INTRODUCING SCHEME, THE SIMPLEST LISP LANGUAGE

FRIEDRICH DOMINICUS

In this series of articles we would like to introduce Lisp, one of the oldest language families, but one that is by no means ready for the scrap heap.

Lisp- 'sounds familiar? If your preferred text editor is Emacs, you will have come across at least one example of Lisp. Should you have had a more detailed look at Emacs Lisp you may have got the impression that this is a very substantial language. But don't be fooled, this substance is not intrinsic to the language itself, but results from a feature that is characteristic to all Lisp dialects. As Paul Graham puts it: "Lisp is a programmable programming language." The language core of all Lisp languages is relatively small, and that of Scheme is certainly the smallest. The Scheme standard (IEEE Standard, 1178-1990 R 1995) is probably the shortest for any language. Interestingly, that of another Lisp variant, Common Lisp, may well be one of the longest. It is therefore safe to say that the Lisp family offers something for everyone, a statement with which you will hopefully agree after reading this series of articles. As an introduction to Lisp programming I would like to start with its smallest exponent, Scheme.

Some Scheme history

There is an article on the Internet which illuminates the history of the Lisp family. The text comprises more than seventy pages, which is beyond the scope of an article like this. Here therefore a summary of the information about Scheme which can be found there.

Scheme was developed by Gerald Jay Sussman and Guy L. Steele in the mid-seventies as an implementation aimed at helping them to understand a theory by Carl Hewitt. Sussman and Steele are defining personalities in the development of the Lisp languages. Sussman is the co-author of one of the most highly acclaimed books on programming called Structure and Interpretation of Computer Programs, in which Scheme is used to illustrate different aspects. Reading this book is highly recommended. Steele is the author of the standard work on Common Lisp called Common Lisp, The Language.

This book was one of the starting points for the standardisation of Common Lisp and can therefore be found on the shelves of any Lisp programmer. It is not a textbook, but a reference manual and more than 1000 pages long. Its index alone is longer than the complete standard for Scheme.

Scheme was initially developed as a playground for programming experiments. Extensive experimentation lead to a number of different implementations of Scheme being developed, including some commercial versions. To get an overview, start here: *ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/misc/scheme_2.faq*.

The origin of the name Scheme has its own anecdote. Sussman and Steele were very pleased with this toy actor implementation and named it Schemer in the expectation that it might be developed into another AI language in the tradition of Planner and Conniver. However, the ITS operating system had a 6-character limitation on file names and so the name was truncated to simply Scheme and that name stuck.

Why Scheme?

Scheme, like Common Lisp, offers the opportunity of testing any programming paradigms. Scheme allows imperative, functional (one of its strengths) or, with extensions, object-oriented work. Many

Scheme systems contain an OO system, as does Common Lisp, which features one of the most flexible (CLOS). Scheme does not force its own scheme of things on you, which should please Linux fans especially.

Different Schemes

As previously stated, there are a number of implementations. The following is a short list of notable free Scheme implementations – obviously without any claim to completeness:

- MIT Scheme. Probably the most mature Scheme going (the current version is 7.5, and unlike Windows software they have been counting steadily upwards from one). Forms the basis of teaching at MIT, together with the book by Sussman mentioned above. Very substantial (probably the Scheme equivalent of Common Lisp) with interesting extensions and applications (object systems, graphics, an Emacs clone (Edwin) that uses Scheme as its extension language, etc.)
- Guile. The standard script language of the FSF. Scheme is used here especially as an extension language. There are even discussions underway to replace Emacs Lisp with Guile. The window manager SCWM uses Guile as its extension language. Guile is also very suitable for use as a Unix scripting language. As is often the case with FSF favourites, you either like Guile or avoid it
- Elk. An implementation developed especially for embedding in C or C++. The idea is certainly attractive: instead of creating a special language for each tool you use the full functionality offered by Scheme
- Scsh. My personal favourite when it comes to shell programming. In many cases where I once used shell scripts or Python scripts I now use Scsh. It is remarkable how cleanly the, sometimes unconventional, shell syntax has been converted to Scheme
- Kawa. A Scheme interpreter written in Java, which also translates into Java byte-code
- DrScheme or MzScheme. Almost as substantial as MIT Scheme and designed especially for teaching. DrScheme is a comprehensive Scheme development environment with very helpful extensions and the best documentation. I will therefore mainly be using DrScheme

Installation of DrScheme

- 1. Download the software from *http://www.cs.rice.edu/CS/PLT/*
- 2. *cd /usr/local/lib*
- 3. *tar -xvzf plt.i386-linux.tar.gz*
- 4. *cd plt*
- 5. Execute *./install*.
- 6. Set an environment variable *$PT_HOME* and add *$PT_HOME/bin* to your path.

