function calls and operators are reduced according to the common rules explained in chapter 2, and *CellExpressions* are finally handled by evaluating the expressions in the single registers starting at the left.

It is important to notice that `VOID` is treated like a constant expression. That is the reason why it can still be a register contents of a symbolic constant after having passed the evaluation (compare to exp.31). **VOID**

Such a constant expression - as a result of this evaluation procedure - is now available under the name *Identifier*, that means, whenever there appears the name of this constant within the following source code, it will be replaced by its (constant) value.
Its declaration causes a symbolic constant to be an object of the same type as the value it got.

Exp.36:

```
      ...
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 Statement Part

Before looking at the statement part, let us have a quick review on what we have already achieved by the declaration part. We are now able to localize the CA in an Euclidean space of arbitrary dimension, more concretely, we can - in a unique way - identify every cell of its retina with a coordinate of the space. Furthermore we have already decided in favour of a general structure of the state set. But we have not mentioned yet, in which state each cell should be before running the simulation. This is the main purpose of the statement part of a RDL program!

3.2.1 Initialization of the Retina

Just as variables (and symbolic constants) SCARLET initializes the cells of the retina, too: integer registers get the value `0` and string registers are filled with the empty string `""`. To say it in other words, we need not worry about possibly undefined states, for SCARLET automatically puts the retina together with an initial configuration to our disposal. But it would, of course, be very boring to watch its evolution in time because of its great homogeneity! What else could, therefore, be our task than changing the contents of - at least a few - cells?!

But before learning something about how to assign new values, we are going to see in how far it is possible to deal with sets of cells, and how to characterize both cells and sets of cells in a syntactically right way:

3.2.2 Cells

Cell

As already mentioned, we imagine the cells of a CA to be placed in a finite area of the $\mathbb{Z}^n$, we call retina. It has been determined in the declaration part. Like we have done there, we address the single cell by indicating its position within the space as a n-dimensional vector (coordinate).

Syntax:

The given coordinate must, of course, be within the retina. In particular, *IntExpressionList* must consist of n elements of *IntExpression*.

Exp.37: Looking at a three-dimensional retina specified by `RANGE [-5,0,7]..[9,2,-1];`, we consider the following examples for cells:

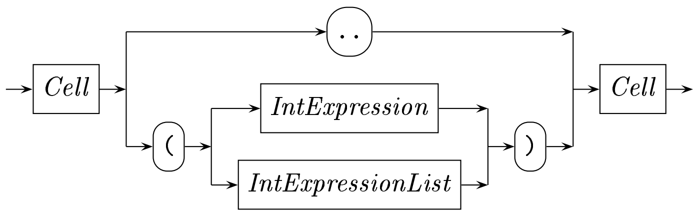
$\surd$	`[-5,0,7]`	(right)
$\times$	`[-5,0,8]`	(wrong)
$\surd$	`[abs(-7),0,0]`	(right)

3.2.3 Cell Grids

Loop

Generally we are now able to specify the whole initial configuration by assigning *CellExpression*s to each cell, one by one. It need not be underlined, how boring this could be! Usually there are quite big amounts of cells which are to be treated in the same way. SCARLET supports this by giving a possibility to look at all cells in subsets of rather simple forms at the same time. The complete syntax is given by

Syntax:



The basic form of all these sets is the cube, which is completely determined by two diametral corners *Cell*. We have already used this method to fix the range of the retina itself. (For further information refer to pg. 67). The cube must certainly be a part of the retina. (Therefore, it is sufficient to check that the coordinates of the corners are within the range of the retina.)

Besides the "full cube" we can also decide to consider only those sites which form the crossing points of special grids (grid points), in which the distances between two grid lines in a certain coordinate direction must be constant. As most general syntax of a set of grid points *Loop* we have

Step Size

$$Cell_1 \ (IntExpressionList) \ Cell_2$$

where $Cell_1$ and $Cell_2$ are suitable sites generating the cube and lying within the retina, and where the single *IntExpressions* of (*IntExpressionList*) are values for the step sizes in the single coordinate directions. ($IntExpression_1$ belongs to the first, $IntExpression_2$ to the second, and so on.) Looking at a n-dimensional space we have to put n values for n step sizes.

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

Furthermore we restrict the step sizes to positive nonnegative numbers, and $IntExpression_i$, $1 \leq i \leq n$, can only be zero, if the cube's extension in coordinate direction i equals one, what means that the i-th components of $Cell_1$ and $Cell_2$ are the same.

$\boxed{!}$

Although the syntax of *Loop* looks like a DO-WHILE structure, SCARLET evaluates all its expressions - especially (*IntExpressionList*) - only once to grant the step size to be constant.

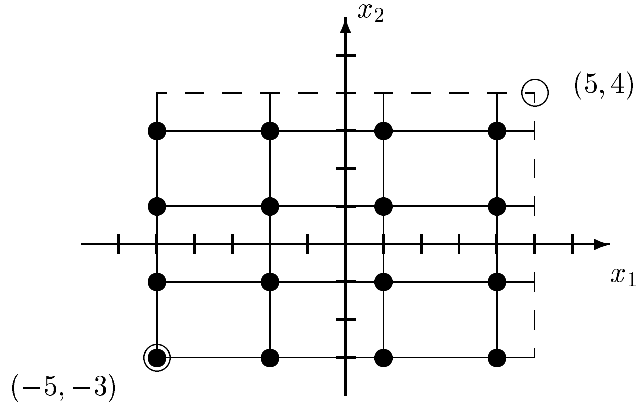
Note that SCARLET takes $Cell_1$ always for a grid point. Suppose that the set of coordinates of the full cube is called Q. The coordinate of a certain cell is a grid point, if there exist $m_1, m_2, \cdots, m_n \in \mathbb{Z}$ such that the following statement is valid:

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

Exp.38: In a two-dimensional space

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

generates the following set of grid points (pointed out by black dots):



For some special cases SCARLET allows abbreviations instead of the normal syntax:

Short Forms

- If *IntExpression* is the common step size for all coordinate directions, it is sufficient to indicate it once like this

  $Cell_1$ (*IntExpression*) $Cell_2$

- If furthermore in this case *IntExpression* is equal to one - the whole cube is meant - then we may omit the parentheses and type

  $Cell_1$ .. $Cell_2$

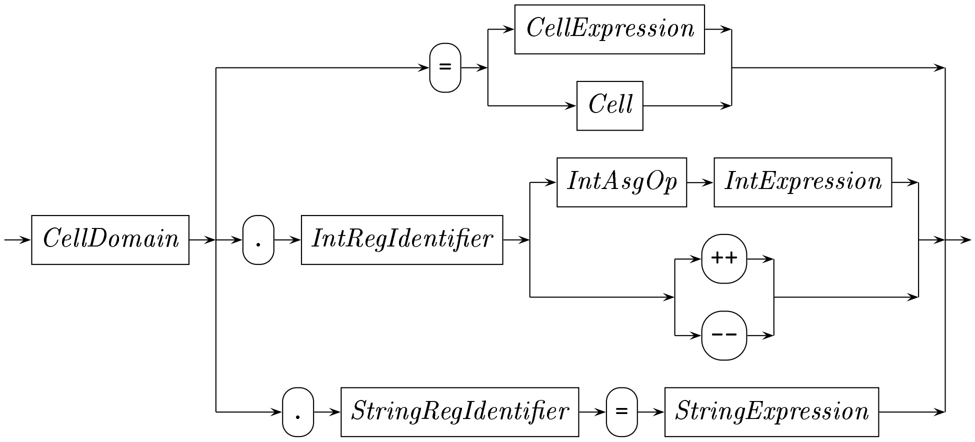
Exp.39: Three ways to describe the same set:

```
[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 Assigning Values to Sites

CellAssignment ;

Syntax:



After giving the whole syntax diagram, we will discuss its different parts one by one. Since we have already learnt something about both the syntax of a single cell and of sets of cells *Loop*, one can easily think of a syntax used to describe an assignment to these objects: we just treat them like normal variables of type `CELL`! But first we combine *Cell* and *Loop* to a new class called *CellDomain*. To let all sites meant by *CellDomain* have the value *CellExpression* we simply type[1]

Cell-Domain

CellDomain = *CellExpression* ;

Values of other Cells

If *CellDomain* should receive the state of another site *Cell*, we type

CellDomain = *Cell* ;

Please notice that every time *CellDomain* is a *Loop* set, all expressions on the right side of the equal sign are updated for each cell. This will also be done if the assignment affects only one register of the cell.

