078schemeed.qxd 05.03.2001 10:47 Uhr Seite 78

Advanced Elements of Scheme Programming

FULLY FUNCTIONAL

FRIEDRICH DOMINICUS

After introducing the basic elements of Scheme programming in the previous issue, this article will focus on one of Scheme's essential advantages – functional programming.

A great advantage of languages like Lisp is the existence of higher order procedures. These are procedures that can either accept other procedures as parameters, in which procedures can be return values, or both. This ability is especially useful in cases such as the following example.

Procedures as parameters

The procedures *double-all* and *add-two* will be used to demonstrate how useful procedures are as parameters. The first procedure doubles every element of a list, while the second adds two to each element (Listing 1).

apply-to-all

```
(define (apply-to-all op a-list)
  (if (null? a-list)
     '()
     (cons (op (first a-list))
          (apply-to-all op (cdr a-
list))))
In double-all  op  looks like this:

 (* 2 x), which can be translated
directly into Scheme as:

 (define (double x) (* 2 x))
```

078schemeed.qxd 05.03.2001 10:47 Uhr Seite 79

The similarity between the two procedures is so remarkable that we may be tempted to ask whether there isn't some scope for generalisation – there is (at least in Scheme).

The only difference in each case is in the respective operation on the first element of the remaining list. After processing, the result is added to the returned list. Both examples can be simplified by using a procedure as a parameter. In order to do this, we will retain the implementation and parameterise the differences (in this case using the parameter *op*).
Let's use *double-all* and *apply-to-all* together with *double*:

```
(define a-list '(1 2 3))
(double-all a-list) -> '(2 4 6)
(apply-to-all double a-list) -> '(2 4 6)
```

The arrow -> represents 'results in' or 'evaluates to'. The two procedures return the same result, but the second version can be applied more universally.

What is remarkably easy here is impossible in some languages, or can only be reproduced with a lot of effort. The possibilities arising from higher order procedures should be so obvious even from this example that one has to ask why all languages do not offer them. To avoid having to give every

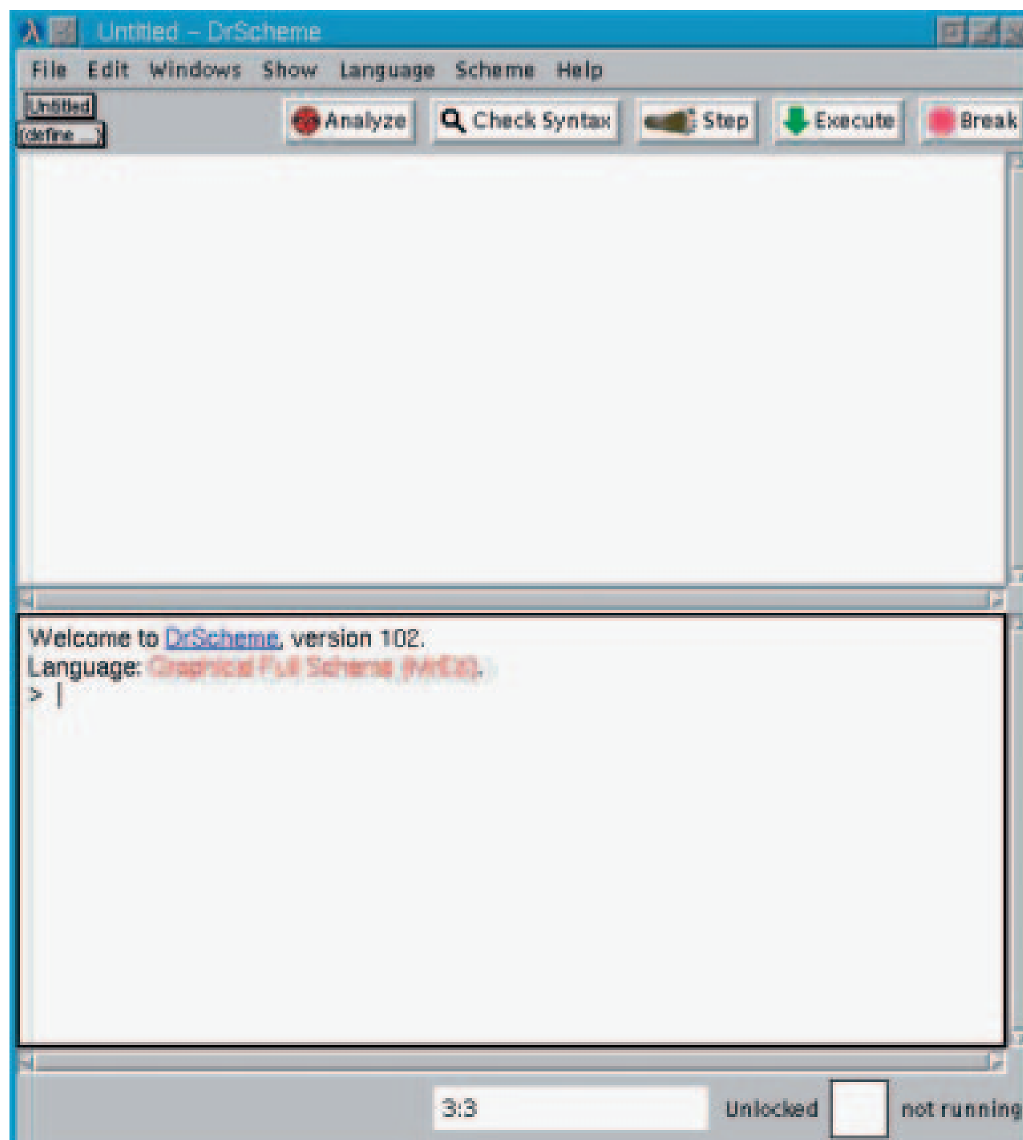
Listing 1: Two very similar procedures (double-all, add-two)

```
(define (double-all a-list)
  (if (null? a-list)
          '()
          (cons (* 2 (first a-list))
                (double-all (rest a-list)))))
(define (add-two a-list)
  (if (null? a-list)
          '()
          (cons (+ 2 (first a-list))
                (add-two (rest a-list)))))
```

