LispMe: An Implementation of Scheme for the Palm Pilot

Fred Bayer
Elisabethstr. 8
80796 Munich, Germany
fred@lispme.de

Abstract

LispMe is an implementation of the Scheme programming language for the Palm Pilot series of Personal Digital Assistants (PDA). The very limited resources of these kinds of devices had significant influence on design decisions, regarding both memory architecture and user interface integration within the Palm environment, as well as the handling of top-level definitions. LispMe compiles Scheme expressions to a byte code for a virtual machine which is based on the SECD model.

In this article we show how to overcome the restrictions of the Palm Pilot and present several extensions to the basic SECD model and to the compiler.

1 Introduction

Scheme [6, 10, 18] is a lexically scoped dialect of LISP, including first-class continuations. It is extensively used for research, education, and application programming, since it is a small, but elegant and expressive language.

Palm Pilots [14] are small electronic organizers with very limited memory and computing power, but easily portable in a shirt pocket.

The SECD [8, 13] virtual machine is a simple, but elegant model for implementing LISP-like languages.

This article describes how we combined these three ingredients to form a tool called *LispMe*[1] to program in Scheme anywhere.

1.1 The Palm Pilot and its constraints

Palm Pilots are small hand-held devices and mainly used for organizing purposes, for example storing addresses and dates. Data is input primarily by stylus movement recognition (Graffiti^{TM}) and is backed up to a desktop computer via a RS-232 serial port.

However, Palm Pilots are complete computers running an operating system called PalmOS, and they can be programmed by users, since 3COM (the manufacturer) published the API and a number of development kits, including a port of the GNU C compiler, are available.

[1] There is no special meaning with this name other than it is a cross between LISPKIT and Scheme. One could also read 'Me', though, as an acronym for 'mobile environment' or 'micro edition' like in Java ME.

Due to their size and the need for low power consumption, there are many restrictions[2] for applications running on them:

- 12 kB usable RAM
- 2 kB stack
- max. 64 kB contiguous data blocks
- max. 64 kB code size
- single-tasking OS
- 160 × 160 pixels monochrome screen
- 16 MHz 68000 processor

1.2 Motivation

Now, why would one want to have Scheme running on this kind of hardware? There are several reasons: First, due to its size, the Palm Pilot organizer can accompany its owner anywhere, so one can program in Scheme anywhere one feels like it or anywhere one needs to do so. For example, a number of computer science students reportedly use LispMe to do their homework assignments while on public transfer, and at least one university professor has used it to design class materials while travelling on project work.

Another application of LispMe is as a general scripting language for simple tasks. For example, many people are using the Palm Pilot for gathering data like article numbers and amounts in a warehouse. Since LispMe can access any file on the Palm Pilot, it can evaluate this data and create reports or graphical plots, for example.

Last but not least, implementing Scheme on a device as small as a Palm Pilot was a challenge hard to resist.

1.3 Henderson's LISPKIT

In his textbook [8] Peter Henderson describes a purely functional language called LISPKIT, since it works on S-expressions like Lisp and he also employs a Lisp-like syntax for it, which facilitates meta-syntactic applications introduced in later chapters.

LISPKIT is a fairly small language (much smaller than Scheme), supporting only symbols, integers, and lists. LISPKIT uses static scoping and supports first-class closures like

[2] The figures given here are those for the original Pilot. Some of them where relaxed with later models, but for backward compatibility only in an evolutionary manner, e.g., the size of a memory block is still restricted to 64 kB, so our argumentation still applies.

Scheme. Henderson introduces a virtual machine architecture based on Landin's SECD machine [13], and describes a compiler from LISPKIT to SECD for it. This compiler is written in LISPKIT, translated by hand to machine code, and is bootstrapped afterwards.

Furthermore, Henderson shows some low-level implementation techniques for the basic (von Neumann) machine operations, e.g., storage allocation, input/output, and the virtual machine itself.