Single Registers

As former mentioned, it is possible to assign values only to a single register of the sites of *CellDomain*. In this case, instead of *CellDomain*, we use the projection

CellDomain.*IntRegIdentifier*

on an integer register *IntRegIdentifier* or

[1] As in former cases, the semicolon is not a part of *CellAssignment* (compare to *VarAssignment*)

CellDomain.*StringRegIdentifier*

on a string register *StringRegIdentifier* (see also 2.9, pg. 49.) and regard them (syntactically) as integer and string variables respectively. In 2.11, pg. 60, we have already pointed out how to do this.

Notice that there exists no analogue to

CellDomain = *Cell* ;

for single registers.

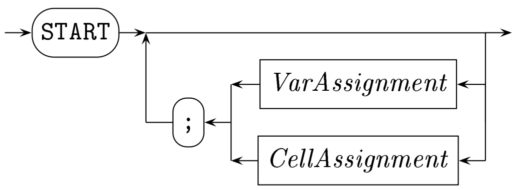
Besides this, it is not possible to assign the value of a site to a cell variable. That seems to be a restriction, but it is not. We can replace it by doing all the relevant assignments twice, to the site itself and to the cell variable!

$\boxed{!}$

But actually we have not seen yet that we are allowed to do so in a RDL program. That is what the last paragraph of this chapter is about:

3.2.5 Structure of the Statement Part *StatementPart*

Syntax:



This second (and last) main part of a RDL program starts with the keyword

`START`

Like this the statement part is set off against the preceding declaration part. `START` is followed by an arbitrary amount of assignments to sites of the retina

CellAssignment ;

or to cell variables

VarAssignment ;

in any order whatever.
SCARLET evaluates all these devices only once beginning at `START`. Contents of cells and cell variables are changed immediately after each assignment. This seems not to be worth mentioning, since it is the usual way of evaluating computer programs. But glancing at SDL, we will soon see that those programs form an exception to the rule... **Evaluation**

Exp.40: This is an example of a trivial two-dimensional retina. For there is nothing changed at all, the sites stay with the initialization values in their registers, thus here {`0,""`}. Nevertheless the keyword `START` must appear.

```
RETINA initial;
RANGE [0,0]..[5,5];
REGISTER INT num;
         STRING str;
VAR /* could be omitted */
CONST /* -"- */

START
```

Exp.41: Just another example of a retina.

The last device causes each site within a certain - but yet not fixed - area to get the state of another site that is found by a random procedure.

```
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)];
```

Chapter 4

SDL - Sigma Definition Language

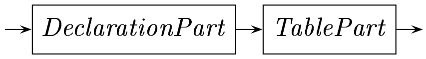
In this chapter we are going to discuss the second programming language, which is also an integral part of SCARLET, the so-called Sigma Definition Language (SDL). It deals with the typical behaviour of a CA, which is given by its local rule, a function often called σ. (This is, by the way, the reason for the name of the language).
To be able to design a local rule in SCARLET, we have to know

- the dimension of the space,
- the state set (that means, kind and number of used registers)
- the neighbourhood
- the boundary value (here: the state, common to all cells on the border)

Unlike this, the extension of the retina or even its initial configuration are not relevant in this context. Because of that, indications concerning the retina are saved in a separate file as a RDL program. (This has already been the topic of the preceding chapter.)
Later a complete CA will be constructed by SCARLET using the information of a RDL program as well as of a compiled SDL program. What are exactly the conditions to grant the compatibility? We are going to see that in thelast paragraph of this chapter.

Giving a look to the typical structure of a SDL program, we recognize immediately that it is - like RDL programs - subdivided into two main parts: the declaration part and the table part:

Sigma:

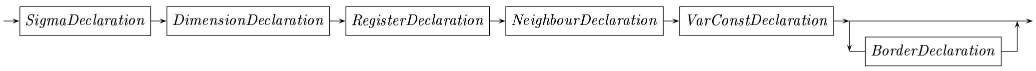


Sigma

4.1 Declaration Part

DeclarationPart

Syntax:

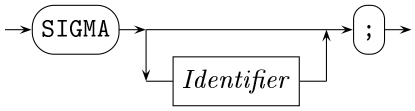


Within the declaration part we find information concerning the dimension of the space, used registers, neighbourhood, variables, symbolic constants, and the boundary value. Almost all of them must appear in the order given here. Only the parts about constants and variables *VarConstDeclaration* and about the boundary value *BorderDeclaration* are optional. Because of the form of *VarConstDeclaration* it is furthermore possible to mix up declarations of variables and symbolic constants.

4.1.1 Headline

SigmaDeclaration

Syntax:



A SDL program starts with

`SIGMA` *Identifier* ;

or simply

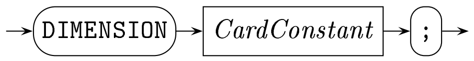
`SIGMA ;`

if the program should stay unnamed (compare to RDL, page 65).

4.1.2 Space Dimension *DimensionDeclaration*

The first thing we really have to do is to decide in favour of a space, which the CA should be based on. That may sound difficult, but in fact we just have to indicate the dimension of $\mathbb{Z}^n$, $n \in \mathbb{N}$, where a standard CA is always located in.

Syntax:



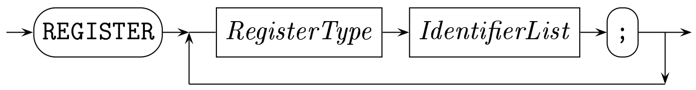
Here *CardConstant* - as a positive dimension (!) - has to be a non-zero element without sign of the class *IntConstant*.

Remark:
There are several reasons why zero cannot be a legal dimension. Nevertheless it is possible to construct such a CA (consisting only of one cell and **Dim. = 0** surrounded by boundary cells). We just take a CA of arbitrary dimension and restrict its range on one cell.

4.1.3 Register Declaration *RegisterDeclaration*

Also unevitable is the definition of a register structure for all objects of type `CELL`, especially for the sites themselves. We do this right the same way as in RDL, and even the syntax does not differ from the syntax there.

Syntax:



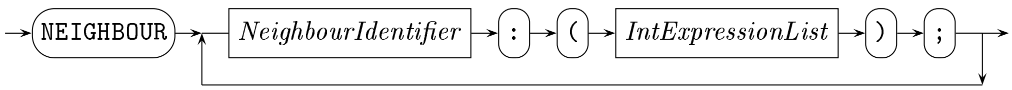
To get more information about the use please refer to section 3.1.4, pg. 69.

4.1.4 Neighbourhood *NeighbourDeclaration*

In the very beginning we have seen that a CA is in fact nothing else than a network of equal Moore-automata, the cells. In contrast to them, the cell's transition function does not depend on an external input but on the states of certain "cells in their neighbourhood" (neighbours). Of course, we have to state more precisely which are the neighbours of an arbitrary cell.
First of all it seems to be quite logical to postulate an homogeneous environment for an also homogeneous set of cells. According to this, neighbours

are normally defined by their relative positions to the cell which they should be neighbours to (compare to chapter 1.1). Like this, the neighbours of each cell can easily be calculated by adding this relative coordinate to their own coordinate.