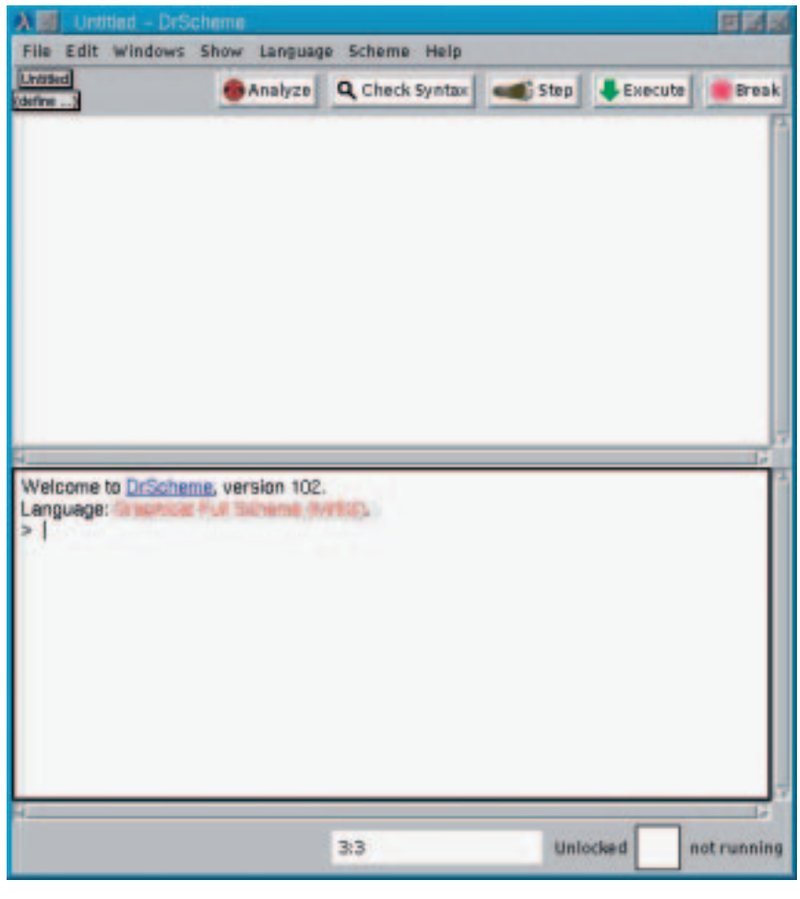
You can add the following to your .bashrc or .zshrc:

```
# for DrScheme
PT_HOME=/usr/local/lib/plt
export PATH=$PATH:$PT_HOME/bin
export MANPATH=$MANPATH:$PT_HOME/man
```

You should now be able to start DrScheme by typing *drscheme*. I would also recommend the installation of MrSpidey, a static debugger. However, this does not have to be done immediately, as DrScheme offers a more agreeable way of installing additional packages. When the Help Desk is open (menu Help->Help Desk) and you encounter a link which informs you that the required package has not yet been installed, you are given the chance to connect to the DrScheme Web site to install the desired software.

It is, of course, advisable not to install the software as root, and it is certainly also a good idea to put the packages onto your hard disk first before blindly accepting anything going. However, I have made things easy for myself by installing the entire software under an unprivileged account. Therefore I regarded the risk of possible data loss as not very high.

When you call DrScheme using *drscheme*, you will see the following opening screen: B:Figure 1: DrScheme Start Window The upper part of the window is an editor, where you can enter Scheme programs. These definitions can be saved if required. DrScheme uses the extension .ss, but .scm



is also common. When you click on execute the program text is transferred, and you can call the procedure you have defined above from the prompt in the lower part of the window. You could of course just go straight into the lower part and get started. On pressing return your entry is evaluated and a result is returned. You will probably be familiar with this process from your favourite scripting language.

Should the buttons Analyze and Step not be available to you then the corresponding packages have not been installed and you can upgrade via the Help Desk. Also helpful is Check Syntax, which lets you perform syntax checking. DrScheme offers several language levels; in figure 1 the most advanced level has been set. At this level you are able to use the graphics toolbox. A graphical user interface created in this way can be used anywhere there is an implementation of DrScheme. It is therefore possible to develop on Linux and, should someone insist, to run the programs written there on FreeBSD, Solaris, Macintosh and, last but least, Windows. As you can see, it is a real cross-platform development tool.