Finally, he mentions that heap sizes of 10k cells should be sufficient to run all examples and exercises in his book, including the compiler itself. This property made Henderson's approach eminently relevant for implementing Scheme on the Palm Pilot.

1.4 The LispMe dialect

LispMe implements most concepts and data structures of the R^4RS [6] as well as some common extensions like `eval` and macros. LispMe does not support hygienic macros from the R^5RS [10], but uses tree transformation procedures as in "Scheme and the Art of Programming" [16].

One difference to R^4RS Scheme is how LispMe handles top-level definitions. The reasons and implications are described in Section 5.2.

From the tower of numeric types, small integer, real, and complex numbers including transcendental functions are supported. Big integers and rationals will probably be supported in future versions.

LispMe also includes wrappers for functions specific to the Palm Pilot like graphical I/O, sound output, and database access. Perhaps the most novel feature of LispMe is its seamless integration within the Palm Pilot's event-driven user interface framework.

1.5 Related work

Several implementations of Scheme are based on virtual machines, such as Scheme 48 [11] and PC Scheme [2]. Implementations of continuations are discussed in "Lambda, the Ultimate Label" [5]. Currently no other implementation of Scheme exists for the Palm and there are no published articles about other languages[3] for this platform.

There exists an implementation of Caml Light for the Palm [15], but its development seems to have stalled.

1.6 Source code

LispMe is published under the GNU General Public License, so its complete source code is freely available at the web site [3].

1.7 Terms and abbreviations

A number of key terms require definition before we proceed.

immediate objects Values which are not heap addresses but are special bit patterns used to encode values from [small] domains, like characters and [small] integers.

VM Virtual machine.

[3]Apart from cross-compilers, other languages running entirely on the Palm are Basic, Forth and C. Michael Winikoff's web site [20] contains a fairly complete list of programming language implementations for the Palm.

variable-arity closure A closure accepting a variable number of arguments. They are defined by using a dotted parameter list (see also Dybvig and Hieb [7]).

pickling a value Transferring a value from the dynamic heap to the database heap, possibly requiring changing its internal structure.

unpickling a value The inverse operation of pickling.

dynamic heap The section of the Palm RAM freely usable by the current application.

database heap The rest of the Palm RAM. It is organized like a file system and (normally) write-protected.

Scheme heap The part of the memory where Scheme pairs are stored (in contrast to the last two terms originating from PalmOS).

1.8 Overview

The rest of this article is organized as follows:

- Section 2 shows how we designed LispMe's memory architecture to fit within the Palm Pilot's memory model.
- Section 3 describes the interaction between the Palm Pilot user interface and LispMe.
- Section 4 describes LispMe's extension module mechanism.
- Section 5 discusses some design decisions in the context of the Palm Pilot environment.
- Section 6 reminds the reader of the original SECD machine model and describes LispMe's extensions to it.
- Section 7 does the same with the compiler.

2 Memory architecture

2.1 Palm Pilot memory overview

The main processor of a Palm Pilot is the Motorola Dragonball, which is a 68000 architecture extended with LCD controller and other features which are irrelevant for this article.

Palm devices contain 1–2 MB of ROM containing PalmOS and the built-in applications (Datebook, Memopad, etc.) and 1–8 MB of RAM, which is divided into two blocks:

- 32–96 kB of "real" (writable) RAM called the *dynamic heap*. This range is further divided into
  - Screen memory
  - System variables, buffers, etc.
  - Stack (only 2–4 kB)
  - Finally, only 12–36 kB are usable for applications as RAM in the classical meaning.
- The rest of the RAM (the *database heap*) is write-protected and is managed like a file system (Palm Computing calls those files "databases"). Each database consists of a number of records of variable length, each of which must be less than 64 kB. Each record may be moved around in the address space when the operating system tries to defragment the

memory, unless the record is locked during use by an application. To write to a record, special system calls are employed, which carefully check their parameters and disable hardware write-protection for a short time.