Syntax:



Because the local rule cannot work without neighbours, never forget this declaration in your SDL programs. It starts with

`NEIGHBOUR`

followed by at least one entry of the form

NeighbourIdentifier : (*IntExpressionList*) ;

Consider n to be the space dimension (page 81), then *IntExpressionList* must consist of exactly n *IntExpression*. Since variables and symbolic constants are defined later, it is not allowed to use elements of the classes *IntVar*, *Int-ConstantIdentifier* and *IntReg*.

While *NeighbourIdentifier* is the name of the kind of neighbourhood to be created, (*IntExpressionList*) denotes the relative positionof an arbitrary cell to this neighbour, or more formally:
Suppose that K is the coordinate of an arbitrary cell, we get the coordinate N of its neighbour with regard to the kind of neighbourhood *NeighbourIdentifier* by the following relation:[1]

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

[1] The definition shows that it would be more adequate to speek about "influencing sites" than of "neighbours", because we usually expect a neighbourhood to be a symmetrical relationship, which is not given in this case.

Exp.42: For the two-dimensional general von Neumann neighbourhood H_1^2 of example 3 we obtain

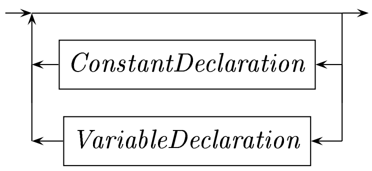
```
NEIGHBOUR  center :  ( 0, 0);
           left   :  (-1, 0);
           right  :  ( 1, 0);
           upper  :  ( 0, 1);
           lower  :  ( 0,-1);
```

4.1.5 Symbolic Constants and Variables

VarConstDeclaration

Between the declaration parts for neighbours and the boundary value, there is now space left to define variables and symbolic constants, if there is a need of it. Otherwise this part can be omitted.

Syntax:

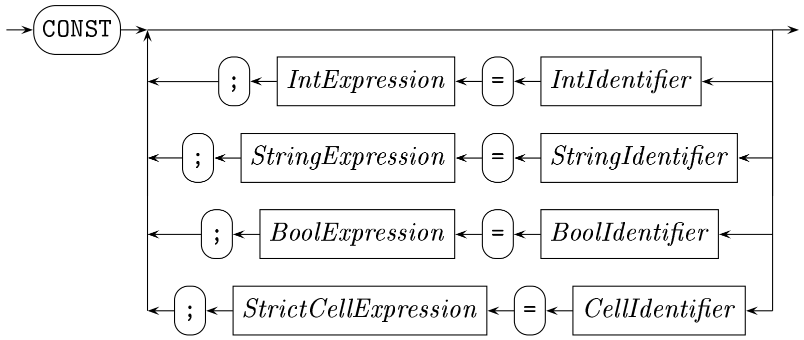


together with

VariableDeclaration:

and

ConstantDeclaration:



We are not going to give a description of how to use this syntax, because it is exactly the same as in RDL, where we have already discussed it. So please refer to these pages (pg. 69) as well as to the paragraphs about constants and variables in general (pg. 35) to get further information.

No Neighbours

There is but one thing special for SDL: while variables are initialized after each change of site the local rule is applied to, symbolic constants keep their values during the whole simulation, so that neighbours used as symbolic constants have no meaning here. That is the reason why neither in RDL nor in SDL it is allowed to use elements of *NeighbourIdentifier* in this part.

4.1.6 Boundary Value *BorderDeclaration*

In the formal standard CA model, there exists a special state that is - and only is - assumed by cells lying on the border of the retina. In this point Scarlet differs a bit from the standard model. Instead of a special state, it provides the function `isborder()`, which tells, whether or not the cell it is applied to is a border (or: boundary) cell. This additional state information allows us, on one hand, to distinguish them from the sites in the interior of the retina (as demanded by the model), but at the same time to treat both kinds of cells in the same way! This means concretely: Like for normal cells, the state of a boundary cell is divided up into registers, and can be chosen by the user as he pleases. That is exactly what the *BorderDeclaration* is for. The chosen state is common to all boundary cells and remains unchanged during the whole simulation. This method suggested for Scarlet seems to be more practical, because there is no need for additional entries which handle the boundary cases in the local rule table, if we want them to be treated like the normal cells. (Sometimes this saves indeed a lot of work!)

Default Value

As default value integer registers have got the `35`, and string registers `#`. To change these arbitrarily chosen values we use the following syntax:

Syntax:

where *StrictCellExpression* represents the new boundary value. It is only evaluated once, right before the start of the simulation, to grant it to represent a constant state throughout the run.

Remark:
In spite of the syntax, this is an assignment, so that `VOID` - as a register contents - causes the default values to stay invariant. VOID

Exp.43: For two registers, one of type `INT` and the other of type `STRING`, we could for example do the following boundary value changes:

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

4.2 Table Part

Finally we get to the main purpose of a SDL program: the definition of one (or more than one) local rules σ for the CA in form of a rule table. The local rule controls the whole course of the simulation. At each discrete time step all sites of the retina are updated simultaneously[2] depending on the entries of the table. (Usually they themselves refer to the states of certain cells in the neighbourhood.) Now one of SCARLET'S advantages compared with an own implementation becomes visible: because SCARLET uses internal procedures to control the simulation (e.g., to apply the table to each cell, to create the needed data types for the retina itself as well as for the results, ...) our original task is reduced to constructing the rule table(s) in a rather abstract form.
Nevertheless we have to talk a bit about how such a table will be interpreted by SCARLET, at least as far as it affects our proceding.

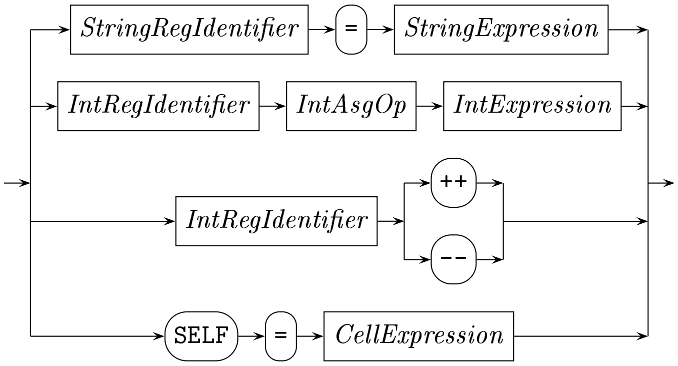
But before we can do this, it is necessary to introduce a special syntax for assignments to cells of the retina, which is only used in SDL.

[2]This is, of course, only possible if SCARLET is running on a computer with as many processors as sites. Otherwise the procedure is replaced with an equal serial one.

4.2.1 Assignments to Sites in SDL

SelfAssignment ;

Syntax:



Perhaps you have - reading the headline of this paragraph - remembered that we have already shown a syntax which enables assignments to arbitrary cells of the retina, accessing them by their coordinates, in the chapter about RDL, and you wonder why we do not use this syntax here as well? Simply because we do not want to affect only those cells with certain coordinates, but all cells at the same time! In RDL it has been our aim to create once and forever an initial configuration. Therefore, it has been necessary to treat different cells in different ways; but running the simulation, the situation changes completely: Now all state changes on the retina must be done by the global mapping, what is realized by applying the local rule to all cells simultaneously. Since SCARLET is fortunately an highly abstract programming language, it is sufficient to write each assignment in a table only once for all cells.

Generic Cell

This non existent cell, which all the assignment are made to, and by which actually every site of the retina could be meant, we will call "generic cell"; its syntactical name is `SELF`, so that we finally get the following syntax:[3]

`SELF` = *CellExpression* ;

Neighbours

Note that in SDL *CellExpression* can also be a *NeighbourIdentifier*. In this case *CellExpression* is no longer constant but varies from cell to cell.

`VOID`

For this is an assignment, the register contents of `SELF` is left unchanged, when `VOID` is contents of the corresponding register of *CellExpression*.