First steps

Try some entries in the lower part of the window. Please note that all Lisp languages use prefix notation. The sequence is always (procedure name parameter1 parameter2) Just give it a go:

```
> 1
1
> (+ 1 2)
3
> "Hello"
"Hello"
> (/ 1 3)
1/3
> (* (expt 2 32) 2)
8589934592
> a
reference to undefined identifier: a
> 'a
a
```

You can already notice some interesting features of Lisp languages. Numbers can be of any length and among the numeric data types offered by Scheme are fractions. The arithmetic should not surprise you, but possibly the reaction to the entry of *'a* might. The behaviour when entering a is likely to be familiar to you, a is seen as a variable, if nothing has been assigned to it you get an error message about undefined identifiers. The behaviour of 'a may surprise you. The ' represents (quote, that means the following variable is seen as a symbol. In this case the symbol is called a, i.e. if symbols are quoted they evaluate to themselves. Other elements that evaluate to themselves are numbers, characters and character strings.

How do you assign something to a variable in Scheme? By using define. Note that parentheses must be opened before using define, hence:

```
> (define a 1)
```

What happens to a is not shown, so we'll ask:

```
> a
1
```

Scheme is a dynamically typed language, therefore variables do not have to have their type explicitly declared. If you now enter:

```
> (define a "Hello")
> a
"Hello"
>
```

a becomes a string. The value of a variable can be determined using something called predicates. The names of the predicates are pretty obvious: string? for a string, number? for a number. The question mark is optional, but the convention is to append it:

```
> (string? a)
true
> (number? a)
false
> (define a 1)
> (string? a)
false
> (number? a)
true
>
```

Data types

What data types does Scheme provide? They are quickly listed: numbers (including ones of arbitrary size, fractions and floating point numbers), characters, strings, lists, fields, symbols, procedures and macros. You may ask yourself, is that all? Basically yes. DrScheme contains a construct for defining structures, but this is already an extension which is not included in the standard.

Talking of standards, the current standard is called *R5RS* and can be found easily using the Help Desk. Call the Help Desk, click on Manuals and select *Revised(5) Report on the Algorithmic Language Scheme*, then you can immerse yourself in the standard. Or better still: follow the link *http://www.schemers.org* to documentation about Scheme, print the report and treat yourself to a pleasant evening's reading.

First procedures

After this long introduction we should look at how procedures are defined and called. For this we will be using one of Scheme's central data structures, the list, and calculating the sum of all elements of a list. Our first attempt looks like this:

```
(define (my-sum-1 a-list)
(cond
((null? a-list) 0)
(else (+ (car a-list) (my-sum-1 (cdr a-list))
))))
```

define *(my-sum-1 a-list)* defines *my-sum-1* as the procedure name, with a parameter being expected. The name *a-list* implies that we are dealing with a list. This is how procedures are defined in Scheme, they are called in a similar manner, by enclosing the procedure and its parameters in parentheses. Let's have a look at the implementation: there we find *(cond*, this is the Scheme name for a multiple conditional or case differentiation. The case differentiation starts with a termination condition, the only one in this example, *((null a-list?*. *null?* checks whether the list is empty. If yes, then 0 is returned, otherwise the following is executed: *(+ (car a-list) (my-sum-1 (cdr a-list)))*.

A procedure call is always enclosed in parentheses, which means the following are procedure calls: +, *car*, *my-sum-1* and *cdr*. You can assign (almost) any name to a procedure in Scheme. There are certain conventions for different types of procedures, such as the question mark in predicates. What is being added? Something called *(car a-list)* and also something else. The significaance of *(car a-list)* an *(cdr* (pronounced could-er) originates in the history of Lisp and is a relic from its beginnings: *car* stands for contents of the address part of the register and *cdr* for contents of the decrement part of the register. To translate: *car* denotes the first element of a list, and *cdr* all elements of a list apart from the first one, that is to say, the rest of the list. Since these names are anything but mnemonic for anyone who is not a Lisp expert, we will instead define:

```
(define first car)
(define rest cdr)
```

Now *car* and first are synonyms, as are *cdr* and rest. This shows how simple Scheme is, it does not matter to *define* whether it is dealing with numbers, strings or procedures. The syntax is uniform and as you can see from the following example, procedures really are first-class citizens.

