Tutorial

SAC 1.0

Single Assignment C

August 24, 2009

Sven-Bodo Scholz

University of Hertfordshire

Department of Computer Science

Hatfield, Herts, AL10 9AB

United Kingdom

Contents

Chapter 1

Running the first program

The following instructions will help you write your first SaC program.

1.1 A Checklist

To succesfully write and run your first SaC program, you will need:

- An **ANSI C compiler**, such as `gcc`. Though not needed directly, the SaC compiler relies on it.
- The **SaC compiler** `sac2c`. It can be downloaded at <http://www.sac-home.org/>. For convenience, you should make sure, that `sac2c` is located in your `$PATH`.
- The **SaC standard library** that comes with `sac2c` as part of the SaC distribution. To make sure `sac2c` will be able to find it, the environment variable `$SACBASE` should point to it, i.e., `ls $SACBASE` should yield at least the subdirectories `runtime` and `stdlib`.
- Your favorite **text editor**, such as `vi` or `emacs`.

1.2 Create your first SaC Source File

Start your editor and type the following program:

Listing 1.1: Hello World

```
1 use StdIO: all;
2 use Array: all;
3
4 int main()
5 {
6   printf( "Hello World!\n");
7   return(0);
8 }
```

As you can see, it has a strong resemblance to C. The major difference are the module use declarations at the beginning of the program. They are explained to more detail in another trail of the tutorial in Chapter **??**. For now, it suffices to keep in mind, that these two use declarations for most experiments will do the job.

In order to proceed, save this program into a file named `world.sac`.

1.3 Compile the Source File and Run the Program

The SaC compiler invocation is similar to the standard invocation of C compilers. A typical shell session for compiling `world.sac` could be:

```
> cd /home/sbs/sac/
> ls
world.sac
> sac2c world.sac
> ls
a.out      a.out.c    world.sac
> a.out
Hello World!
>
```

The compilation process consists of two steps. First, the SaC compiler generates a C file, which then is compiled into target code by utilizing the system's C compiler. If no target file name is specified, the intermediate C file is named `a.out.c` so that the subsequent invocation of the C compiler creates an executable named `a.out`.

In the same way the default target name `a.out` is borrowed from standard C compilers, the `-o` option for specifying target names is adopted as well. For example, `sac2c -o world world.sac` results in files `world.c` and `world`.

Note here, that the compiled program is linked statically and does not require any further linking or interpretation.

Chapter 2

Array Programming Basics

This trail gives an introduction to the basic concepts of array programming in SaC. It consists of two lessons: **Arrays as Data** and **Shape-Invariant Programming**. In the former lesson, the major differences between arrays in SaC and arrays in more mainstream languages are explained. The lesson **Shape-Invariant Programming** gives an introduction into the most important array operations available in SaC. Based on these operations, several small examples demonstrate how more complex array operations can be constructed by simply combining the basic ones.

2.1 Lesson: Arrays as Data

In SaC, arrays are the only data structures available. Even scalar values are considered arrays. Each array is represented by two vectors, a so-called **shape vector** and a **data vector**. An array's shape vector defines its **shape**, i.e., its extent within each axis, and its **dimensionality** (or **rank**), which is given implicitly by the shape vector's length.

The section on **Defining Arrays** explains how arrays of various dimensionalities can be defined in SaC, and how they can be generated via nesting. Furthermore, some elementary notation such as **scalars**, **vectors**, and **matrices** is defined.

The section on **Arrays and Variables** discusses the purely functional array model used in SaC.

2.1.1 Defining Arrays

In this section, several means for specifying arrays are explained.

In principle, all arrays in SaC can be defined by using the `reshape` operation. `reshape` expects two operands, a shape vector and a data vector, both of which are specified as comma separated lists of numbers enclosed in square brackets.

To get started, try the following program:

Listing 2.1: Defining Arrays I

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    print( reshape( [5], [1,2,3,4,5]));
7    print( reshape( [3,2], [1,2,3,4,5,6]));
8    print( reshape( [3,2,1], [1,2,3,4,5,6]));
9    return(0);
10 }
```

It prints three arrays:

- an array of dimensionality 1 with 5 elements `[1,2,3,4,5]`;
- an array of dimensionality 2 with 3 rows and 2 columns, and
- a 3-dimensional array with 3 elements in the leftmost axis, 2 elements in the middle axis, and one element in the rightmost axis.

Note here, that the function **print** can be applied to arbitrary arrays. Besides printing its argument's dimensionality and shape, i.e., its shape vector, a more intuitive representation of the array's data vector is shown. However, as the terminal allows for 2 dimensions only, arrays of higher dimensionality are interpreted as nestings of 2-dimensional arrays. Therefore, the 3-dimensional array is printed as a 2-dimensional array of vectors.

Exercise 1 *In all these examples, the product of the shape vector matches the length of the data vector. What do you expect to happen, if this condition does not hold?*

For reasons of convenience, we use the following terminology:

scalar always denotes an array of dimensionality 0,

vector always denotes an array of dimensionality 1, and

matrix always denotes an array of dimensionality 2.

As **all** arrays can be defined in terms of `reshape`, the following program as well is perfectly legal:

Listing 2.2: Defining Arrays II

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    print( reshape( [1], [1]));
7    print( reshape( [], [1]));
8    return(0);
9  }
```

The most interesting aspect of this program is the array defined in line 7. The empty shape vector makes it a 0-dimensional array, i.e., a scalar. The data vector carries the scalar's value, which, in this example, is 1.