[3]Again the semicolon is not a part of *SelfAssignment* (compare to *VarAssignment*).

Exp.44: For a two-dimensional retina, three neighbours `left`, `right`, `center` (the generic cell), and a state structure consisting of three integer registers `num1`, `num2`, `num3` we could invent the following examples of assignments:

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

Single Registers

Finally it is again possible not to affect the whole state, but only single registers of the generic cell. According to what we have described several times before for similar situations (the most general case is found on page 49), we first have to select the register of interest by adding its name to the cell's name separated from each other by a point. For a register with type `INT` e.g., we would get `SELF.`*IntRegIdentifier*; but since `SELF` contains no information, we omit it and type

IntRegIdentifier

for integer registers and

StringRegIdentifier

for string registers, instead.

Now it is very easy to complete the syntax, because these objects are treated again like normal variables of type `INT` or `STRING`. We get

StringRegIdentifier `=` *StringExpression* ;

or

IntRegIdentifier `=` *IntExpression* ;

respectively.

Short Forms

In case of an integer register we have the same short forms with the same meaning as for integer variables:

IntRegIdentifier IntAsgOp IntExpression ;

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

For further information, please refer to 2.11, pg. 60.

Exp.45: In example 44 instead of `SELF = {center.num1+1,VOID,VOID}`:

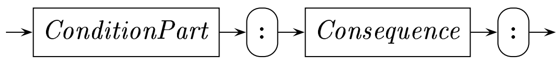
```
num1++;
```

4.2.2 Entries of the Local Rule Table

Entry

Before discussing a local rule table (short: table) as a whole, we are going to have a closer look at its components, the entries. A table is simply a listing with an arbitrarynumber of entries[4] The entries themselves conform to the following syntax:

Syntax:



Maybe it was already expected, but after having interpreted the names of the two main parts, at the latest, it becomes rather clear that an entry is nothing else than a conditional command, similar to an IF-THEN-ELSE structure. Actually the structure of *Entry* is not quite so strict, but we are going to see that in detail later. Nevertheless the condition part of *Entry* has always got - although sometimes only indirectly - a boolean expression, which causes the devices of the *Consequence* part to be executed if it is `TRUE`, and to be skipped otherwise.

Exp.46: By this little example (see [6]), we would like to illustrate a bit the structure of a table and its entries.
Therefore, we take a two-dimensional lattice, sites with one integer register `state`, and a neighbourhood consisting of the generic cell itself as well as the eight surrounding sites. The resulting CA is meant to be a simplified model for the **excitation of neurons** in a brain provoked by neurons in the neighbourhood.

In our model, a cell can assume three possible states:

- ready (to fire) $\leftrightarrow$ `0`
- firing $\leftrightarrow$ `1`
- refractory $\leftrightarrow$ `2`

[4]This phrasing may lead to the impression that the order within the list is of no importance, but this is not right, as will soon be seen in paragraph 4.2.3, pg. 92.

Furthermore we make up the following rules:

- a cell which is ready will fire, if at least three neighbours are firing.
- a firing cell will be refractory in the next timestep,
- and a refractory cell finally will be ready again in the following timestep.

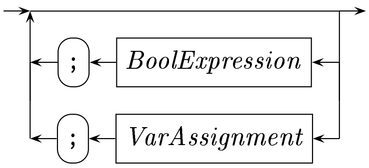
By a suitable choice of states (`0, 1, 2`) it can be achieved that all possible state changes can be realized by adding 1 and taking the remainder of a following division by 3 . Then the belonging local rule table contains only two entries: The first entry checks whether the site is ready. If not (`center.state > 0`), the addition in the first consequence part will be done (and all the following entries will be ignored). If the site is ready, the first consequence part will be skipped, and SCARLET will go on with the second entry. Here it is actually not necessary to check again if `center.state == 0`, this is already clear. But, if the sum of neighbours with state `1` is greater than or equal to 3, - therefore, each value is bitwise compared with `1` and the results are summed up - then the site will change its state from ready to firing, otherwise it will keep its state:

```
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.  entry */
  center.state > 0;     /* Condition */
       :  state++; state %= 3; :      /* Consequence */

/* 2.  entry */
  center.state==0;     /* Condition */
  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 */
```

Condition Part *ConditionPart*

The more general form of table entries compared to IF-THEN-ELSE structures is mainly caused by the condition part:

Syntax:



As visible in the syntax diagram above, it consists not only of one but as many conditions as we want:

BoolExpression ;

Furthermore there can also be assignments to variables:

VarAssignment ;

Both kinds of units may be mixed up and put in any order. SCARLET will interprete them according to the rules given below:

- The boolean value of a condition part is given by the conjunction of its boolean expressions. If the result is `TRUE`, the devices in the consequence part will be executed; if it is `FALSE`, the consequence part will be ignored.

  Exp.47: Note: `bool1 || bool2; bool3;` is the same as `(bool1 || bool2) && bool3;`

- If *ConditionPart* has got no boolean expression at all, its boolean value is `TRUE`.

- The order in which the different units separated by semicolon appear in *ConditionPart* is also the order of their execution. Variables are updated immediately.

Nonstrict Evaluation

Notice that SCARLET works nonstrictly, this means, as soon as the evaluation of one boolean expression leads to `FALSE`, SCARLET will skip the rest, because it is already sure that *Consequence* will be ignored.

Exp.48: We give again a look at example 46. To calculate `sum` - instead of in the second entry - already in the first after `center.state > 0;` (as shown here), would lead to mistakes: In case that `center.state == 0;` is TRUE, `center.state > 0;` would be FALSE, so the evaluation of the first entry would be stopped immediately and `sum` would not be computed:

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

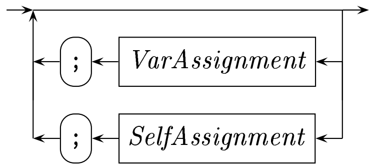
Remark:
It is quite important to realize that the condition part only allows assignments to variables, but not to cells. **!**

Hoping all these rules and conditions on the condition part become a bit clearer, we give a little view of what it is for. During the simulation, the table will be applied to each cell in order to update their states. As soon as a cell complies with the condition of a certain entry, its state changes according to the corresponding consequence part. Therefore, the state of a site can only be changed in *Consequence*, but not in the condition part. Once a site is updated, the rest of the table is no longer of interest to it: It can be skipped.
Besides, since the state usually depends on the states of the cell's neighbours, it is quite comfortable to just string the conditions on the neighbours in the condition part together.

Consequence Part *Consequence*

The consequence parts of the entries contain in fact the most important actions for the evolution of the CA's configuration. But they are only executed, if the belonging *ConditionPart* has been executed before, and all its boolean expressions have been evaluated to TRUE (compare with the paragraph above).

Syntax:



In addition to the already known variable assignments

VarAssignment ;

we also find assignments to the generic cell `SELF` in *Consequence*, described in paragraph 4.2.1

SelfAssignment ;

(The semicolon are not a part of these assignment units, but of the syntax of *Consequence* to separate them from each other.) Order and amount of those units are arbitrary.

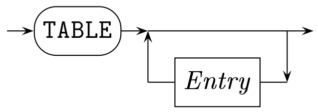
Evaluation

As in *ConditionPart* the order of their occurence is also the order of evaluation. Variables are updated immediately as well, but not the value of the generic cell! Because of the simultaneity of events the sites are only updated after a full simulation step.

4.2.3 Local Rule Table

Table

Syntax:



Each table is started with the keyword

`TABLE`

(The end of a table is either the begin of an additional table or at the same time the end of the entire SDL program.) After `TABLE` we place the already discussed entries of the form

ConditionPart : *Consequence* :

One table may contain as many entries as we want.

Remark:
For both *ConditionPart* and *Consequence* end with a colon, it may be of advantage to think of a layout which makes it possible to oversee the structure. Because of the author's lack of imagination we simply suggest to set every second colon on tab.