Let's re-write the procedure:

```
(define (my-sum-2 a-list)
(if  (null? a-list) 0
(+ (first a-list) (my-sum-2 (rest a-list)))))
```

In this case *(if* was used instead of *(cond*. This *(if* works differently from, for example, the one in Python. There is no explicit *else* keyword, instead everything is controlled by parentheses. The 0 after *(null? a-list)* is the "then" part and *(+* represents the else branch. Should there be several expressions in one branch you would need to use *(begin BLOCK)*. Let's have a closer look at the solution. As you can see it is recursive. This is usual in Scheme, and Scheme provides support for effective processing. The optimisation conditions are not right, however. You can check this in the following way: set the language level to Beginner and write the procedure into the upper window, along with a procedure call to it:

```
(define (my-sum-2 a-list)
(if  (null? a-list) 0
(+ (first a-list) (my-sum-2 (rest a-list)))))
(my-sum-2 (list 1 2 3))
```

Highlight the text and click on "Step". Now you can see step by step how the result is calculated. You will notice that all recursive calls are executed first and that results are only calculated upon return from the recursion. If you tried this with an extremely long list it could lead to a stack overflow. However, Scheme would not be Scheme if there was not a more elegant solution: tail recursion

Tail recursion

The Scheme standard explicitly demands the optimisation of tail recursion, and any compliant implementation of Scheme must support it. What is tail recursion? Tail-recursive programs do not have to execute the entire recursion, but instead calculate interim results for each step, which are then used in the next recursive call. The recursion does not build a stack frame for the recurring processes, instead for example, a jump instruction can be used. Let's put the procedure into a tail-recursive format:

```
(define (int-my-sum acc a-list)
(if (null? a-list) acc
(int-my-sum (+ (first a-list) acc) (rest a-li➥
st)))))
(define (my-sum-3 a-list)
(int-my-sum 0 a-list))
```

Normally you should define *(int-my-sum* internally as *(my-sum-3*. This is not accepted in the beginner levels of DrScheme, at a higher level however, the following is not a problem:

```
(define (my-sum-3 a-list)
(define (int-my-sum acc a-list)
(if (null? a-list) acc
(int-my-sum (+ (first a-list) acc) (rest a-li➥
st))))
(int-my-sum 0 a-list))
```

This shows how easy it is to shift Scheme code. Internal definitions can be applied at a higher level or a previously global procedure can be used internally. No special provisions have to be made in order to do this. Should you ever have problems with a long procedure you can simply export it bit by bit, test its parts separately and then put them back together once you have finished. This uniform syntax is a curse as well as a blessing. Without the support of an editor you are likely to despair of the parentheses. But on the other hand it shows how elegant the syntax can be. Please expand the procedure again and look at the step by step processing of the program. You will see that in this solution results are calculated at each step. The recursive call is the last one in this procedure and represents the tail. As stated above, in Scheme this

can (and must) be replaced with an iterative solution. Although I am not a Scheme guru, I would like to say a few word about programming style. Scheme programmers will certainly prefer the last version, they can regard this approach as a design pattern. *acc* stands for accumulator and is a sort of informal standard for a variable that accumulates values and is used to return a value at the end of the recursion. You don't need to feel tied to *acc*, but it pays in the long run to adopt customs that have developed over time as it makes things easier for the programmers that come after you.

Other solutions

We have already found three different solutions to the same problem. Now I would like to demonstrate some other elements of Scheme programming using different solutions:

- *letrec*
- named let
- iterative solutions

Solutions with *(letrec* are very similar to solutions with internal procedures. A *(letrec* solution looks like this:

```
(define (my-sum-with-letrec a-list)
(letrec ((my-int-sum
(lambda (acc a-list)
(if (null? a-list) acc
(my-int-sum (+ acc (first a-list)) (rest a-li?
st))))))
(my-int-sum 0 a-list)))
```

Which one you prefer is a matter of taste, to me the solution using an internal *define* is simply clearer. In the last solution you can see the first use of *(lambda*; this defines an anonymous procedure, in this case the name is set to *(my-int-sum* by using *letrec*. The recursive call occurs in the else part of the case differentiation and is used in the same way as in the solution with an internal *define*. I would like to come back to the solution using the internal *define*. I thought it would be clearer to start with the simplified format. Actually, a solution with an internal *define* would look like this:

```
(define my-sum-2-revealed
(lambda (a-list)
(define my-int-sum
(lambda (acc a-list)
(if  (null? a-list) 0
(+ (first a-list)
(my-sum-2-revealed (rest a-list))))))
(my-int-sum 0 a-list)))
```

There are situations in which you can only use this version. However, I find the previous definition clearer and use it as often as I can. Using anonymous procedures with lambda is the central mechanism for all calculations within Lisp. If you are interested you should have a look at the basics of the lambda calculus sometime.