Listing 2: Implementation of *filter*

```
(define (filter pred? a-list)
  (cond

    ((null? a-list) '())
    ((pred? (first a-list)) (cons (first a-list)
                              (filter pred? (rest a-list))))
    (else
     (filter pred? (rest a-list)))))
```



078schemeed.qxd 05.03.2001 10:48 Uhr Seite 80

little function its own name, anonymous procedures can be used in Scheme. Therefore *apply-to-all* can be simplified in the following way:

```
(apply-to-all (lambda (n) (* 2 n)) a-list)
```

The result would once again be '(2 4 6). *Lambda* is an principal element of Scheme programming and forms the basis of other elements which at first sight may seem to have nothing in common with the less flashy *lambda*.

Apply-to-all can be replaced by the standard Scheme procedure (*map*), with which our solution would look like this:

```
(map(lambda (n) (* 2 n)) a-list)
Naturally, double-all can also be re-impleme
nted using map.
(define (double-all al)
   (map (lambda (n) (* 2 n)) al))
```

Another useful procedure, (*filter*) extracts elements that satisfy certain conditions. *Filter* can be implemented as shown in Listing 2:

If you would like to filter out all positive elements from a list, you can use the following Scheme procedure:

```
(filter positive? a-list)
```

Map and *filter* can be combined as required. With

```
(filter positive? (map double a-list))
```

for instance, you can double all elements and then

extract the positive ones. In this case

```
(map double (filter positive? a-list))
```

would be equivalent and preferable for efficiency reasons.

Procedures as return values

Procedures and data are equivalent in Scheme. Procedures are permitted anywhere 'normal' variables can occur, within other methods, as parameters and also as return values. In technical terms, they are first-class objects. Here is an example:

```
(define (make-adder n)  (lambda (m)    (+ n
m))) (define add-5 (make-adder 5))
```

Add-5 is a procedure that adds 5 to its argument.

Local variables

As in other languages, it is possible to declare and use local variables in Scheme. Unlike most other languages, it is irrelevant whether the values of the local variables are data or procedures. Local variables are declared and initialised in a block that starts with a *let* or *let**. The difference between *let* and *let** is that you can refer to previous declarations in *let**. A notable aspect of the two *let* statements is that they are both simply macros around *lambda* with which we are already familiar. In principle, the following lines are equivalent:

```
(let ((i 1)) (display i) (newline))
((lambda (i) (display i) (newline)) 1)
```

In my opinion, the first line is clearer and easier to read. Please be careful not to get the 'named-let' in part 1 mixed up with this local variable declaration though. The valid format here is: *(let ((variable init))*. In Scheme, initialisation cannot be omitted.

