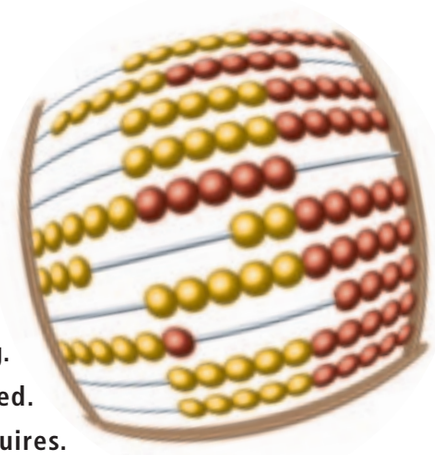
Scheme as a teaching device
LEARNING CURVE

CHRISTIAN WAGENKNECHT & RONALD SCHAFFHIRT



We've already covered quite a lot of ground in just a few Scheme articles, dealing with fairly advanced procedures such as first-class objects, macros and GUI programming. Other programming languages however, are a little more involved. Scheme is unsurpassed in the level of abstraction it requires.

Due to very simple syntax on the one hand and semantically powerful language elements on the other, Scheme is excellently suited to formulating abstract concepts. For this reason Scheme is one of the favourite *didactic* vehicles in the teaching of students: "Represent it with Scheme and play around with the defined functions, then you will better understand what it's all about."

In this article we would like to introduce two examples of this didactic approach. The first concerns *calculation with infinite objects*, and the second deals with the area of *Web programming*. In both cases we will express our ideas in Scheme and use interactive Scheme programming as a means rather than an end.

We have used Chez Scheme for the implementation. This can be downloaded free from *http://www.scheme.com/* as Petite Chez Scheme. The conditions of application are described there.

Calculating Infinity – Representing infinite objects with finite memory

The statement that computers can only handle finite objects is normally taken for granted. For instance, rational numbers are generally implemented as fixed-point numbers. However large the mantissa, a recurring decimal fraction like $0.\overline{3}$ will be ruthlessly truncated from a certain decimal digit onwards. The dots or overscore in the finite notation indicate that the threes continue indefinitely. Due to truncation, not even rational numbers, let alone real ones, are represented adequately by computers. Consequently, you end up working with approximate values rather than the actual quantities. This means you are limiting yourself to machine numbers, for which some of the mathematical laws that apply to rational numbers have no effect. Clearly, the capacity for infinite objects is limited by the always finite memory.

Or perhaps not. Looking at the equivalent fraction $1/3$ instead of $0.\overline{3}$, it offers a *finite*, "dotfree" representation of the same number. This observation leads to the idea of representing rational and even real numbers by the *method used for their creation*. In the case of 1/3: 'Divide 1 by 3'.

A warning: The aim is not to *perform* an algorithmic operation, but rather the definition of a number through the (where necessary continuous) operation employed in its creation. But how is a calculation with numbers represented in this way supposed to work? How much is, for example, $\sqrt{2}/\sqrt{3}$? Do the respective operations need to be added to each other in this case?

Data type "stream"

In the following text we will be introducing an abstract data type "stream" (a conceptually infinite list or sequence), which is characterised as follows:

A *stream* is a pair whose first member is any Scheme object (such as a numeral), but not a stream and whose second member is a stream. We access the first member with stream-car and the second with stream-cdr.

The reader is strongly advised to refrain from questions regarding the implementation of language elements for streams at this point. Let's just assume that everything we require is available (or built in).

To create an actual stream we are using a constructor, stream-cons. This expects two arguments, the two members of the pair to be created as mentioned above.

Let's look at two example streams:

```
(define integers
 (letrec
  ((integer-stream-maker
    (lambda (from)
     (stream-cons from (integer-stream-maker (+ from 1))))))
  (integer-stream-maker 0)))
```

Now we would like to look at 10 elements of this numeric sequence.

```
> (stream-print integers 10)
```

displays

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

on the screen.

The second example concerns the Fibonacci number sequence, i.e.

```
1, 1, 2, 3, 5, 8, 13, 21, 34, ...
```

The n-th Fibonacci number is defined by the following simple rule: the first two Fibonacci numbers are 1. Each subsequent number is the sum of the two previous ones. This is easily written as a Scheme procedure.

```
(define fib
 (lambda (n)
  (if (< n 2)
     1
     (+ (fib (- n 1)) (fib (- n 2))))))
```

Calculate (fib 29) and see how long it takes your computer to do this.