Layout

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

Interpretation of the Table

for so far the syntax. In order to understand how such a table is interpreted by SCARLET, we remind once again of the properties, the local rule of a CA has got. It is a function σ that is in every time step simultaneously applied to each cell of the retina, in order to change their states depending on the states of certain cells in their neighbourhood. With σ we talk about a function from S^d to S, if S denotes the state set and the neighbourhood is of order d.

Now we are going to compare this to the structure of a local rule table as a whole (which should finally represent σ in SCARLET). For the features of the condition part as well as the consequence part have a big influence on it, they will be shown explicitely again there. Let us suppose that $Entry_1, \ldots, Entry_m$, $m \in I\!N$ are the entries of the table, $c_1^{(i)}, \ldots, c_{j_i}^{(i)}$, $j_i \in I\!N_0$, $i \in \{1, \ldots, m\}$, all variable assignments of $ConditionPart_i$ in right the order in which they are given there, except those which are ignored because of the nonstrict treatment (pg. 90). Let $b_i, i \in \{1, \ldots, m\}$, be the boolean value of $ConditionPart_i$ (compare with pg. 90), then a SDL program with this table roughly implies the algorithm beneath:

$$
\begin{array}{l}
c_1^{(1)};\ c_2^{(1)};\ \ldots\ ;\ c_{j_1}^{(1)}; \\
\text{IF } b_1 == \text{TRUE} \\
\quad \text{THEN } Consequence_1
\end{array}
$$

ELSE
  $\{c_1^{(2)};\ c_2^{(2)};\ \ldots\ ;\ c_{j_2}^{(2)};$
   IF $b_2 ==$ TRUE
     THEN $Consequence_2$
     ELSE
       $\{c_1^{(3)};\ c_2^{(3)};\ \ldots;\ c_{j_3}^{(3)};$
        $\vdots$
       $\}$
  $\}$

It describes what happens, when the table is applied to an arbitrary cell z. According to this, a single step of the simulation will certainly require as many calls of the table part as there are cells in the lattice. This is automatically done by SCARLET.

Remember furthermore that

- all **variables** are initialized before each call of the table,

- **symbolic constants** are only initialized before each simulation step, and

- the **cells** of the retina are updated only after each simulation step.

But back to the algorithm above. It says, 'Starting with the first entry, execute the commands c_i and evaluate the boolean expressions in $ConditionPart_i$ of the i-th entry in the order they are given. If the value of all boolean expressions b_i is `TRUE`, then execute $Consequence_i$ - that means, assign the given values to sites and variables - skip the rest of the table, and start from the beginning with another cell.
Otherwise ignore the rest of $ConditionPart_i$ as well as $Consequence_i$, as soon as the evaluation of one of the boolean expressions leads to `FALSE`, and go on with $entry_{i+1}$'.

! However, during one run of the table part, there will at most one consequence part of every table be executed.[5] It is exactly the first one for which all conditions in *ConditionPart* are satisfied. This is remarkable, for the belonging

[5]In paragraph 4.2.4, pg. 98 we will see that a SDL program can consist of more than one table.

Consequence is the only part which may affect the state of the site, or, to put it in other words, in which there can be assigned a value according to the function σ. In the next paragraph we will see which influence the order of the entries within a table has on the choice of that *ConsequencePart* that will be the affecting one.

Exp.49: We give again a look at example 46 in order to show the table in an algorithmic form:

```
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};
```

Note: `sum` is only calculated, if `center.state == 0`. That is the reason for the additional condition before `sum = ...` .

Order of the Entries

In order to show that the order of the entries in a table play quite an important part in the course of the simulation, we give the following example:

Exp.50:

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

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

applied to the following 1-dimensional lattice with length l

1	0	0	0	...	0	0

causes a signal "1" to run from the left to the right side at "speed 1". Arriving there it disappears.

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

But just exchanging the two entries, the behaviour will be completely different: The signal disappears already after the first step.
The reason for that is obvious, because all the combinations of neighbour configurations for which condition 1 is satisfied form a subset of the ones which conform to condition 2.
If condition 2 had been given in a form which had left the intersection empty (`left.state != 1;`), an exchange would not have led to a different behaviour.

Thus for not losing the track creating complexer tables, it seems indeed to be of advantage to follow the strategy we have put together:

How to Order Entries

Suppose that $E_1, \ldots, E_m, m \geq 2$, are the entries of a table, $z \in \mathbb{Z}^n$ the coordinate of the looked at cell, $N = (n_1, \ldots, n_d)$ the n-dimensional neighbourhood of order d. In this paragraph, M_i will represent the set of all d-tuples $(s_1, \ldots, s_d) \in S^d$ for which the condition of entry E_i is satisfied, if $s_1 = c\,(z + n_1),\ \ldots,\ s_d = c\,(z + n_d)$ for a configuration $c : \mathbb{Z}^n \longrightarrow S$. We get the following rules:

1. If $M_i \cap M_j = \emptyset, i \neq j$, then the order in which E_i and E_j appear in the table does not matter.

2. If $M_i \subseteq M_j, i \neq j$, then entry E_i (with the more special condition) must precede E_j.

3. All the other cases can be reduced to 1. or 2. by splitting up the sets M_i in a suitable way.

Nondirect State Assignments

We have to pay particular attention to the behaviour of a local rule table, if the generic cell does not receive its new state from a *SelfAssignment* unit. Such a situation rises, if

1. none of the entry conditions is fulfilled by the configuration of the neighbours.
2. The configuration of the neighbours fulfils a *ConditionPart*, but in the belonging *Consequence*
   (a) does exist no *SelfAssignment* unit,
   (b) the state of `SELF` is only changed in parts. (That means, only single registers are affected, pg. 87).

In case of 1. and 2.(a) the site keeps its old state. In case of 2.(b) only the values of registers which have not been captured by the assignment stay unchanged.
But in every case the new state of a site depends on its old state in the previous time step!

That is no problem, if the site itself is a neighbour (that means, if $(0,\ldots,0) \in \mathbb{Z}^n$ belongs to the neighbourhood), for then dependencies of this kind are allowed. The table constructed in SDL still competes with the demands on the local rules in the abstract model.

But troubles arise, if the site to be changed should not be a neighbour! In this case the user has to take care by himself that situations as described above do not become reality, if he wants the table to represent a valid local rule. $\boxed{!}$

Incomplete Tables

While it is rather easy to avoid 2. (since it is sufficient to check each entry separately), for 1. it is sometimes so difficult that the only chance is to simply guarantee that the table is complete ($M_1 \cup M_2 \cup, \ldots, \cup M_m = S^d$; the definition of M_i can be found on page 96).

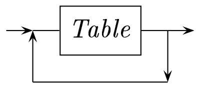
If the table should indeed stay incomplete, it must be granted that all neighbour configurations missing are really not assumed.

Remark:
Therefore, the appearance of the initial configuration must also be taken in $\boxed{!}$

account!

4.2.4 Structure of the Table Part *TablePart*

Syntax:



More than one Table

The statement part just consists of one or more elements of *Table*. In the last paragraph we have already seen how they work. If there is more then one, then in each step every table will be applied to each site in the known way starting with the table at the very top. Thus it is a bit like combining functions.

4.3 Compatibility of RDL and SDL Programs

We hope this manual has helped the SCARLET user to gain all skills necessary to successfully compose RDL and SDL programs. But the programs are still kept in different files, though and will be put together to a complete CA only later for the duration of the simulation.
Like that, it is always possible to combine different initial configurations with different local rules without occupying too much space.
But, of course, not every retina goes with every local rule. Only the conditions listed beneath guarantee their compatibility:

1. The space dimensions must be identical.
   (In RDL programs it is indirectly defined by the `RANGE` command, in SDL directly by `DIMENSION ... .`)

2. The state sets must be the same, more precise:
   In both programs there must be declared the same register types in the same order.
   (They do not need necessarily to have the same names.)