Exercise 2 *The arguments of* `reshape` *are vectors, i.e., arrays of dimensionality 1. Can they be specified by* `reshape` *expressions themselves?*

The `reshape` notation is relatively clumsy, in particular, when being used for scalars. Therefore, scalars and vectors can alternatively be specified by shortcut notations as well.

For experimenting with these, try the following:

Listing 2.3: Shortcut Notation for Arrays

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    print( 1);
7    print( [1,2,3,4,5]);
8    print( [ [1,2], [3,4], [5,6]]);
9    print( genarray( [4,3,2], 1));
10   print( genarray( [4,3], [1,2]));
11   return(0);
12 }
```

From these examples, we can see that scalars can be used in the same way as in most programming languages, and that the notation used for the parameters of `reshape` in the examples above in fact is a standard abbreviation of SaC. The example in line 8 shows that nestings of arrays are implicitly eliminated, i.e., the resulting array is identical to

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

For this reason, array nestings in SaC always have to be **homogeneous**, i.e., the shapes of the inner array components have to have identical shapes.

Furthermore, a new function is introduced: `genarray`. It expects two arguments, a shape vector that defines the shape of the result and a default element to be inserted at each position of the result. As shown in the example of line 10, the individual array elements can be non-scalar arrays as well which implicitly extends the dimensionality of the result array.

Exercise 3 *Given the language constructs introduced so far, can you define an array that would print as*

```
Dimension:  3
Shape    : <  5,  2,  2>
< 0  0 > < 0  0 >
< 1  0 > < 0  0 >
< 0  1 > < 0  0 >
< 0  0 > < 1  0 >
< 0  0 > < 0  1 >
```

, but whose definition does not contain the letter `1` *more than once?*

2.1.2 Arrays and Variables

This section explains why in SaC arrays are data and not containers for values as found in most other languages.

So far, all examples where expression based, i.e., we did not use any variables. Traditionally, there are two different ways of introducing variables. In conventional (imperative) languages such as C, variables denote memory locations which hold values that may change during computation. In functional languages, similar to mathematics, variables are considered placeholders for values. As a consequence, a variable's value can never change. Although this difference may seem rather subtle at first glance, it has quite some effects when operations on large data structures (in our case: large arrays) are involved.

Let's have a look at an example:

Listing 2.4: Variables as Placeholders

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    a = [1,2,3,4];
7    print( a);
8
9    b = modarray( a, [0], 9);
10   print( b);
11
12   return(0);
13 }
```

The function `modarray` expects three arguments: an array to be "modified", an index that indicates the exact position within the array to be "modified", and the value that is to be inserted at the specified position. As we would expect, the resulting array `b` is almost identical to `a`, only the very first element has changed into `9`.

Note here, that indexing in SaC always starts with index `0`!

Referring to the container / placeholder discussion, the crucial question is: does the variable `a` denote a container, whose value is changed by `modarray`? If this would be the case, `a` and `b` would share the same container, and every access to `a` after line 9 would yield `[9,2,3,4]`. If `a` in fact is a placeholder, it will always denote the array `[1,2,3,4]`, no matter what functions have obtained `a` as an argument.

To answer this question, you may simply shift the first call of `print` two lines down. As you can see, in SaC, variables are indeed placeholders.

> **A note for efficiency freaks:**
> *You may wonder whether this implies that* `modarray` *always copies the entire array. In fact, it only copies* `a` *if the placeholder property would be violated otherwise.*

As a result of this placeholder property, it is guaranteed that no function call can affect the value of its arguments. In other words, the underlying concept

guarantees, that all functions are "pure". Although this helps in avoiding nasty errors due to non-intended side-effects, it sometimes seems to be an annoyance to always invent new variable names, in particular, if arrays are to be modified successively.

To cope with this problem, in SaC, variables do have a so-called **scope**, i.e., each variable definition is associated to a well-defined portion of program code where its definition is valid. In a sequence of variable definitions, the scope of a variable starts with the left hand side of its definition and either reaches down to the end of the function, or, provided at least one further definition of a variable with the same name exists, to the right hand side of the next such definition. This measure allows us to reuse variable names. A slight modification of our example demonstrates the effect of variable scopes in SaC:

Listing 2.5: Variable Scopes

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    a = [1,2,3,4];
7
8    b = modarray( a, [0], 9);
9    print( a);
10   a = b;
11   print( a);
12
13   a = modarray( a, [1], 8);
14   print(a);
15
16   return(0);
17 }
```

Here, the use of `a` on the right hand side of line 9 still refers to the definition of line 6, whereas the use in line 11 refers to the definition in line 10.

The definition in line 13 shows, how variable scopes can be used to specify code that looks very much "imperative". However, you should always keep in mind, that in SaC, the placeholder property **always** holds!

Exercise 4 *What result do you expect from the following SaC program?*

Listing 2.6: Scope Exercise

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    a = [1,2,3,4];
7    b = [a,a];
8
9    a = modarray( modarray( a, [0], 0), [1], 0);
10   b = modarray( b, [0], a);
11   print(b);
12
13   return(0);
14 }
```

2.2 Lesson: Shape-Invariant Programming

The term **shape-invariant programming** refers to a programming style where all array operations are defined in terms of operations that are applied to entire arrays rather than to individual array elements. In order to realize such a programming style, it is an essential prerequisite to be able to apply functions to arbitrarily shaped arguments. In SaC, this is the case.