The following example shows that you can also bind procedures to local variables in Scheme.

```
(let ((d (lambda (n) (* n 2))))
   (display (d 1)) (newline))
```

Let statements can be found not only within other procedures or additional *let* blocks, but also at the top level. In every case the declared variables are only visible until the end of the block in which they are defined.

Nesting not only occurs with *let* statements, but also for combinations of *let* and *define*. Whether you are going to use *(let ((i 1) (j 10)) (+ i j))* or *(let ((i 1)) (let ((j 10)) (+ i j)))* depends on the situation or your preference. Scheme does not impose any restrictions on you regarding the position of additional procedures or the introduction of local variables.

Lexical binding

Scheme is leading the way in this area, as far as the Lisp family is concerned. Lexical binding is again best illustrated with an example.

```
(define counter 50)
(define (inc-counter)
  (set! counter (+ 1 counter))
  counter)
(let ((counter 100))
  (display (inc-counter)) (newline)
  (display (inc-counter)) (newline))
```

Since Scheme uses lexical binding, the output will be 51 52. Even though *counter* in the *let* block overrides the global variable of the same name, this is not true for the procedure *inc-counter*. When this procedure is defined, only the global variable is known, and *inc-counter* therefore only has access to this. Procedures or variables defined within the block, however, would only have access to the overriding variable.

The standard for early versions of Lisp was another type of binding, called dynamic binding, which is still used by Emacs Lisp. DrScheme provides this type of binding in the form of a *fluid-let*:

```
(fluid-let ((counter 100))
  (display (inc-counter) (newline))
  (display (inc-counter) (newline)))
```

The output here would be: 100 101. Since the behaviour of lexical and dynamic bindings is different, you ought to be aware of the type of

078schemeed.qxd 05.03.2001 10:48 Uhr Seite 81

binding that is being used. Scheme normally only supports lexical binding, Common Lisp offers both types and the standard for Emacs Lisp is dynamic binding (with the possibility of emulating lexical binding).

Data encapsulation in Scheme

One interesting application for a combination of lexical binding and the use of higher order procedures in Scheme is controlled data access.

In Listing 3 you will notice the following innovation in the methodhead: . *args*. This parameter separated by a dot represents a list that comprises additional optional elements, similar to the ellipsis (three dots) in C.

The initialisation of the first variable *balance* is achieved through *(if (not (null? args)) (car args) 1000))*. If the list *args* exists its first element will be regarded as the initial value, otherwise this is set to 1000. The interest rate is similarly initialised with a value. Local variables cannot be accessed from outside, but they can be exported in the following way:

```
(lambda (message)
  (case message
    ((name) (lambda () name))
    ((balance) (lambda ()
             balance))
    ...
```

This is an anonymous procedure with a parameter, and it also becomes the method's returned value. The parameter is used to access a method defined in the case differentiation. For example, if *message* is '*name*, the following procedure is returned: *(lambda () name))*. As this is a procedure, it has to be called before you get the result that you are actually interested in. This call can be initiated in various ways, either by an object-oriented, or functional route. The OOP option could look like this:

```
define (send fun message . args)
            (apply (fun message) args))
```

The procedure *apply* is expecting a procedure and a list of elements to which this procedure will be applied. Should you for instance want to perform an addition, you could either use *(+ 1 2 3)* or *(apply + '(1 2 3))*. If we send the message *'name* to

Backtick vs. normal quote

```
(define al '(1 ,(+ 1 2) 3))
    -> (1 ,(+ 1 2) 3))

    (define a2 `(1 ,(+ 1 2) 3))
     -> (1 3 3)
```

These almost identical looking expressions show that a comma before an expression forces its evaluation. The official names for ' and ` are quote for ' and quasiquote for `. When expanding ,@ it should be noted that body is a list, while ,@ is used to divide and insert each individual list element.