Named let – The last explicitly recursive solution:

```
(define (my-sum-with-named-let a-list)
(let loop ((acc 0)
(al a-list))
(if (null? al) acc
(loop (+ acc (first al)) (rest al)))))
```

This is a very elegant solution in my opinion. Let's go through it step by step. You are already familiar with the procedure definition, but *(let loop* is new. This is what is called a named let, *loop* is simply a label or a name to which you want to refer. The introduction of loop variables is very readable: 0 is assigned to *acc* and the initial list to *al*. During a recursive call both variables change as follows: the value of the first list element is added to *acc* and within the list we move forward by one element.

"Iterative" solutions

The quotes around iterative are deliberate, because we are not necessarily dealing with real iterative solutions, although it does look that way. It is simply expected that these are quasi-iterative and therefore efficient solutions. Here is an iterative solution:

```
(define (my-sum-iterative-without-body a-list)
(do ((acc 0 (+ acc (first al)))
(al a-list (rest al)))
((null? al) acc)))
```

There's dense code for you. All the work is done in the loop header, there is no loop body. C programmers should get a feeling of déjà vu at this

Information

Structure and Interpretation of Computer Programs (SICP for short); Harold Abelson and Gerald Jay Sussman; MIT Press; second edition 1996
ON LISP, Advanced Techniques for Common Lisp; Paul Graham; Prentice Hall, 1994
Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp; Peter Norvig; Morgan Kauffmann publishers; 1991
Common Lisp the Language; Guy L. Steele jr.; Digital Press; second edition 1990
Writing GNU Emacs Extensions; Bob Glickstein; O'Reilly & Associates, 1997

Online:

ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/pubs/Evolution-of-Lisp.ps.gz
MIT Scheme:ftp://ftp.cs.indiana.edu:/pub/scheme-repository/imp
SCWM window manager: http://scwm.mit.edu/scwm/
Guile homepage: http://www.gnu.org/software/guile/guile.html
Elk: http://www-rn.informatik.uni-bremen.de/software/elk/
Scsh: http://www-swiss.ai.mit.edu/ftpdir/scsh/
Kawa: http://www.gnu.org/software/kawa/
DrScheme: http://www.cs.rice.edu/CS/PLT/
Scheme: http://www.schemers.org

point. So what is happening in detail? First, 0 is assigned to *acc*, then in each iteration the value of the first element of the remaining list is added to *acc*. The initial list is assigned to the variable *al* and then in each iteration the remaining list is assigned to it in turn. The termination condition is *((null? al)*. If this is true, the accumulator *acc* is returned.

Finally, I would like to show you a solution that is more similar to Pascal or Eiffel. In this case we move the update from the loop header to the loop body. The solution looks like this:

```
(define (my-sum-iterative-with-body a-list)
; now a bit more pascal-ish or eiffel-ish ;-)
(do ((acc 0)
(al a-list))
((null? al) acc)
; body starts here
(set! acc (+ acc (first al)))
;; attention set! is a destructive update,han➥
dle with care
(set! al (rest al))))
```

Not much has changed. The loop variables are initialised, but the update now takes place in the loop body rather than in the loop header. This is the first time we have come across a destructive operation. Up to now none of the procedures had any side-effects. In Scheme, procedures without side-effects are good form. You ought to try to adhere to this yourself. Destructive variants are indicated by the suffix *!*, to show that these methods should be used with caution. In principle *(set! variable new-value)* has exactly the same effect as an assignment. *(set! acc (+ acc (first al)))* is the equivalent of *acc = acc + first(al)* in Python.

Where do we go from here?

In these examples we have looked at the basic elements of Scheme programming. In subsequent articles on the subject of programming in Lisp languages I would like to introduce you to other elements, for example local variables, list operations, higher-order functions, scoping, OO programming and similar issues. I will also show you different Scheme implementations and discuss their strengths and applications.

As always I would welcome any comments, suggestions or questions. My email address is: Friedrich.Dominicus@inka.de ■