Variables, Symbolic Constants

The names of variables and symbolic constants used do not need to be unique for both programs, since they are only known in the program, which they have been defined in.

Exp.51: Parts of a RDL and a SDL program which are compatible to each other:

```
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;
     ...                                     ...
```

Remark:

Besides the above mentioned ones, there exist no other <u>formal</u> restrictions on a successful combination of RDL and SDL programs. But it is highly recommened to check, if they semantically match with each other, too. (What normally does not seem to be worth a single thought, may well cause some troubles in reviewing the program after a while...)

Semantical Restrictions

If we suppose that both programs to be put together are based on the same abstract CA (that means they are independent of the implementation), then we have essentially to take care about the state set, in detail

- how is it subdivided into registers (how many registers are there; in which order are they) and which meaning do they have;
- which meaning do the single register contents have; and close to this
- which values can be assumed by each register. (This is especially of interest, when the generic cell should not be member of the neighbourhood, compare to pg. 97).

Chapter 5

User Interface

Finally we would like to introduce the reader to SCARLET's user interface, which is a window surface created under Motif. It permits us to watch and to control the simulation.

Unlike many other software packages concerned with CA, SCARLET does not restrict CA to having only a few states, which are during the simulation often shown as coloured dots on the screen, but supports the usage of complex data types and state structures. Because of that fact SCARLET generally displays the whole state information (or parts of it) as defined in the program.

Because we do not know to which degree the reader knows how to handle the X11 window system, this chapter contains both, a short description of the most important SCARLET interface functions as well as a detailled one to refer to in case there are still unanswered questions. In every case you should try to first explore the features of the user interface on your own, and only consult the manual, if troubles arise.

5.1 Short Description

1. **Create** a local rule table as a **SDL program** as well as a matching retina together with an initial configuration as a **RDL program** using a text editor. Save them in different files (suggested file extensions: `sdl` and `rdl`).

2. **Run** SCARLET by executing the program file:

   `./scarlet` <Return>

3. **Activate** a local rule table which is already compiled or Sigma

   load and **compile** a **SDL program**. Compile

4. **Load** an initial configuration (**RDL program**) which is compatible to the local rule table. Retina

5. *Select the desired* ***view and simulation options***

   - *part of the retina shown*
   - *state indications and coordinates*
   - *coordinate system*
   - *(In case of a 1-dimensional retina: Wrap)*
   - *step size*

6. **Run** as many time steps of the simulation as set by the chosen step size. Run

7. *If necessary:* ***Change***

   - *retina or local rule table*
     (if it is not compatible choose `Drop old` *to react to the appearing message.)*
   - *view or simulation options*

   Reload *the initial configuration.* Reload
   *(**Important:** the current configuration displayed on the screen will be lost!)*

8. **Quit** SCARLET. Done

5.2 Detailled Description

5.2.1 Run / Quit Scarlet

In order to use Scarlet, you are in need of a computer system with operating system SUN Solaris, version 2.0 or higher, together with the X11 window system.
After having installed Scarlet successfully on your system, there are several possibilities to run the program.
If you use a terminal or console window simply set the current directory to the directory where you keep Scarlet and type

`./scarlet` <Return>

(In case this directory is already included in the system path, there is, of course, no need to change the directory.)
Certainly you may also use the file manager. Then just activate the executable `scarlet` in the Scarlet directory by a double mouse click on the icon.

Remark:
Please remember that the user interface is only intended for running simulations. So first do not miss to create and to save the RDL and SDL programs you need using a text editor (e.g., vi, emacs,...).

Once you have initiated a program run Scarlet's user interface will appear in a window on the workspace. It will probably look somehow like in figure 3.

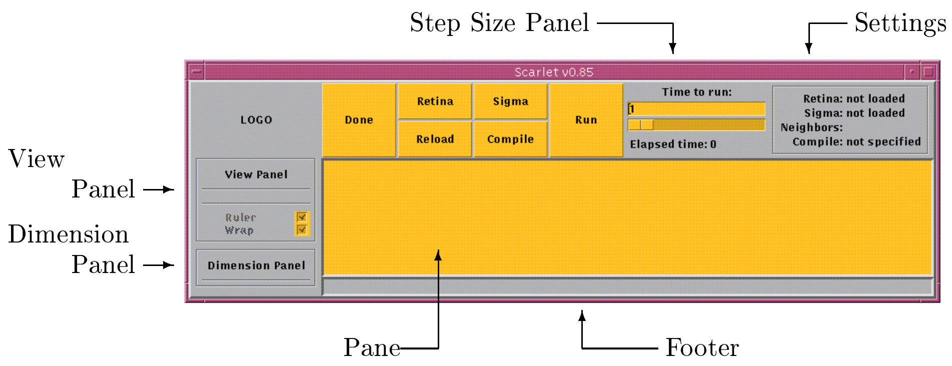


Fig. 3: The Scarlet program window.

At that time the **pane** of the window is empty. Later the configuration of the retina loaded will be displayed there.
In the **footer**distinct types of messages will be shown to report on SCARLET's status and actions.
A frame box located in the right top corner of the window informs about current **settings**. At the moment neither a retina nor a local rule table is loaded and, therefore, no neighbourhood is active. Furthermore no SDL program to be compiled is designated.

To run a simulation SCARLET must be told which local rule table and which initial configuration it should use. It is of no importance which file is loaded first.

To quit SCARLET click the `Done` Button (or use the window menu button).
Please note that all results will be lost, when you exit the program.

Done

5.2.2 How to Handle the X11 Window System

Sometimes it seems indeed that the easiest things cause the most irritating mistakes and problems. That is why we decided to also include a (very) short description of some of the most important actions on X11-windows:

- **Activate**: Single mouse click on the window.
- **Close**: Double-click on .
- **Move**: Move the mouse pointer on the header of the window. Press the left mouse button while dragging the pointer to the new position.
- **Change size**: Move the mouse pointer somewhere on the frame or on a corner of the window. After the pointer has changed its shape press the mouse button and drag the edge of the window to the new position.
- **Minimize**: A single click on closes the window to an icon.
- **Restore** the window from the icon: Double-click on the window icon.
- **Maximize**: A click on lets the window occupy the whole screen.
- **Restore window from maximization**: Click again on .

By the way, all actions named above (except "Activate") can be performed clicking on and choosing from the pull-down-menu there (or using the short cut indicated there).

5.2.3 Load a Retina

In order to load a new retina and - at the same time - to get an initial configuration click the button `Retina`. A new window will appear(see figure 4), which enables the user to choose a retina file from an arbitrary directory. SCARLET's retina interpreter will accept every valid RDL-program. Other files are rejected returning an error message in a separate window (i.e., `parse error...` together with more detailed information on the kind of error).

If the file has been accepted the settings will change. The name of the retina - which has been determined in the headline of the belonging program using (`RETINA` <*Name*>`;`), - and its range will be shown in the settings-box behind `Retina:`
But, of course, it is much more important to notice that the initial configuration of the retina loaded is displayed in the pane of the window. You find more about that in 5.2.6.

Sometimes it may be desirable to restore the initial configuration again after having performed some steps of the simulation. But now that the name of the file is already known it is not necessary to open the load window and to do all the choosing again. In this case simply use the `Reload` button.