Each application is also a special kind of database, which contains both the machine code and the graphical resources. The machine code need not be loaded into the "real" RAM for execution, but executes in place, which makes application switches very fast and justifies the claim that PalmOS is in fact a single-task OS. Applications should save their state when the user switches to another program and restore it immediately upon re-starting the application.

2.2 LispMe memory usage

Having only 12–36 kB of free RAM for the entire Scheme heap and other run-time data structures is much too restrictive, so we looked for other methods. One possibility (which is employed in LispMe) is keeping the entire memory image in a Palm database, but the requirement to use system traps for every single write access is very expensive in terms of time.

2.3 Disabling write-protection

Fortunately, it is possible to disable write-protection of the database records by a system call. This technique can be dangerous, since a buggy program may not only crash the application itself, but destroy other applications (code and data), too. In the worst case it may erase the entire device by overwriting central memory allocation structures.

But considering the alternatives (only a very small Scheme heap or very slow write access), this seems to be the only solution. There are several other Palm Pilot programs using a similar technique to access a larger amount of memory than the system usually permits. As long as such a program is carefully designed and tested, disabling write-protection appears to be a practical solution. In fact, we never received any report from users about data loss[4].

2.4 The session concept

Another advantage of storing the Scheme heap in a Palm database becomes obvious when considering application switches: since PalmOS is effectively single-tasking, LispMe would have to save its memory image to a database while another application is running (for example the user is editing some Scheme source with the Memopad) and restore it afterwards. But LispMe's data *is* already in a database, so nothing need to be done at all (with the exception of certain foreign data types, see section 4.2). Even a running evaluation can be suspended by switching to another application and will be continued on re-starting LispMe.

Furthermore, there can be many databases, each containing an entire memory image, making it possible to have several independent co-existing LispMe instances. These instances are called *sessions* and the corresponding databases can be backed up, transferred to other Palms or distributed over the Internet.

Any session can be associated with an icon on the Palm application selection screen, which invokes LispMe with the specified session database and starts evaluating the current expression. This method allows writing "stand-alone" Palm applications in LispMe.

2.5 Memory layout

In each session database, at least 6 records are used:

1. Global variables, root pointers
2. The atom store; all symbol names (which are not built-in names) are entered here by the reader. The names are simply stored one after another separated by `'\0'` bytes.[5] Atoms are not garbage collected, which is no real problem with program sizes possible on the Palm.
3. The floating point store; all floating point numbers are stored here in IEEE-754 *double precision* format, occupying 8 bytes each. Unused floating point slots are linked into a list.
4. The Scheme heap; each heap cell is 32 bits and consists of two pointers (`car` and `cdr`), 16 bits each. The cells do *not* carry type tags, instead the pointers do so, as shown in table 1.

   All pointers into the Scheme heap are *relative pointers* (offsets) to a virtual heap base lying 32 kB behind the actual beginning of the heap. This approach allows relocation of the heap by the operating system. Additionally, this saves memory (16 bit relative pointers vs. 32 bit absolute pointers) and maps nicely to the 68000 addressing mode *address register indirect with index and offset*, which uses signed offsets [19].
5. The contents of the input field (see 3.1 for details)
6. The contents of the output field (see 3.1 for details)
7. Each vector and string occupies a single database record as described in 2.7.

This memory model can easily be extended to 32 bit pointers by changing the access macros for use on desktop computers. In fact, we did this in a test implementation of the SECD machine running under Linux.

2.6 Garbage collection

LispMe uses a mark/scan garbage collector for heap cells and floating point cells. To store the mark bits, a temporary bit vector in the dynamic heap is used. The reasons for preferring mark/scan to copying garbage collection are:

- Memory usage has top priority on the Pilot. Wasting half of the memory with copying GC is not affordable.
- All heap cells have the same size, so fragmentation cannot happen
- Heap compaction (provided by copying GC) does not affect performance, since there is no virtual memory or cache on the Pilot.
- The disadvantage that mark/scan GC has to touch every memory cell in the Scheme heap is insignificant with heap sizes possible on the Pilot.