Now let's define the (infinite) sequence of Fibonacci numbers using

```
(define fib-stream
 (letrec
  ((fib-stream-maker
    (lambda (from)
     (stream-cons (fib from) (fib-stream-maker (+ from 1))))))
  (fib-stream-maker 0)))
```

and then display the first 30 members of this sequence:

```
> (stream-print fib-stream 30)
```

As expected, this takes even longer than (fib 29) above. The result is:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,➥
 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, ➥
121393, 196418, 317811, 514229, 832040
```

If we now evaluate the same expression again (stream-print fib-stream 30), the result is returned without any noticeable use of computing time. This is a welcome efficiency advantage caused by the fact that elements of a stream are not re-evaluated once they have been calculated. Instead, *Scheme 'remembers' already calculated stream elements*.

Evaluation concepts

Before we continue to work with streams we should look at the reason for this odd evaluation behaviour. By default, a Scheme expression in the format

```
(Operator Operand_1 Operand_2 ... Operand_n)
```

is evaluated according to the following rule: evaluate all elements of the list and then apply the operator to the operands. The sequence of evaluation for the individual parts of the expression is not fixed. All that matters is that the operands are evaluated *before* the (evaluated) operator is applied to them. This is called *applicative order evaluation*, which has efficiency advantages compared to *normal order evaluation* (from left to right), as demonstrated by the following example:

```
((lambda (x) (x (x 5))) ((lambda (w) w) fib))
= ((lambda (x) (x (x 5))) fib)
= (fib (fib 5))
= (fib 8)
= 34
```

Normal order evaluation would first apply the expression ((lambda (w) w) fib) to the left hand side (twice, once for each x) and then continue to reduce the resulting expression.

```
((lambda (x) (x (x 5))) ((lambda (w) w) fib))
= (((lambda (w) w) fib) (((lambda (w) w) fib) 5))
= (fib (fib 5))
= (fib 8)
= 34
```

This obviously leads to a loss of efficiency through multiple evaluation of one and the same part of the expression. In our example ((lambda (w) w) fib)) is evaluated twice.

This advantage of applicative order evaluation has led to the strategy being built into all common Scheme systems. The efficiency benefits are rated more highly than correctness when reducing particularly unusual Scheme expressions, like

```
((lambda (x) 3)((lambda (x) (x x))(lambda (x) (x x))))
```

which are relatively rare. The (applicative order) evaluation will not reach a result, even though the normal order evaluation terminates:

```
((lambda (x) 3)((lambda (x) (x x))(lambda (x) (x x))))
```

Each x in expression 3 is replaced by ((lambda (x) (x x))(lambda (x) (x x))). Since expression 3 does not contain any x at all, there is nothing to do.

However, there is an efficiency problem with the standard Scheme evaluation itself that is tackled in other functional languages (such as Gofer) by a change in strategy. The value of the expression

```
((lambda (w x y z) w) 1 (fib 29) (fib 29) (fib 29))
```

is 1. The three elaborate calculations of (fib 29) are completely unnecessary.

This leads to the idea of only ever performing evaluations when the value in the expression in question is actually required. Sometimes - as in the example above - it is never needed. This strategy is known as *call by need*. In contrast to the *eager evaluation* implemented in Scheme as standard, call by need is a *delayed evaluation (lazy evaluation)*.

The realisation of delayed evaluation in Scheme simply requires two (built-in) language elements, delay, to create a *delayed expression* (called promise), and force, to force the evaluation of a delayed expression.

Compare the following two versions of crazy

```
(define crazy
 (lambda (w x y z)
   x))

(define crazy-lazy
 (lambda (w x y z)
  (force x)))



> (crazy (fib 29)(fib 29)(fib 29)(fib 29))
832040

> (crazy-lazy
    (delay (fib 29)) (delay (fib 29))
    (delay (fib 29)) (delay (fib 29)))
832040
```

and try to interpret the different computing times.

There is another advantage to delayed evaluation. An expression that is forced to evaluate using force is not re-evaluated, as we've already seen above when we were working with streams. A small experiment will emphasise this message.