All built-in array operations as well as all array operations supplied by the standard library can be applied to arguments of arbitrary shapes. However, most of the operations that require more than one argument do have certain restrictions wrt. which array shapes can be combined as valid arguments. If an operation is applied to a set of arguments whose shapes constitute an illegal combination, usually, a type error message is given. In cases where this cannot be decided statically, the compiler may accept such a kind of domain error and produce a runtime error instead.

This lesson consists of three parts: The section on **Standard Array Operations** introduces the most important standard array operations provided by actual SaC compiler release. The next section explains **Axis Control Notation**, a powerful but simple way of manipulating the focus of array operations wrt. individual axes of argument arrays. With it, the basic operations often can easily be combined into rather complex operations as demonstrated in the section on **Putting it all Together**.

2.2.1 Standard Array Operations

In the sequel, several toy examples demonstrate the functionality of the most basic array operations of the actual SaC release. Their design is inspired by those available in APL. However, several aspects - in particular wrt. special case treatment in APL - have been adjusted to allow for a more runtime favorable compilation in SaC.

> **A note for language design freaks:**
> *You may have your own ideas on what primitive array operations should be available and how the precise semantics of these should look alike. Therefore, it should be mentioned here, that* **all(!)** *array operations introduced in the remainder of this section are not hard-wired into the language, but they are defined in the module* `Array` *from the standard library. This is to say that the advanced SaC programmer may write his own set of standard array operations. Consult the trail on defining primitive array operations in chapter* **??** *for detailed information on how to do so.*

The individual parts of this section are all organized according to the following scheme: first, a semi-formal introduction to the functionality of individual operations is given. Then, several examples shed some more light on their exact semantics by varying the argument shape constellations and by exploring "border-line cases“ wrt. to domain restrictions if these do exist.

Basic Operations

The most basic operations are very close to the model of arrays in SaC. They comprise functions for inspecting, creating, and modifying an array's shape and content. If not stated otherwise, they are applicable to arbitrarily shaped arguments of built-in element type. Note here, that some of them have been introduced in earlier lessons already.

dim(a) returns the (scalar) dimensionality of the argument array `a`.
Domain restrictions: none.

shape(a) returns the shape vector of the argument array `a`.
Domain restrictions: none.

a[iv] constitutes a short-cut notation for `sel( iv, a)`. It selects the array element of `a` at index position `iv`. As `a` may be of any shape, the index position is given as an **index vector**. The dimensionality of the result is identical to the dimensionality of `a` minus the length of `iv`. Accordingly, its shape is derived from the last components of the shape of `a`.
Domain restrictions:

- `dim(iv)` $=$ `1`
- `shape(iv)[[0]]` $\leq$ `dim(a)`
- $\forall i \in \{0, ...,$`shape(iv)[[0]]`$\}$: `iv[[`i`]]` $<$ `shape(a)[[`i`]]`.

a[iv]=expr; is a short-cut notation for an assignment of the form `a = modarray( a, iv, expr);`. The result of this application is a new array which is almost identical to `a`. Only the element (subarray) at index position `iv` is different, it is replaced by `expr`.
Domain restrictions:

- `dim(iv)` $=$ `1`
- `shape(iv)[[0]]` $\leq$ `dim(a)`
- $\forall i \in \{0, ...,$`shape(iv)[[0]]`$\}$: `iv[[`i`]]` $<$ `shape(a)[[`i`]]`
- `shape(expr)` $=$ `shape(a[iv])`.

reshape(shp, expr) computes an array with shape vector *shp* and data vector identical to that of *expr*.
Domain restrictions:

- `dim( shp)` $=$ `1`
- $\prod_{i=0}^{\texttt{shape(shp)[[0]]}-1}$ `shp[[`i`]]` $=$ $\prod_{i=0}^{\texttt{dim(expr)}-1}$ `shape(expr)[[`i`]]`.

genarray(shp, expr) generates an array of shape `shp`, whose elements are all identical to `expr`.
Domain restrictions: `dim(shp)` $=$ `1`.

Although these operations are fairly self-explaining or known from the Lesson 2.1 on **Arrays as Data**, let us have a look at a few example applications:

Listing 2.7: Basic Operations

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [1,2,3,4,5,6,7,8,9,10,11,12];
7
8    mat = reshape( [3,4], vect);
9    print( mat);
10
11   print( mat[[1,1]]);
12   print( mat[[2]]);
13   print( mat[[]]);
14
15   mat[[1,1]] = 0;
16   print( mat);
17   mat[[2]] = [0,0,0,0];
18   print( mat);
19   mat[[]] = genarray( [3,4], 0);
20   print( mat);
21
22   empty_vect = [];
23   print( empty_vect);
24   empty_mat = reshape( [22,0], empty_vect);
25   print( empty_mat);
26   print( dim( empty_mat));
27   print( shape( empty_mat));
28
29   return(0);
30 }
```

The different selections in lines 11-13 show how the dimensionality of the selected element increases as the length of the index vector decreases. If the index vector degenerates into an empty vector, the entire array is selected. Similarly, the applications of modarray in lines 15-19 demonstrate the successive replacement of individual elements, rows, or the entire array.

Lines 22-27 are ment to draw the reader's attention to the fact that there exists an unlimited number of distinct empty arrays in SaC!

Exercise 5 *Assuming* `mat` *to be defined as in the previous example, what results do you expect from the following expressions:*

- `reshape([3,0,5], [])[[]]`*?*
- `reshape([3,0,5], [])[[1]]`*?*
- `reshape([3,0,5], [])[[1,0]]`*?*
- `mat[reshape([2,0], [])]`*?*

Element-wise Extensions

Most of the operations that can be found as standard operations on scalars in other languages are applicable to entire arrays in SaC. Their semantics are simply element-wise extensions of the well-known scalar operations. The binary operations in general do have some domain restrictions. First, the element types of both arguments do have to be identical. Furthermore, either one of the arguments has to be a scalar value, or both arguments have to have identical shapes. In the former case, the scalar value is combined with each element of the (potentially non-scalar) second argument, which dictates the shape of the result. The latter case results in an array of the same shape as both arguments are, with the values being computed from the corresponding elements of the argument arrays.

In detail, the following operations are available:

arithmetic operations including addition (`e1 + e2`), subtraction (`e1 - e2`), negation (`- e1`), multiplication (`e1 * e2`), and division (`e1 / e2`). Furthermore, division on rational numbers (`e1 div e2`) and a modulo operation (`e1 % e2`) are supported.
Domain restrictions: the element types have to be numerical.

logical operations including conjunction (`e1 && e2`), disjunction (`e1 || e2`), and negation (`! e1`).
Domain restrictions: the element types have to be boolean.

relational operations including less-than (`e1 < e2`), less-or-equal (`e1 <= e2`), equal (`e1 == e2`), not-equal (`e1 != e2`), greater-or-equal (`e1 >= e2`), and greater-than (`e1 > e2`).
Domain restrictions: none.

max(e1, e2), min(e1, e2) compute the element-wise maximum and minimum, respectively.
Domain restrictions: none.

where(p, e1, e2) is an element-wise extension of a conditional. It expects `p`, `e1`, and `e2` either to have identical shapes or to be scalar. If at least one of the three arrays is non-scalar, that shape serves as shape of the result, whose values are taken from `e1` or `e2` depending on the value(s) of `p`.
Domain restrictions:

- the element type of `p` has to be boolean
- the element types of `e1` and `e2` have to be identical
- $\exists$ `shp`: ((`shape(p)` = `shp` $\vee$ `shape(p)` = `[]`) $\wedge$ (`shape(e1)` = `shp` $\vee$ `shape(e1)` = `[]`) $\wedge$ (`shape(e2)` = `shp` $\vee$ `shape(e2)` = `[]`)).

Again, these operations are fairly self-explanatory. Nevertheless, we present a few examples:

Listing 2.8: Elementwise Extensions

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [1,2,3,4,5,6,7,8,9];
7
8    mat = [ vect, vect+10, vect+20];
9    print( mat);
10
11   mat2 = where( (mat % 2) == 0, mat, -mat);
12   print(mat2);
13
14   print( max( mat2, 0));
15
16   return(0);
17 }
```

The most interesting part of this example is the definition of the matrix `mat2` in line 11. The even numbers from the matrix `mat` are taken as they are, whereas the odd numbers are negated. Note here, that all subexpressions in predicate position are in fact non-scalar arrays: `(mat % 2)` denotes a matrix of zeros and ones and `(mat % 2) == 0` denotes a matrix of boolean values.

Exercise 6 *What results do you expect from the following expressions:*

- `min( reshape([3,0,5], []), 42)`*?*
- `reshape([3,0,5], []) + reshape([3,0,5], [])`*?*
- `reshape([1,1], [1]) + reshape([1], [1])`*?*

Restructuring Operations

The operations to be introduced here do not compute new values at all. Instead, they are ment to create slightly differently structured arrays from existing ones. Therefore, they are applicable to arrays of all built-in element types.

take(sv, a) takes as many elements from the array `a` as indicated by the shape vector `sv`. Each element of `sv` corresponds to one axis of `a` starting from the leftmost one. For positive components of `sv`, the elements are taken from the "beginning", i.e., starting with index 0, otherwise they are taken from the "end" including the maximum legal index of the corresponding axis. All axes of `a` where there exists no corresponding element in `sv` are taken entirely.
Domain restrictions:

- `dim(sv)` $= 1$
- `shape(sv)[[0]]` $\leq$ `dim(a)`
- $\forall i \in \{0, ...,$`shape(sv)[[0]]`$\}$: $|$`sv[[`i`]]`$| \leq$ `shape(a)[[`i`]]`

drop(sv, a) drops as many elements from the array `a` as indicated by the shape vector `sv`. Each element of `sv` corresponds to one axis of `a` starting from the leftmost one. For positive components of `sv`, the elements are dropped from the "beginning", i.e., starting with index 0, otherwise they are dropped from the "end" starting from the maximum legal index of the corresponding axis. All axes of `a` where there exists no corresponding element in `sv` are left untouched.
Domain restrictions:

- `dim(sv)` $= 1$
- `shape(sv)[[0]]` $\leq$ `dim(a)`
- $\forall i \in \{0, ...,$`shape(sv)[[0]]`$\}$: |`sv[[`i`]]`| $\leq$ `shape(a)[[`i`]]`

tile(sv, ov, a) takes a tile of shape `sv` from `a` starting at the index specified by the offset vector `ov`. For axes where no values of `sv` or `ov` are specified these are assumed to be identical to the extent of `a` or 0, respectively.
Domain restrictions:

- `dim(sv)` $=$ `dim(ov)` $= 1$
- `shape(sv)[[0]]` $\leq$ `dim(a)`
- `shape(ov)[[0]]` $\leq$ `dim(a)`
- $\forall i \in \{0, ...,$`shape(ov)[[0]]`$\}$: `ov[[`i`]]` $\leq$ `shape(a)[[`i`]]`
- $\forall i \in \{0, ...,$`min( shape(ov)[[0]], shape(sv)[[0]])`$\}$:
  `ov[[`i`]] + sv[[`i`]]` $\leq$ `shape(a)[[`i`]]`
- $\forall i \in \{$`min( shape(ov)[[0]], shape(sv)[[0]]), ..., shape(sv)[[0]]`$\}$:
  `sv[[`i`]]` $\leq$ `shape(a)[[`i`]]`

e1 ++ e2 concatenates arrays `e1` and `e2` wrt. the leftmost axis. As in SaC all arrays are homogeneous, this requires all but the leftmost axis to be of identical extent.
Domain restrictions:

- `e1` and `e2` have to be of identical element type
- `drop( 1, shape(e1))` = `drop( 1, shape(e2))`.

rotate(ov, a) rotates the array `a` wrt. those axes specified by the offset vector `ov`. Starting from the leftmost axis, the elements of `ov` specify by how many positions the elements are rotated towards increasing indices (positive values) or towards decreasing indices (negative values). *Domain restrictions:*

- `dim(ov)` $= 1$
- `shape(ov)[[0]]` $\leq$ `dim(a)`

shift(ov, expr, a) shifts the array `a` wrt. those axes specified by the offset vector `ov`. The element positions that become "void" are filled by the (scalar) default element `expr`. Again, depending on the sign of the values of `ov` the elements are either shifted towards increasing or decreasing indices.
Domain restrictions:

- `dim(ov)` $= 1$
- `shape(ov)[[0]]` $\leq$ `dim(a)`
- `shape(expr)[[0]]` $=$ `[]`

A few examples:

Listing 2.9: Restructuring Operations

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [1,2,3,4,5,6,7,8,9];
7
8    mat = [ vect, vect+10, vect+20];
9    print( mat);
10
11   print( take( [2,-2], mat) );
12   print( take( [2], mat) );
13   print( take( [], mat) );
14
15   print( take( [0], mat) );
16   print( take( [2, 0], mat) );
17   print( take( [2], reshape( [3,0,5], [])) );
18
19   print( drop( [0, -1], mat) );
20
21   print( mat ++ mat);
22
23   print( rotate( [-1, 42], mat) );
24   print( rotate( [ 1], mat) );
25
26   print( shift( [0, -2], 0, mat) );
27   print( shift( [0, -22], 0, mat) );
28   print( shift( [1], 0, mat) );
29
30   return(0);
31 }
```

The applications of `take` in lines 11-13 demonstrate, how the dimensionality of `mat` remains unaffected by the length of the first argument. Only the shape of the result and the "side" from which the elements are taken is defined by it.

The applications in lines 15-17 demonstrate how empty arrays are dealt with in the individual argument positions. In particular from the example in line 17 it can be seen how well the concept of having an unlimited number of different empty arrays available fits nicely into the overall framework.

The remaining examples are rather straightforward. the only aspect of interest here may be the "overflows" in the rotation and shift parameters in lines 23 and 27, respectively.

Exercise 7 *Which of the following expressions can be reformulated in terms of* `take`*,* `++`*, and the basic operations defined in the previous parts?*

- `drop( v, a)`*?*
- `tile( v, o, a)`*?*

- `shift( [n], e, a)`*?*
- `shift( [m,n], e, a)`*?*
- `rotate( [n], a)`*?*
- `rotate( [m,n], a)`*?*

Can we define the general versions of `shift` *and* `rotate` *as well?*

Reduction Operations

The library for the standard array operations also contains a set of functions that fold all (scalar) elements of an array into a single one. The most common ones of these are described here.

sum(a) sums up all elements of the array `a`. If `a` is an empty array, `0` is returned.
Domain restrictions: the element type has to be numerical.

prod(a) multiplies all elements of the array `a`. If `a` is an empty array, `1` is returned.
Domain restrictions: the element type has to be numerical.

all(a) yields `true`, iff all elements of `a` are `true`. If `a` is an empty array, `true` is returned.
Domain restrictions: the element type has to be boolean.

any(a) yields `true`, iff at least one element of `a` is `true`. If `a` is an empty array, `false` is returned.
Domain restrictions: the element type has to be boolean.

maxval(a) computes the maximum value of `a`. If `a` is an empty array, the minimal number of the according element type is returned.
Domain restrictions: the element type has to be numerical.

minval(a) computes the minimum value of `a`. If `a` is an empty array, the maximal number of the according element type is returned.
Domain restrictions: the element type has to be numerical.

A few examples:

Listing 2.10: Reduction Operations

```
1 use StdIO: all;
2 use Array: all;
3
4 int main()
5 {
6   vect = [1,2,3,4,5,6,7,8,9];
7
8   mat = [ vect, vect+10, vect+20];
9
```