This approach is justified by the fact that a typical garbage collection of 16k heap cells takes about 0.2 seconds which is barely noticable.

[4] According to web server statistics, LispMe has been downloaded about 15000 times (until October 2000). This number includes all version updates, so we estimate several hundreds of LispMe users.

[5] Symbols do not have additional structure in LispMe, they consist solely of their printing name. Additionally, symbols can only be created by the reader or the **gensym** procedure, there is no **string->symbol** in LispMe, so the `'\0'` byte used as a separator cannot be part of a symbol.

`sddd dddd dddd ddd1`	15 bit signed integer (-16384–16383)
`sddd dddd dddd dd00`	pointer into the Scheme heap, 16 bit signed offset from virtual heap base
`dddd dddd dddd 0010`	unsigned 12 bit index into atom table (shift right 4 bits), 4 kB atom space
`dddd dddd dddd 0110`	unsigned 12 bit index into double table (shift right 1 bit and unmask 3 low bits), 4096 reals
`aaaa aaaa 1000 1110`	8 bit ASCII char
`uuuu uuuu 0100 1110`	vector in heap, upper 8 bit of UID, cdr field of this cells contains lower 16 bit (untagged)
`uuuu uuuu 0101 1110`	string in heap, upper 8 bit of UID, cdr field of this cells contains lower 16 bit (untagged)
`ffff ffss sss0 1010`	built-in symbol, 6 bit frame and 5 bit slot index
`ffff ffss sss1 1010`	built-in value, 6 bit frame and 5 bit slot index
`tttt tttt 0111 1110`	foreign data, 8 type bits, cdr field of this cell points to 32 bit value (untagged)

Table 1: Bit patterns used for LispMe values.

2.7 Strings and vectors

Strings and vectors are kept outside the main Scheme heap, in a separate database record each. They are accessed by a descriptor cell in the heap containing the 24 bit "unique identifier" of the record. The length of the vector or string need not be stored since it can be retrieved from the record handle by a system call.

Strings can contain any characters, including `'\0'` bytes, which allows using strings for low-level binary data (bitmaps for example) in interfaces to Palm system calls. Since records of size 0 are not allowed, the empty string and the empty vector are special immediate objects.

2.8 Immediate objects

Several kinds of objects are directly encoded in the pointer value without requiring cells from the Scheme heap or other objects. These are

- small integers from -16384 to 16383
- all characters
- some special values `#t`, `#f`, `()`, `""`
- special tags used to identify internal types (closures, continuations, macros, delayed expressions)

2.9 Built-in symbols and primitive operations

To avoid filling the very limited atom store with the names of LispMe's primitive operations, they are kept in a two-dimensional (ragged) table. This table is accessed by a 6 bit *frame index* and a 5 bit *slot index*, fitting in a single 16 bit pointer. Separating frame and slot indices is advantageous for grouping related operations and is the basis of the module extension mechanism described in Section 4.1.

Frame 0 of the primitives table is reserved for the special values and tags mentioned in Section 2.8.

2.10 Foreign types

To store PalmOS-specific values (like database handles, sockets, or calendar dates), LispMe uses ordinary cells of the Scheme heap where the car contains a type tag as shown in table 1 and the cdr points to a cell containing a 32 bit word, which is interpreted depending on the type tags. By putting the descriptor cell at a lower address than the value itself, we can ensure that the arbitrary bit pattern will not be misinterpreted by the garbage collector.

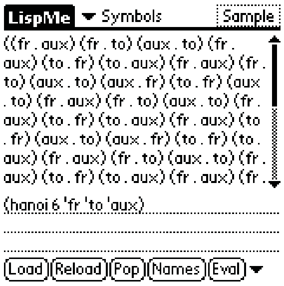

Figure 1: The LispMe main dialog.