```
> (define x (delay (fib 29)))
> (force x)
832040
> (force x)
832040
```

The calculation of (fib 29) for the first force takes noticeably longer than for the second one.

Language elements		
	list	**stream**
structure	(head . <list>)	(head . <stream>)
first element	(car <list>)	(define stream-car car)
remaining list	(cdr <list>)	(define stream-cdr (lambda (stm) (force (cdr stm))))
constructor	(cons x <list>)	stream-cons
empty object	'()	'()
Predicates	list?	(define stream? pair?)
	null?	(define stream-null? null?)

Implementing language elements for streams

Everything is now ready for implementing the language elements used above to work with streams. Since streams are closely related to (always finite) lists, we shall use an analogy between the two.

The definition of stream-cons poses a problem for us: The approach

```
(define
 stream-cons
 (lambda (head tail)
  ...))
```

is not much use, because when calling stream-cons, tail would also be evaluated, instead of being delayed. We resolve this problem with the help of a macro.

```
(define-syntax stream-cons
 (syntax-rules ()
  ((stream-cons head tail) (cons head (delay tail)))))
```

Another useful language element for streams that we have already used above is

```
(define stream-print
 (lambda (stm n)
  (cond
   ((= n 0) (printf "...~%"))
   (else
    (printf "~s, "(stream-car stm))
    (stream-print (stream-cdr stm) (- n 1))))))
```

stream-print is used to return the first n elements of a sequence. If you are only interested in the n-th member, then

```
(define stream-n-print
 (lambda (stm n)
  (if (= n 0)
   (printf "~s~%" (stream-car stm))
   (stream-n-print (stream-cdr stm) (- n 1)))))
```

will come in handy.

Stream representations of real numbers

For irrational numbers $x = \sqrt{a}$ with natural $a \geq 2$, interval nesting (l_i, r_i) with $l_i \leq x \leq r_i$ for $i \in N$ and rational interval boundaries l_i, r_i can be specified.

For example, (l_i, r_i) can be constructed using the split-half method.

$(l_{i+1}, r_{i+1}) = \left(l_i, \frac{l_i + r_i}{2}\right)$, if $a < \left(\frac{l_i + r_i}{2}\right)^2$ else $(l_{i+1}, r_{i+1}) = \left(\frac{l_i + r_i}{2}, r_i\right)$.

$l_0 = 1$ and $r_0 = a$ apply for the initial values

The procedure itvs deals with interval nesting.

```
(define itvs
 (lambda (a)
  (letrec
   ((interval-stream-maker
     (lambda (left right)
      (stream-cons
       (cons left right)
       (let ((middle (/ (+ left right) 2)))
        (if (< a (* middle middle))
            (interval-stream-maker left middle)
            (interval-stream-maker middle right)))))))
   (interval-stream-maker 1.0 a))))
```

Therefore (infinite!) interval nesting (itvs 2) defines the real number $\sqrt{2}$

```
> (define sqr2 (itvs 2))
> (stream-print sqr2 10)
(1.0 . 2), (1.0 . 1.5), (1.25 . 1.5),
(1.375 . 1.5), (1.375 . 1.4375), (1.40625 . 1.4375),
(1.40625 . 1.421875), (1.4140625 . 1.421875),
(1.4140625 . 1.41796875), (1.4140625 . 1.416015625)
```

That makes sqr2= $\sqrt{2}$, even though we only get approximate values for a sufficiently large i when looking at the rational intervals (with stream-print or stream-n-print). Terminating/non-terminating continued fraction expansions are also used for defining rational/irrational numbers. There are other theoretical construction methods for defining real numbers, the important thing is that we can perform (exact!) calculations using the real number sqr2= $\sqrt{2}$ in Scheme.

Calculating with stream-represented numbers

We are going to demonstrate this for the division of the irrational numbers $x = \sqrt{k_1}$ and $y = \sqrt{k_2}$, with $a_i^l \leq x \leq a_i^r$ and $b_i^l \leq x \leq b_i^r$.

Because of $a_i^l \leq x \leq a_i^r$ and $\frac{1}{b_i^r} \leq \frac{1}{y} \leq \frac{1}{b_i^l}$, $\frac{a_i^l}{b_i^r} \leq \frac{x}{y} \leq \frac{a_i^r}{b_i^l}$

applies. It is also possible to show that interval lengths become as small as you want. These theoretical considerations lead directly to the following Scheme procedure, the result of which is a stream.