The Retina Selection Window

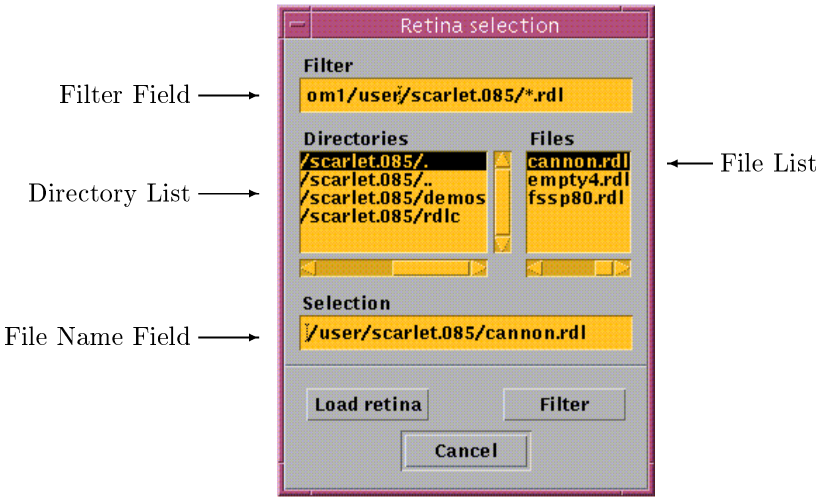


Fig. 4: **Retina Selection Window**

The **filter field** located in the upper part of the window determines which kind of files are shown in the file list. As long as the filter has not been changed all files ending on `.rdl` located in the SCARLET directory, (here `scarlet.085`) will be displayed:

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

Indeed SCARLET suggests the usage of file names with this extension for RDL programs; but actually any file name will do. To change the filter settings click on the filter field and enter your choice (i.e., `.../scarlet.085/*` to display all files in the SCARLET directory).

Like all changes made in the load window the settings will be kept until you quit SCARLET.

To select a directory from the **directory list** use the scrollbar or the arrow keys. Confirm you choice with a double-click on the directory or press the `<Return>` key to choose the highlighted one.
(Note that `/.` stands for the current and `/..` for the directory above. Like that it is possible to reach each point in the directory tree.)
Directory changes will automatically change the filter settings and the path in the file name field in an appropriate way.
The **file list** displays all files in the chosen directory that are in accordance with the filter settings. To open a file select it from the list by a double-click or directly enter name and path in the **file name field** below and press `<Return>` or the `Load Retina` button instead.

Occasionally it may be necessary to update the list of files (or directories) - i.e., because some objects have been added or deleted in the meantime - this can easily be done clicking on `Filter`.
`cancel` will close the load window without opening a file.

5.2.4 Load a Local Rule

Unlike RDL programs, which are loaded as source-code files to be read by an interpreter SDL programs must be compiled before they can be activated. This is easily be done by the user from out the user interface (see below). The SCARLET will automatically save the compiled version of the rule table in a new file. (Otherwise an error message will appear.) Moreover the name of the originate source-code file will be displayed in the settings-box.

`Compile:` <source-code file name>

Please notice that compiling a rule table will not activate it! This must be done in a second step.

To activate a local rule table which is already compiled the table file generated during compilation must be loaded (see below). When SCARLET displays the name of the table (defined in the headline of the SDL program) and the belonging neighbourhood in the settings-box, the rule table has been activated successfully.

To perform one of the actions described above click the button `Sigma`. SCARLET will open an additional window consisting of two smaller units, each of them very similar to a 'Load Retina' window. It is shown in figure 5.2.4.

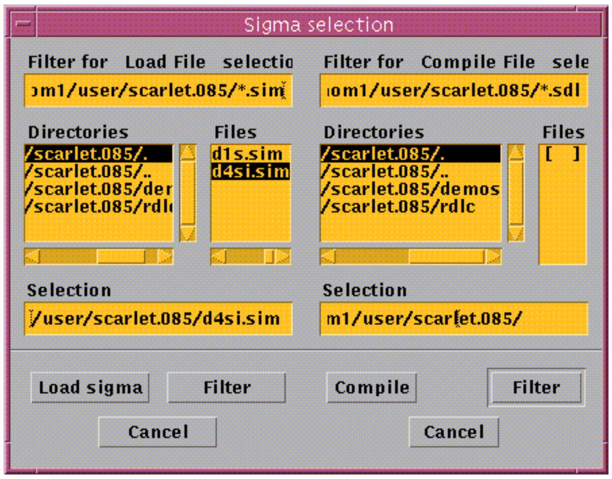


Fig. 5: **Sigma Selection Window**

The right unit is intended to select (and to compile) a SDL source-code file, the left one to load an already compiled table. The filter settings (only) suggest the usage of file names with extension `sdl` to label source-code files and `sim` to label compiled files.
After an appropriate file has been selected, it may be compiled clicking the `Compile` button (if it was a source-code file) or activated clicking `Load sigma` (if it was already compiled).

You find more about the general features of the **sigma selection window** in 5.2.3.
To perform the compilation you may also use the `Compile` button in the main window.

5.2.5 Incompatibility

Sometimes it may happen that a certain retina should be loaded although an incompatible local rule table is still active, or the other way around - maybe by accident or because it has been your intention to load a completely new CA. However, SCARLET will indicate the invalidity of this action displaying an error message (see figure 6).

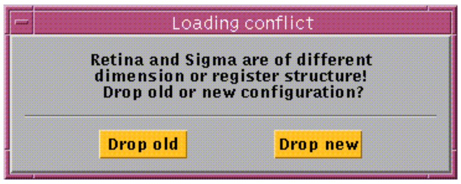


Fig. 6: **Incompatibility Message**

In case of a mistake you may want to keep to the old situation (and not to add the second incompatible component); then you choose `Drop new`; but normally you will be sure you want to load the new file, then click on `Drop old`; SCARLET will remove the existing incompatible component automatically.

5.2.6 View Options

As soon as a retina is loaded, parts of it will be shown in the pane of the SCARLET program window in a graphical manner. First a 2-dimensional slice is shown such that the cell with coordinate $(0, ..., 0)^t$ is located in the left corner on the bottom of the pane. The size of the area visible depends on both the view options and the size of the SCARLET program window. To change the size of the window will not affect the cell in the left bottom corner of the pane. It keeps its location there.

Each cell of the retina is symbolized by a small frame box. Normally it will contain certain information about the cell's position (see below) as well as

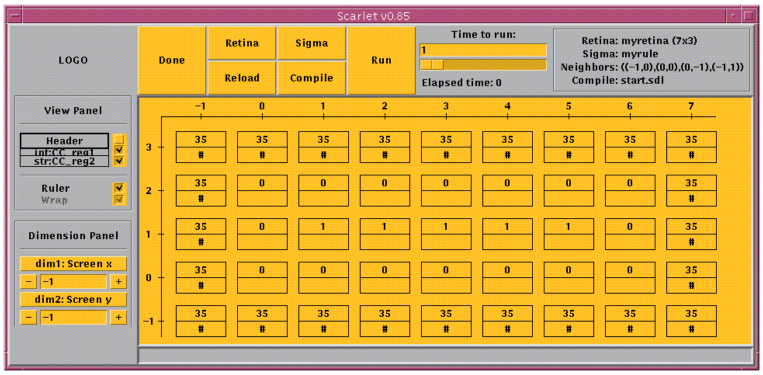
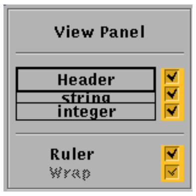


Fig. 7: Scarlet, **ready for a simulation run.**

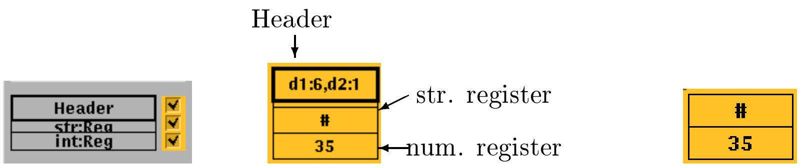
the complete state divided up into the single registers.



View Panel

But these default settings can be changed. At the left border of the window the **view panel** is located. There you find all types of registers in use listed. By toggling the check boxes you can decide which of the registers should be displayed.