Listing 3: Data encapsulation "la Scheme"

```
(define (account name . args)
  (let ((name name)
      (balance (if (not (null? args)) (car args) 1000))
      (interest-rate
       (if (and
            (not (null? args))
            (not (null? (cdr args))))
           (cadr args)
           0.06)))
    (lambda (message)
      (case message
        ((name) (lambda () name))
        ((balance) (lambda ()
                balance))
        ((deposit) (lambda (amount)
                (set! balance (+ balance amount))
                balance))
        ((withdraw) (lambda (amount)
                 (set! balance (- balance amount))
                 balance))
        ((interest-rate) (lambda ()
                      interest-rate))
        ((add-interest) (lambda ()
                     (set! balance
                           (* balance
                              (+ 1 interest-rate)))
                     balance))
        (else (lambda () "Method unknown"))))))
(define (send fun message . args)
  (apply (fun message) args))
```

078schemeed.qxd 05.03.2001 10:48 Uhr Seite 82

account, we receive the parameterless procedure mentioned above. Starting from the following definition: *(define acc1 (account "someone"))*, the substitution for the call *(send acc1 'name)* is *(apply (acc1 'name) '())*. If you prefer functional calls, use: *((acc1 'name))* (Please note the use of parentheses!)

This approach can form the basis for an OO system. If that system is to meet the expectations of object-oriented programming, then macros, another Scheme programming tool, would certainly help.

Macros

All Lisps offer the possibility to define new data and control structures, such as new case differentiations or new loops. In most Schemes there are different ways of defining macros. We will call them the 'old' and the 'R5RS' way. This example is a unilateral case differentiation.

The old way

```
 my-when Macro
(define-macro my-when
  (lambda (test . body)
  `(if ,test (begin ,@body))))
```

You can see how easy it is to implement such extensions in Scheme. Macros start with *define-macro*, receive a name (*my-when*), expect parameters (here *test . body* and a "suitable " substitution (*`(if ...)*). Let's take a closer look at the substitution:

In front of the *if*-parenthesis is a backtick (inverted single quote). This backtick can be compared to the double quote (") in shell programming. The already familiar single quote (') on the other hand has a similar function in Lisp and in shells. While in the first case certain characters obtain a special significance (, and ,@ in Scheme, the dollar sign for shells), in the second case this special significance is revoked. (See Backtick vs. Normal Quote)

After these explanations it would be interesting to see our newly-defined control flow element in action:

```
(my-when 1 0) -> 0
(define foo 2) (my-when (< foo 3)
                  "foo is less than 3")
                        -> "foo is less than 3"
```

Our macro seems to be expanding as required, which we can verify. In order to do this, *mzscheme* must be called from the command line, since this validation does not work in the Scheme development environment. Here is the expansion of our *my-when* macro:

```
 > (expand-defmacro '(my-when (< 1 2)"True"))
     ->  (#%if (< 1 2) (#%begin "True"))
```

The integration of new and old elements is seamless. This creates attractive application areas, such as the ability to configure your software using a language embedded in Scheme, without having to miss out on the advantages of a complete programming language. These advantages are inherent in Lisp, therefore the success of Emacs with Emacs Lisp or the decision of the FSF to use Guile, a Scheme implementation, as their scripting language, should come as no surprise.

The R5RS way

If you compare *define-syntax* to what you already know, you might detect analogies to case differentiation. With *define-syntax my-when-new* the macro is given a name (*my-when-new*).

```
my-when-new Macro
(define-syntax my-when-new
  (syntax-rules ()
    ((my-when-new test body ... )
                    (if test (begin body ...)➘
))))
```

After *syntax-rules* follow a number of patterns in which the macro can appear. In our case there is only one possibility: *(my-when ...)*. This pattern requires first the name of the macro, then a test *test* and afterwards something like *body ...* , where the ... represent any number of further expressions.

Should a *my-when-new* in the required form be found, then the subsequent transformation (*(if test (...* is performed. This substitution does not require special expansion characters. The names of the variables that appear in the template are allocated automatically.