```
(define itv/
 (lambda (stm1 stm2)
  (stream-cons
   (let ((head1 (stream-car stm1))
         (head2 (stream-car stm2)))
    (cons (/ (car head1)(cdr head2))
          (/ (cdr head1)(car head2))))
   (itv/ (stream-cdr stm1)(stream-cdr stm2)))))
```

As an example, we are going to calculate $\sqrt{2}/\sqrt{3}$,

```
> (define sqr2/sqr3 (itv/ sqr2 (itvs 3)))
```

and look at the twentieth interval.

```
> (stream-n-print sqr2/sqr3 20)
(0.8164958693703516 . 0.8164973191071839)
```

As you can see, it is possible to calculate with the defined infinite objects. We shall leave the sum $\sqrt{2} + \sqrt{3}$ to the reader as an exercise. If you are going to attempt exponentiation of these numbers, please bear in mind that the interval boundaries do not remain rational.

The sieve of Eratosthenes

Two final examples from the field of number sequences will illustrate how powerful this concept is. The first one concerns the set of prime

numbers, which can be defined as an (infinite) number sequence:

Take a sequence of natural numbers starting with 2: 2, 3, 4, 5, 6, 7, ... = (stream-cdr (stream-cdr integers)) Filter out all multiples of whatever is the first member of the sequence: 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ... = (filter-out (lambda (x)(divides? 2 x) (stream-cdr (stream-cdr integers))) 2, 3, 5, 7, ~~9~~, 11, 13, ~~15~~, 17, ... = (filter-out (lambda (x)(divides? 3 x) (stream-cdr (stream-cdr integers))) etc. The remaining numbers are the prime numbers.

```
(define divides?
 (lambda (t n)
  (zero? (remainder n t))))

(define filter-out
 (lambda (praed stm)
  (if (praed (stream-car stm))
      (filter-out praed (stream-cdr stm))
      (stream-cons
       (stream-car stm)
       (filter-out praed (stream-cdr stm))))))

(define sieve
 (lambda (stm)
  (stream-cons
   (stream-car stm)
   (sieve
    (filter-out
     (lambda (x) (divides? (stream-car stm) x))
     (stream-cdr stm))))))
```

The prime number sequence results from

```
> (define prime-numbers (sieve (stream-cdr (➥
stream-cdr integers))))
```

Let's display the first 100 prime numbers on the screen:

```
> (stream-print prime-numbers 100)
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, ➥
61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, ➥
131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, ➥
193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, ➥
263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, ➥
337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, ➥
409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, ➥
479, 487, 491, 499, 503, 509, 521, 523, 541
```

Sequence of quotients of neighbouring Fibonacci numbers

If you calculate the quotient of any pair of neighbouring Fibonacci numbers, you will find that it seems to settle around a certain value, 1.61803.... Before calculating the respective limit, the quotient sequence helps to establish the hypotheses.