The **header** of each cell shows its position on the 2-dimensional slice of the retina. The first value is related to the coordinate direction which forms the horizontal axis (x-axis) on the screen, the second value belongs to the vertical axis (y-axis).
(I.e., `d3:10,d2:4` means that the coordinate system on the screen is given by the third (horizontal) and the second (vertical) dimension. In this system the belonging cell has got the coordinate $(10, 4)$.)
If header is set off, cells are shown without header, thus without any information about their positions.



View Panel   a single cell   `Header` is on

Fig. 8: A cell with one numerical register and one string register; the contents (`35` and `"#"`) are the default settings for a cell on the boundary.

Beneath this group you find another two more options: `Ruler` und `Wrap`.

Fig. 9: **Ruler** and **Wrap** function.

If you set the option `ruler` off, the **coordinate axes** will be suppressed. On the other hand the coordinate axes may be very useful, if the single cells should be displayed without header.

Whenever you use a 1-dimensional retina, you may profit from the `wrap` option. (Otherwise `wrap` is disabled.) `wrap` is kind of a carriage return function for line shaped retinas. It causes the retina to be cut whenever it reaches the right border of the pane and to go on in the "next line". Like that it is possible to watch a larger area of the retina than suggested by its shape.

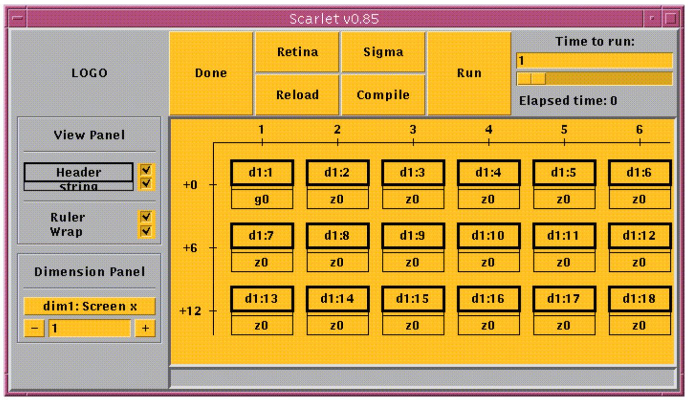
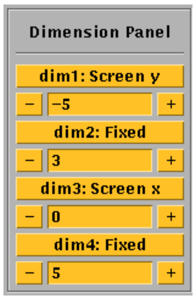


Fig. 10: Parts of a 1-dimensional retina with `wrap` on.



Dimension Panel

If `wrap` is set on the `ruler` function has a slightly different meaning: to get the position of a certain cell (in the 1-dimensional space!) just add the value on the vertical axis to the one on the horizontal axis.
If enabled the default value for `wrap` is "on".

Moreover, on the left side of the window you find the **dimension panel**, which controls which part of the retina is shown in the pane. Each dimension has got its own control elements.

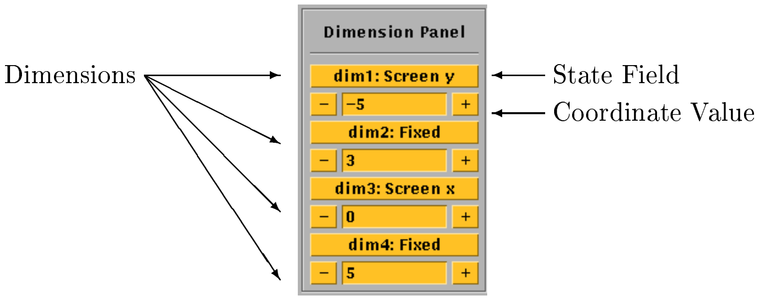


Fig. 11: **Dimension Panel:** 4-dimensional space; dimension 3 and 1 as coordinate axes, the left bottom corner of the pane occupied with the cell with coordinate $(-5, 3, 0, 5)$.

To determine the retina area of interest do the following:

1. Choose the dimensions which should be the coordinate axes on the screen (labeled with `Screen`). - The other dimensions take fixed values for all sites (`Fixed`).

   Scarlet will always suggest to take the first dimension for the x- and the second for the y-axis.

   These settings can be changed clicking on the **state field** of <u>that</u> dimension that is supposed to become the new x-axis. The former x-axis will then become the y-axis!

   (Thus, if both coordinate axes should be determined, <u>first</u> choose the new y-axis and only then the x-axis.)

2. Now fix the coordinate of the cell which should appear in the left bottom corner of the pane.

   Enter the desired values directly using the keyboard or make use of the the + and - buttons. (Only enter integer numbers. Signs are to be put directly in front of the numbers.)

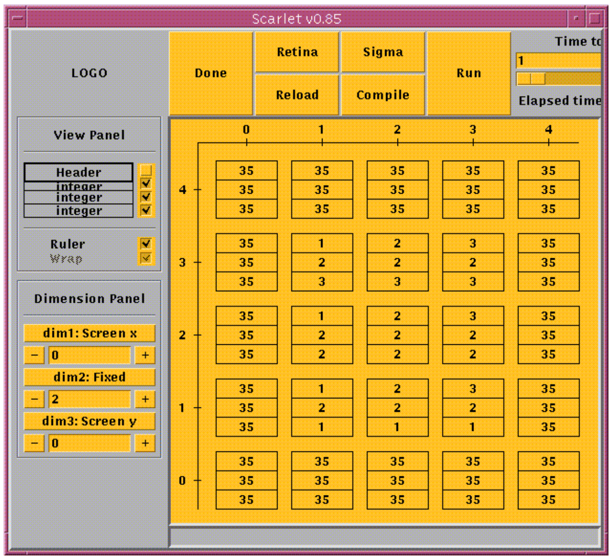


Fig. 12: Effect of the Dimension Panel: The 2-dimensional slice of a 3x3x3 retina with the second dimension fixed at value 2. Three numerical registers contain the cells' coordinates.

5.2.7 Simulation

Arriving at this paragraph hopefully everything is ready to finally dare a simulation run: a local rule table is compiled and activated, a compatible initial configuration is added, thus all potential difficulties using the SCARLET programming language have been mastered.
After all it is easy to set the simulation going: just click the Run button! ... and it will stand still again almost immediately! No wonder; each push of the button will perform only as much steps as given by the step size panel

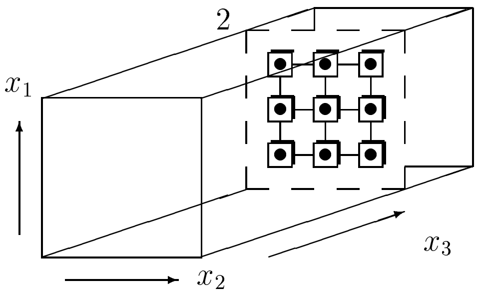


Fig. 13: Fig. 12 again in a more abstract way.

(see 5.2.8). The default value is `1`.

The display `Elapsed time` shows how many simulation steps have already been performed. SCARLET allows at most $2^{31} - 1 \approx 2,14 \cdot 10^9$ steps.

5.2.8 Simulation Options

Change Step Size

Perhaps you are not interested in watching every single configuration assumed during the simulation. Then you may skip them. To achieve this enter the desired step size - that means the number of steps to be performed before drawing the configuration again - into the field `Time to run:` of the **step size panel** located on the upper border of the SCARLET window. Only positive integers less than $2^{31}-1$ are legal step sizes. Do not forget to confirm the input with `<Return>`.

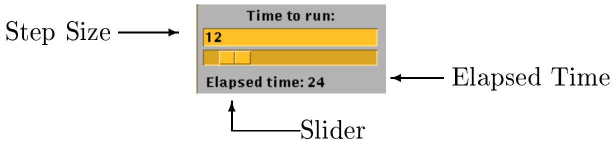


Fig. 14: **Step Size Panel.**

Instead of entering the step size directly the **slider** below can be used. After you have positioned the mouse pointer over the slider, press the mouse button and drag the slider until the new step size value is displayed. First the slider covers values from 1 to 100. But the maxim value can be changed entering something in the field `Time to run:`.

Bibliography

[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