To find out how the templates should look please refer to the R5RS report, which you can find in Dr Scheme's manuals. Go to the menu option *Help Helpdesk* and select *Help Desk Manuals Revised(5) ...* . Here is another example, from the report:

Info

Newsgroups: comp.lang.scheme, comp.lang.lisp, comp.emacs,comp.emacs.xemacs
Homepage of various Lisp projects; i.e. CMU version of Common Lisp: http://www.cons.org
Association of Lisp Users: http://www.alu.org
Everything about Scheme: http://www.schemers.org
Peter Norvig; Paradigms of Artificial Intelligence Programming, Case Studies in Common Lisp; Morgan Kaufman 1992
Harold Abelson and Gerald Jay Sussman with Julie Sussman; Structure and Interpretation of Computer Programs; McGraw-Hill 1996
Oliver Grillmeyer; Exploring Computer Science with Scheme; Springer, Berlin 1997
MzScheme Handbook: http://www.cs.rice.edu/CS/PLT/packages/pdf/mzscheme.pdf
Examples from this article: ftp://ftp.linux-magazin.de/pub/listings/magazin/2001/01/

```
my-and Macro
(define-syntax my-and
  (syntax-rules ()
    ((my-and) #t)
    ((my-and test) test)
    ((my-and test1 test2 ...)
        (if test1 (my-and test2 ...) #f))))
```

Pretty impressive stuff. You have implemented an *and* that works like the one in C, using only six lines of Scheme. The last pattern is of particular interest, it passes several tests to *my-and* . This is a recursive macro expansion, i.e. this macros is expanded until it contains no further macros. As soon as the expanded form results in a truth value of 'false', *#f* is returned.

A more substantial example

We will now look at the initial stages of structure definition in Scheme. Access will be restricted to access procedures. The usage is: *(define-my-record name Fields)*. We will then set up a creation method *make-name*, as well as methods for read- or write-access to the various fields. Read-access methods will start with *get-* , write-access ones with *set-* .

Let's look again at our example of Scheme data encapsulation. The planned extensions result in a partial automation.

```
(define-my-record account (name balance inte?
rest-rate))
     (let (( me (make-account ("Friedrich"
1000?
0.06))))
       ((me 'get-name)) -> "Friedrich"
       (send me 'get-balance) -> 1000
       ((me 'set-balance 2000))) -> 2000
```

In these extensions macros are combined with procedures. Here, as an example, the method for the generation of read-access:

```
(define (create-getter-clauses slots)
  (define (create-getter slot)
    `((,(prepend-string "get-" slot))
      (lambda () ,slot)))
  (map create-getter slots))
```

You can see how the different Scheme elements are joined together. An internal procedure is defined. The output of the call to *create-getter-clauses* corresponds exactly to the form required in *case*. The access method is generated dependent on the name. So when you use *name* in *define-my-record*, the symbol *get-name* is generated with *prepend-string "get-"name*. As this is required for all field names, the inner method is applied to all passed elements (*map*). By calling the procedure you have dynamically created Scheme code!

This code is built in to the following macro:

```
(define-macro define-my-record
  (lambda (name slots)
    `(define (,(make-new-record-name name)
              ,@slots)
       (lambda (message)
         (case message
           ,@(create-getter-clauses slots)
           ,@(create-setter-clauses slots)
           (else (error "Method unknown")))))))
```

The macro *define-my-record* expects a name, a list of elements (*slots*), generates a creation method from the first parameter, as well as a 'get-' and 'set-' method from each of the name elements in *slots*.

All elements in the example can be queried and amended. With a little more programming effort you can add read-only variables, type tests, additional methods and even inheritance.

Summary

This article shows how important procedures are in Scheme and how easily they can be applied. On the one hand, this reduces the reluctance to use procedures as parameters or returned values. On the other hand you have to abandon the idea of the separation of data structures and procedures familiar from other languages, in order to use Lisps to their best advantage. Such a change can only come from working with functional languages for a while.

One notable feature of Scheme is the generation of code by macros. This shows how a minimalist language with (warning: buzz word alert) orthogonal programming elements can provide enormous flexibility.

Outlook

In the next article I will be dealing with Standard Scheme as well as some useful libraries. Also, I will be introducing some 'specialities' of the different Schemes.

As always, I would welcome any comments, suggestions or criticism. You can contact me at *frido@q-software-solutions.com*. Or simply visit the Scheme/Lisp newsgroups. ■

The author

Friedrich Dominicus has worked with Linux since version 0.99pl13, trying pretty much every distribution at some point and finally ending up with Debian. He is currently developing new software using Eiffel (naturally on Linux) with a handful of other people. He has been a father since August 1998 and really enjoys every new day with his daughter. His hobbies are computers, sport and reading.