```
(define fibquot
 (lambda (stm)
  (stream-cons
   (/ (stream-car (stream-cdr stm))
      (fixnum->flonum (stream-car stm)))
   (fibquot (stream-cdr stm)))))
> (stream-print (fibquot fib-stream) 18)
1.0, 2.0, 1.5, 1.6666666666666667, 1.6, 1.625, ➥
1.6153846153846154, 1.619047619047619, 1.61764705288235294, ➥
1.6181818181818182, 1.6179775280898876, 1.6180555555555556, ➥
1.6180257510729614, 1.6180371352785146, 1.618032786885246, ➥
1.618034447821682, 1.6180338134001253, 1.618034055727554➥
```

Summary

Infinite objects can be defined, stored and processed using streams. The components of these objects (intervals, sequence members, etc.) always form a *potentially infinite* (e.g. set of all real numbers) or *countable set* (set of all integers). In contrast to uncountable sets, countable sets have as many elements as there are natural numbers.

The entire set of real numbers cannot be stored on a computer.

That has given us some pretty abstract insights. The Scheme procedures that we developed and implemented helped us to put the facts in concrete terms and (hopefully) contributed to their understanding.

HTML programming with Scheme

Anyone who has ever created or updated HTML documents without a WYSIWYG editor, by just working on the plain source text, will soon notice a certain lack of clarity, even within files they've written themselves, and begin to search behind the multitude of tags for the actual content they were meant to be updating. If there were an option of defining the content at the beginning of the document and of specifying the structure and formatting later on, this task would be much easier. But HTML is set in its ways and relatively inflexible.

The Scheme HyperText Generator allows the generation of HTML documents using Scheme. This involves what is more or less a new Scheme-based scripting language: HTSS (HyperText Scheme Source) provides powerful language elements from the Scheme world, together with the option of adapting and extending it. Scheme language elements make it possible, for example, to organise the document contents in an abstract way to begin with and to deal with the translation into a concrete structure and formatting afterwards. In this way, the source text remains clear and changes to the appearance of similar elements only have to be made once, because they are going to affect all of these elements.

As you will see, HTSS enables you to build up a document description language which bears no relation to HTML, apart from the end result.

The idea of creating HTML documents with Scheme comes from Kurt Normark, who already presented his adaptation (LAML) in the issue 8. In contrast to his approach of providing easy-to-use language elements for document generation, SHTG is aimed at the creative application of Scheme programming knowledge. In this article we are going to show how higher functions can be implemented with existing procedures, in order to demonstrate how powerful HTSS is.

SHTG

HTML is a language that describes the structure and formatting of content with tags and Cascading StyleSheets (CSS). The tags should be correctly nested and ideally result in an HTML tree. This hierarchical tag structure will be represented by similarly nested Scheme procedures from which SHTG (Scheme HyperText Generator) generates an HTML document. However, unlike HTML tags, Scheme procedures can be extended and redefined, which makes them considerably more powerful.

The basic principle is to assign a Scheme procedure of the same name to each HTML tag. (with the exception of the tag <map>, which becomes html-map, because a procedure called map already exists in Scheme.) These procedures can now be used to generate HTML documents directly, or as a basis for implementing more powerful language elements, which constitute the main strengths of this approach. Furthermore, tags can be adapted to language-specific requirements.

HTSS

The language used in connection with SHTG is called HyperText Scheme Source, or HTSS for short. To begin with, let's look at a simple example that only uses standard procedures:

HTSS	HTML
(html	<html>
(head	<title>window-title</title></head>
(title "window-title"))	<body
(body	bgcolor="#000000"
	text="#ffffff">
`(bgcolor "#000000")	Content</body></html>
`(text "#ffffff")	
"Content"))	

That is more or less a 1:1 translation of a tiny HTSS document into a HTML file, which naturally doesn't show the real strengths of HTSS. You will already notice, however, that with HTSS there is no need to worry about closing tags - when a bracket is closed, so is the tag, although there are exceptions where no closing tag exists (such as <img ...>), but HTSS takes these into account.

Of course, tags can also contain attributes. While this happens in the form of </tag name=value> in HTML, specification in HTSS is handled as follows: (tag `(name value) ...), where the value type can be symbol, number or string. The character before the list is not a quote but a backtick (the key to the left of 1), and is the short form of (quasiquote <list>). This allows us to evaluate variables within the list, which must be identified by a leading comma. Instead of the three dots other attributes could, of course, follow with the same format. Once all attributes have been specified, the content follows, which must always be a character string. If further functions are nested within the structure, their return value will still be a string. Let's have a look at an example:

```
(define colour1 "#ff0000")
(define colour2 "#0000ff")
(define text1 "... red text on black background")

(html
 (head (title "window"))
 (body
  `(bgcolor "#000000")
  `(text ,colour1)
  `(link ,colour2)
  (big text1)))
```

As you can see, the colour specifications (colour1 and colour2) in the attribute lists of the body tags are marked with a comma in order to evaluate them within the list, i.e. to insert the respective definitions from the beginning of the document. That allows ordered global formatting changes, if these variables are referenced several times in larger documents.

Defining your own functions

We are going to demonstrate the extendibility of HTSS through definition of new procedures with an example:

```
;================================
; define required procedures
;----------------
(define webpage (lambda (t . k)
  (html (head (title t))
        (apply body k))))
(define chapter (lambda (x)
  (string-append (p _) (h1 x))))
(define picture (lambda (source text . size)
  (let ((width (if (> (length size) 0)
                   (car  size)
                   #f))
        (height (cond ((= (length size) 1) (car  size))
                      ((> (length size) 1) (cadr size)))))
     (if width
        (img `(src   ,source) `(alt   ,text )
             `(width ,width) `(height ,height))
        (img `(src   ,source) `(alt   ,text ))))))
(define section h2)
(define heading h3)
(define text p)
;==========================
; and this is the source text
;-------------------
(webpage "My Homepage"
 (chapter "Introduction")

 (section "Who am I?")
 (heading "General")
 (text "My name is ... and I was born in ... ")
 (heading "Hobbies")
 (text "I am especially interested in ...")