```
10    print( sum( mat) );
11    print( prod( vect) );
12    print( all( mat >= 1) );
13    print( any( mat > 1) );
14    print( maxval( mat) );
15    print( minval( mat) );
16
17    return(0);
18  }
```

Most of these examples, again, are fairly self explanatory. However, you may get an idea of the specificational advantages of shape-invariant programming when having a closer look at lines 12 and 13. They demonstrate the rather intuitive style of program specifications that results from it.

Exercise 8 *All operations introduced in this part apply to* **all** *elements of the array they are applied to. Given they array operations introduced so far, can you specify row-wise or column-wise summations for matrices? Try to specify these operations for a 2 by 3 matrix first.*

2.2.2 Axis Control Notation

As can be seen from the Exercise 8, without further language support, it is rather difficult to apply an array operation to certain axes of an array only. This section introduces two language constructs of SaC which, when taken together, can be used to that effect. While **Generalized Selections** are convenient for separating individual axes of an array, **Set Notations** allow to recombine such axes into a result array after applying arbitrary operations to them. However, as the two constructs in principle are orthogonal, we introduce them separately before showing how they can be combined into an instrument for **Axis Control**.

Generalized Selections

The selection operation introduced in Section 2.2.1 does not only allow scalar elements but entire subarrays of an array to be selected. However, the selection of (non-scalar) subarrays always assumes the given indices to refer to the leftmost axes, i.e., all elements wrt. the rightmost axes are actually selected. Sofar, a selection of arbitrary axes is not possible. As an example consider the selection of rows and columns of a matrix. While the former can be done easily, the latter requires the array to be transposed first.

To avoid clumsy notations, SaC provides special syntactical support for selecting arbitrary subarrays called **Generalized Selections**. The basic idea is to indicate the axes whose elements are to be selected entirely by using dot-symbols instead of numerical values within the index vectors of a selection operation.

Note here, that vectors containing dot-symbols are not first class citizens of the language, i.e., they can exclusively be specified within selection operations directly!

There are two kinds of dot-symbols, single-dots which refer to a single axis and triple-dots which refer to as many axes as they are left unspecified within

a selection. In order to avoid ambiguities, a maximum of one triple-dot symbol per selection expression is allowed.

A few examples:

Listing 2.11: Generalized Selections

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [1,2,3,4,5,6,7,8,9];
7
8    mat = [ vect, vect+10, vect+20];
9    print( mat);
10
11   print( mat[[1]] );
12   print( mat[[1,.]] );
13   print( mat[[1,...]] );
14
15   print( mat[[.,1]] );
16   print( mat[[...,1]] );
17
18   print( mat[[1,...,1]] );
19
20   arr3d = [ mat, mat];
21   print( arr3d);
22
23   print( arr3d[[.,1]] );
24   print( arr3d[[...,1]] );
25
26   return(0);
27 }
```

The examples in lines 11-13 demonstrate different versions for selecting the second row of the matrix `mat`. However, as the rightmost axis is to be selected, a dot-free version (cf. line 11) suffices for this task. The selection of the second column of `mat` is shown in lines 15 and 16. Line 18 demonstrates that the triple-dot notation can also be successfully applied if no axis can be matched at all.

The difference between the single-dot and the triple-dot notation is shown in lines 23 and 24. While the selection in line 23 is identical to `arr3d[[.,1,.]]`, the one in line 24 is identical to `arr3d[[.,.,1]]`.

Only in cases where the number of single-dots plus the number of numerical indices exceeds the number of axes available, an error message will be generated.

Exercise 9 *How can a selection of all elements of* `mat` *be specified using generalized selections? Try to find all 9 possible solutions!*

Exercise 10 *Referring to Exercise 5, can this new notation be used for selecting "over" empty axis? For example, can you specify a selection vector* $< vec >$*, so that* `reshape([3,0,5], [])[`$< vec >$`] == reshape( [3,0], [])` *holds?*

Set Notation

The means for composing arrays that have been described so far are rather restricted. Apart from element-wise definitions all other operations treat all elements uniformly. As a consequence, it is difficult to define arrays whose elements differ depending on their position within the array. The so-called **set notation** facilitates such position dependent array definitions. Essentially, it consists of a mapping from index vectors to elements, taking the general form

{ <*index_vector*> `->` <*expression*> }

where <*index_vector*> either is a variable or a vector of variables and <*expression*> is a SaC expression that refers to the index vector or its components and defines the individual array elements. The range of indices this mapping operation is applied to usually can be determined by the expression given and, thus, it is not specified explicitly.

> **A note for language design freaks:**
> *You may wonder why we restrict the expressiveness of the set notation by relying on compiler driven range detection rather than an explicit range specification. The reason for this decision is the observation that in many situations the capabilities of the set notation suffice whereas an explicit specification of ranges would obfuscate the code.*
> *Furthermore, as you will see in chapter* **??**, SaC *provides a more versatile language construct for defining arrays. However, the expressiveness of that construct comes for quite some specificational overhead.*

Let us have a look at some examples:

Listing 2.12: Basic Set Notation

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [0,1,2,3,4,5,6,7,8,9];
7
8    mat = { [i] -> vect[[i]]*10+vect };
9    print( mat);
10
11   mat_inc = { iv -> mat[iv] + 1};
12   print( mat_inc);
13
14   mat_trans = { [i,j] -> mat[[j,i]] };
15   print( mat_trans);
16
17   mat_diag = { [i,j] -> where( i==j , mat[[i,j]] , 0) };
18   print( mat_diag);
19
20   return(0);
21 }
```

The set notation in line 8 defines a vector whose components at position `[i]` are vectors that are computed from adding a multiple of `10` to the vector `vect`. The legal range of `i` is derived from the selection `vect[[i]]` yielding in fact

a matrix with shape `[10,10]`. An explicit element-wise increment operation is specified in line 11. Since the operation does not need to refer to individual axes a variable `iv` is used for the entire index vector rather than having variables for individual index components. Line 14 shows how the matrix can be transposed, and line 17 changes all non-diagonal elements to `0`.

Exercise 11 *Which of these operations can be expressed in terms of the array operations defined so far?*

Exercise 12 *What results do you expect if* `mat` *is an empty matrix, e.g.,* `reshape( [10,0], [])`*?*

As we can see from the set notation in line 8, non-scalar expressions within the set notation per default constitute the inner axes of the result array. This can be changed by using `.`-symbols for indicating those axes that should constitute the result axis.

A few examples:

Listing 2.13: Advanced Set Notation

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [0,1,2,3];
7
8    mat = { [.,i] -> vect[[i]]*4+vect };
9    print( mat);
10
11   arr3d = { [i] -> vect[[i]]*16 + mat};
12   print( arr3d);
13
14   arr3d = { [.,.,i] -> vect[[i]]*16 + mat};
15   print( arr3d);
16
17   arr3d = { [.,i] -> vect[[i]]*16 + mat};
18   print( arr3d);
19
20   return(0);
21 }
```

These examples show how the result can be directed into any axes of the result. As can be seen in line 17, the axes of the expressions can even be put into non-adjacent axes of the result.

Exercise 13 *The `.`-symbol in the set notation allows us to direct a computation to any axes of the result. This is identical to first putting the result into the innermost axes and then transposing the result. Can you come up with a general scheme that translates set notations containing `.`-symbols into set notations that do without?*

Axis Control

Although generalized selections and the set notation per se can be useful their real potential shows when they are used in combination. Together, they constitute means to control the axes a given operation is applied to.

The basic idea is to use generalized selections to extract the axes of interest, apply the desired operation to the extracted subarrays and then recombine the results to the overall array.

For example, we can easily now sum up the individual rows or columns of a matrix:

Listing 2.14: Axis Control: sum

```
use StdIO: all;
use Array: all;

int main()
{
  vect = [0,1,2,3,4,5,6,7,8,9];

  mat = { [.,i] -> vect[[i]]*10+vect };
  print( mat);

  sum_rows = { [i] -> sum( mat[[i]]) };
  print( sum_rows);

  sum_cols = { [i] -> sum( mat[[.,i]]) };
  print( sum_cols);

  return(0);
}
```

Reduction operations, in general, are prone to axis control, as they often need to be applied to one or several particular axis rather than an entire array. Other popular examples are the maximum (`max`) and minimum (`min`) operations:

Listing 2.15: Axis Control: max

```
use StdIO: all;
use Array: all;

int main()
{
  vect = [0,1,2,3];

  arr3d = { [i,j] -> vect[[i]]*4 + vect[[j]]*16 + vect };
  print( arr3d);

  max_inner_vects = { [i,j] -> max( arr3d[[i,j]]) };
  print( max_inner_vects);

  max_inner_arrays = { [i] -> max( arr3d[[i]]) };
  print( max_inner_arrays);

  max_outer_arrays = { [i] -> max( arr3d[[.,.,i]]) };
  print( max_outer_arrays);

  return(0);
}
```

In line 8, we directly generate a 3 dimensional array from the vector `vect`. Lines 11, 14, and 17 compute maxima within different slices of that array. `max_inner_vects` is a matrix containing the maxima within the innermost vectors, i.e., the 3-dimensional array is considered a matrix of vectors whose maximum values are computed. For `max_inner_arrays`, the array is considered a vector of matrices; it contains the maximum values of these subarrays. The last example demonstrates, that outer dimensions can be considered for reduction as well.

Further demand for axis control arises in the context of array operations that are dedicated to one fixed axis (usually the outermost one) and that need to be applied to another one. Examples for this situation are the concatenation operation (`++`) and `reverse`:

Listing 2.16: Axis Control: ++, reverse

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [0,1,2,3];
7
8    arr3d = { [i,j] -> vect[[i]]*4 + vect[[j]]*16 + vect };
9    print( arr3d);
10
11   print( arr3d ++ arr3d);
12   print( { [i] -> arr3d[[i]] ++ arr3d[[i]] });
13   print( { [i,j] -> arr3d[[i,j]] ++ arr3d[[i,j]] });
14
15   print( reverse( arr3d));
16   print( { [i] -> reverse( arr3d[[i]]) });
17   print( { [i,j] -> reverse( arr3d[[i,j]]) });
18
19   return(0);
20 }
```

Line 11 shows a standard application of the concatenation of two arrays. It affects the outermost axis only, resulting in an array of shape `[ 8, 0, 0]`. The two subsequent lines show, how to apply concatenation to other axis. Essentially, the selections on the right hand sides select the subexpressions to be catenated and the surrounding set notation glues the concatenated subarrays back together again.

The examples in lines 15-17 show the same exercise for a the operation `reverse` which reverses the order of the elements within an array wrt. the outermost axis.

Exercise 14 *The operation* `take` *is defined in a way that ensures inner axes to be taken completely in case the take vector does not provide enough entities for all axes. How can* `take` *be applied to an array so that the outermost axis remains untouched and the selections are applied to inner axes, starting at the second one? (You may assume, that the take vector has fewer elements than the array axes!) Can you specify a term that - according to a take vector of length 1 - takes from the innermost axis only?*

Exercise 15 *Can you merge two vectors of identical length element-wise? Extend your solution in a way that permits merging n-dimensional arrays on the outermost axis.*

2.2.3 Putting it all Together

The array operations presented so far constitute a substantial subset of the functionality that is provided by array programming languages such as APL. When orchestrated properly, these suffice to express rather complex array operations very concisely. In the sequel, we present two examples that make use of this combined expressive power: matrix product and relaxation.

Matrix Product

The matrix product of two matrices A and B (denoted by $A \odot B$) is defined as follows:

Provided A has as many columns as B has rows, the result of $A \odot B$ has as many rows as A and as many columns as B. Each element $(A \odot B)_{i,j}$ is defined as the scalar product of the i-th row of A and the j-th column of B, i.e., we have $(A \odot B)_{i,j} = \sum_k A_{i,k} * B_{k,j}$.

This definition can directly be translated into the following SaC code:

Listing 2.17: Matrix Product

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    id = [ [1d, 0d, 0d], [0d, 1d, 0d], [0d, 0d, 1d]];
7
8    vect = [1d, 2d, 3d, 4d];
9    mat = [ vect, vect+4d, vect+8d];
10   print( mat);
11
12   res = { [i,j] -> sum( id[[i,.]] * mat[[.,j]]) };
13   print( res);
14
15   return(0);
16 }
```

After defining two matrices `id` and `mat` in lines 6 and 8, respectively, the matrix product `id` $\odot$ `mat` is specified in line 12. `id[[i,.]]` selects the `i`-th row of `id` and `mat[[.,j]]` refers to the `j`-th column of `mat`. The index ranges for `i` and `j` are deduced from the accesses into `id` and `mat`, respectively. A variable k as used in the mathematical specification is not required as we can make use of the array operations `*` and `sum`.

Relaxation

Numerical approximations to the solution of partial differential equations are often made by applying so-called **relaxation methods**. These require large

arrays to be iteratively modified by so-called **stencil operations** until a certain convergence criterion is met. Fig. 2.1 illustrates such a stencil operation. A

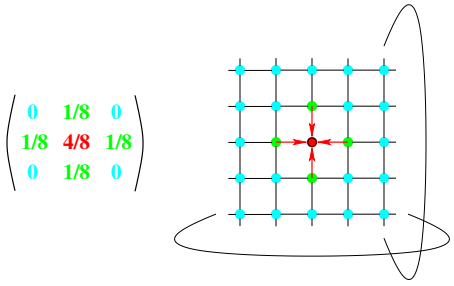

Figure 2.1: A 5-point-stencil relaxation with cyclic boundaries

stencil operation re-computes all elements of an array by computing a weighted sum of all neighbor elements. The weights that are used solely depend on the positions relative to the element to be computed rather than the position in the result array. Therefore, we can conveniently specify these weights by a single matrix of weights as shown on the left side of Fig. 2.1.

In this example, only 4 direct neighbor elements and the old value itself are taken into account for computing a new value. (Hence its name: **5-point-stencil operation**). As can be seen from the weights, a new value is computed from old ones by adding an eight-th each of the values of the upper, lower, left, and right neighbors to half of the old value.

As demonstrated on the right side of Fig. 2.1 our example assumes so-called **cyclic boundary conditions**. This means that the missing neighbor elements at the boundaries of the matrix are taken from the opposite sides as indicated by the elliptic curves.

In the sequel, we concentrate on the specification of a single relaxation step, i.e., on one re-computation of the entire array. This can be specified as a single line of SaC code:

Listing 2.18: Relaxation with Cyclic Boundaries

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    weights = [ [0d, 1d, 0d], [1d, 4d, 1d], [ 0d, 1d, 0d]] / 8d;
7
8    vect = [1d, 2d, 3d, 4d];
9    mat = [ vect, vect+4d, vect+8d, vect+12d];
10   print( mat);
11
12   mat = { [i,j] -> sum( { iv -> weights[iv] * rotate( iv-1, mat)}[[...,i,j]] ) };
13   print( mat);
14
15   return(0);
16 }
```

Line 6 defines the array of weights as given on the left side of Fig. 2.1. Our example array is initialized in lines 8-9. The relaxation step is specified in line 12. At its core, all elements are re-computed by operations on the entire array rather than individual elements. This is achieved by applying `rotate` for each legal index position `iv` into the array of weights `weights`. Since the expression { `iv -> weights[iv] * rotate( iv-1, mat)`} computes a 3 by 3 array of matrices (!) the reduction operation `sum` needs to be directed towards the outer two axes of that expression only. This is achieved through axis control using a selection index `[...,i,j]` within a set notation over `i` and `j`.

Exercise 16 *Another variant of relaxation problems requires the boundary elements to have a fixed value. Can you modify the above solution in a way that causes all boundary elements to be 0? [***Hint:** *You may consider the boundary elements to actualy be located* **outside** *the matrix]*