 (section "What do I do?")
 (text "Within my ...")
 (picture "work.gif" "Me at work" 300 200)

 (chapter "My Projects")
 (section "Project 1: ...")
 (heading "Terms of Reference")
 (text "Drawing up ...")
 (heading "Preparation")
 (text "Before starting with ...")
)
```

First of all, the procedure webpage receives the name of the page as its first parameter (t), which is then displayed as the window title. All remaining entries are combined in the second parameter (k). The first value is used to call the function title, which in turn is located within head. Then body is applied to the actual body of the text with apply. An interim evaluation step will clarify this. First, we process the call (webpage "My Homepage" (chapter ...) ...)

```
(html
 (head
  (title "My Homepage"))
 (body
  (chapter ...)
  .
  .
  .))
```

In the next step ("Introduction" chapter) is evaluated. There should be a blank line before each new chapter. We achieve this using an empty paragraph. In HTML this would be   , whereas in HTSS it simply looks like this: (p _). Underscore is a predefined variable for a non-breaking space. After this blank paragraph we would now like the actual chapter heading with the largest possible font size. Each of the functions p and h1 returns a character string, just like all other tag procedures. The same is true of the procedure chapter, of course, so it must append the two strings with string-append first:

```
(html
 (head
  (title "My Homepage"))
 (body
  (string-append "<p>&amp;nbsp;</p>" "<h1>Introduction</h1>")
  .
  .
  .))
```

We shall not go into details regarding the function for inserting images. Suffice it to say that it receives a path for the image, alternative text and optional size information. The specification of only one number results in a square image of the appropriate size, while in the case of two values the first one is the width and the second one the height.

The remaining functions are self-explanatory since they correspond to their HTML counterparts.

It should be clear by now that it is possible to ignore HTML itself entirely, as long as the relevant procedures are loaded. You could develop your own page description language and use only that. All you need to do is implement the appropriate language elements once (or have them implemented for you). Should you ever find that one is missing, it is easy enough simply to fall back on the standard HTML tags, which are also still available.

Now we are going to demonstrate the special capabilities of HTSS by creating more powerful functions. Scheme offers a wide variety of opportunities that can be utilised in HTSS. Let's assume that we want to perform certain calculations with a sequence of numbers and to display the result as a table on the WWW. In the example we are calculating the Fibonacci numbers (2nd column) from i (1st column) and the quotients of two consecutive Fibonacci numbers (3rd column). You already know the Fibonacci sequence from the first part of the article when we were discussing streams.

```
;======================
; required procedures
;——————————-
(define fib
 (lambda (n)
  (if (< n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2))))))
(define line
 (lambda (i)
  (tr (th (number->string i))
      (td (number->string (fib i)))
      (td (number->string (/ (fib i) (fib (- i 1))))))))
(define table
 (lambda n
  (html
   (head (title "Table"))
   (body
    (div `(align center)
     (table `(border 1) `(cellpadding 5) `(cellspacing 0)
      (tr (th "i")
          (th "fib(i)")
          (th "fib(i) / fib(i-1)"))
      (apply string-append (map line n))))))))
;=============
; "source text"
;————-
(table 1 2 3 4 5 6 7 8 9 10)
```

The procedure map applies the specified function (line) to the parameter list n. The return value of map is a list containing strings. They are no use to us as a list, however, as we can only work with the strings themselves. By applying string-append to the entire list, all strings contained are concatenated and we receive the desired character chain with the results. This is done using apply. If we try this now, we will receive a table containing the values we were looking for.

By making a few amendments, it is also possible to avoid a static representation of the calculation formulas like above, but rather to include them in the call. That would already provide considerable functionality. Without referring to the actual Scheme procedures in detail, the call could then look like this:

```
(calculate
 '(
  (x         . "Nett Price")
  ((* x 0.07) . "VAT (7%)")
  ((* x 0.16) . "VAT (16%)")
  ((* x 1.07) . "Retail Price (7%)")
  ((* x 1.16) . "Retail Price (16%)")
  )
 '(15.78 29.80 14.26 39.03 45.12 19.25 33.45 22.34 25.56))
```

Libraries

The required procedures will be collected in an extendible library and re-used later. This is not possible for the CGI variant, since all of your definitions only apply to the current session. That is different for the installable version, where functions can be combined in files, which can be loaded as required. An example for such a library can also be found on the SHTG Web page.

Order and structure

In contrast to HTML, SHTG uses the bracket-structure that is typical for Scheme. On the one hand this takes care of controlling the structures, on the other it makes them difficult to handle without the support of a special editor, e.g. providing features such as highlighting matching brackets. Once you have about 20 brackets in a row, it becomes impossible to tell which tag is closed where. This is particularly annoying when you want to insert something. In HTML it is completely obvious from the closing tags. For this reason, SHTG offers the possibility of marking brackets appropriately, thereby considerably increasing the clarity of the source text. This is done by inserting the tag name as a symbol before the closing bracket. Symbols can be recognised by the apostrophe or backtick. We recommend the use of backticks, which we already know from the attribute lists. There is no syntax checking, however, so that even nonsensical names will be formally accepted at this point.

```
(html
 (head (title "Main_Window") `head)
 (body
  (div `(align center)
   (p "Table 1")
   (table `(border 1) `(cellpadding 5) `(cellspacing 0)
    (tr
     (td "field 1")
     (td "field 2"))
    (tr
     (td "field 3")
     (td "field 4"))
    `table)
   `div)
  `body)
 `html)
```

Without the identification of the last four brackets, the character chain 'field 4' would be followed by six closing brackets and it would not be immediately obvious which one belonged where.

Style sheets

In order to apply a certain style to a HTML file you can use the font tag or create a consistent document layout with style sheets. The W3 consortium recommends the latter, of course, the reason being that should the entire layout of a finished HTML file that has been formatted with font need to be amended, each individual font tag will have to be adjusted. If the same file had been formatted using a style

sheet, only this would need to be changed. Furthermore, several files can access the same CSS, which means that a certain consistency is apparent within a project that is not only aesthetically pleasing, but also points to a connection between the contents.

Nevertheless, there is still an orderly and standardised way of using the font tag in HTSS. Due to the ability to define new procedures, or tags, it is equally possible to attach certain style properties to these tags. Let's define a few new p-tags:

```
(define p-cn
 (lambda x
  (font `(face "courier new, courier, monospaced") (apply p x))))
(define p-blue
 (lambda x
  (font `(color "#0000ff") (apply p x))))
```

This variant is *almost* as consistent as style sheets and *almost* as easy to amend, but unfortunately not as powerful. Admittedly, it is possible to do a whole lot more with CSS. However, the aim of HTSS it is not to replace style sheets - you can use style sheets just as easily in pages created with SHTG as in any other HTML documents. A page created with SHTG contains its entire content and therefore also - hence the 'almost' - all its formatting. The only advantage becomes apparent when downloading such pages. If the CSS file for a downloaded Web page is missing or cannot be found, the page can sometimes look pretty bad compared to the original on the Web, something that won't happen with SHTG-generated documents. Why don't you experiment a bit for yourself!

Summary

You have learned about Scheme as a means of HTML programming with the capacity for wide-ranging extensions. With Scheme we were able to describe the structure of complex source texts in an orderly manner, without worrying about actually producing the final document. Due to the importance and popularity of Web programming this is also an approach to generate interest in Scheme and its possibilities.

With regard to teaching, SHTG also has the advantage of allowing students with Scheme knowledge to look at the paradigm of scripting languages from a structural perspective.

The authors

Professor Christian Wagenknecht teaches Theory of Information Technology, Programming Paradigms, Web Databases , Scientific Web Publishing, etc. in the Information Science and Technology faculty of the Technical University of Zittau/Gorlitz. For over 20 years he has been studying the use of non-imperative programming (Logo, Scheme, Prolog, Smalltalk, Java) from a didactic perspective. He bought the first SuSE Linux distribution as a set of diskettes in 1993.

Ronald Schaffhirt is one of Professor Wagenknecht's students and has developed Scheme for Web programming (CGI) and SHTG. The resulting material will be included in the course "Programming Paradigms" (section: scripting languages). He is currently in his fourth semester of studying Information Technology while continually developing SHTG on the side. ■