2.11 Summary and conclusion

Even within the severe constraints of the Palm's memory model a Scheme heap of a useful size can be implemented for which a simple garbage collection algorithm is sufficient. The persistence of a memory image during invocations of LispMe and the possibility of having an unlimited number of independent memory images both have proved very useful.

The memory model is scalable and can be adapted to future versions of PalmOS allowing bigger memory blocks. In this case, other garbage collection strategies may be preferable.

3 The user interface model

The layout and attributes of the user interface elements (buttons, entryfields, lists) are stored in *resource databases*, which can be created by a variety of tools.

These controls are assigned behaviors by employing an event loop which waits for events from the system event queue and calls the appropriate event handler.

Typically, a Pilot application consists of several event handlers (one for each dialog or "form" in Palm jargon), which receives elementary events from the system and either handles them or passes them on to other handlers, which may create other more complex events from them[6].

3.1 LispMe UI

LispMe itself must follow this event model, since there is unfortunately no `stdio` style I/O system[12], so a form is used, which contains both an input and an output text field.

Additionally, there is no standard mechanism like *CTRL-C* to interrupt an evaluation and the user application may only use one thread, so the VM itself has to check for the

[6] For example, a *pen down* event followed by a *pen up* event within a rectangle specified in a resource file is transformed to a *button pressed* event.

Break button. The VM does so after 1600 VM steps or after output operations or garbage collections to maintain a balance between check overhead and responsiveness. MacScheme [5] uses a similar technique, but counts procedure calls instead of VM instructions.

Writing programs on a Pilot using Graffiti can be a tedious task, so we tried to provide a sophisticated development environment to minimize taps and pen strokes:

- Parentheses matching
- List of all known symbols with completion option
- Display argument list of both built-in functions and closures
- History of expressions evaluated
- Evaluate selected text region

3.2 Application UI

Some Palm Pilot programming environments, like *PocketC* and *KVM*, use an ad-hoc model for implementing graphical interfaces. This means they draw the elements of the interface using graphic primitives and handle all events themselves without using the PalmOS UI functions.

Those interfaces generally behave in a different way than native PalmOS applications and furthermore require additional libraries and thus memory.

A better way is to use the existing PalmOS UI model: Dialogs are created as resource databases using any of the available tools, and the event handlers are coded in Scheme. LispMe handles all necessary details to call a Scheme closure when receiving a native system event.

Early versions of PalmOS did not support dynamic creation of UI controls, all UI elements had to be placed in a resource database and had to be accessed by their numerical id, so LispMe had to follow this low-level mechanism, too. However, PalmOS 3.0 introduced dynamic creation of controls, so a higher-level mechanism for UI access (by using *foreign types* encapsulating the underlying PalmOS data structures, e.g.) is now possible. Future versions of LispMe will utilize this mechanism.

3.3 Lifecycle of a dialog

A dialog is created in LispMe using the call `(frm-popup` *id handler*`)`[7] which does the following:

1. Saves possible current dialog context onto a stack (see below).
2. Displays the dialog with resource identifier *id* (which is loaded from the current open resource database).
3. Installs the Scheme closure *handler* (which should accept a variable number of parameters) as the current event handler.

Now each low-level event received by LispMe is transformed into a symbolic representation and the Scheme event handler is called. The handler should return `#t` to indicate that the event has been handled and should not be passed on to other (system) handlers, or `#f` to allow passing on the event.

[7]The user interface functions are named similarly to the underlying functions in the PalmOS API[14]. The prefix `frm` is an abbreviation for *form*, the Palm jargon for dialog.

The call `(frm-return` *value*`)` terminates the current dialog and makes *value* the return value of the original `frm-popup` call, so the 'context' indicated above is in fact the Scheme continuation of the `frm-popup` call.

3.4 Dialog hierarchy

Dialogs can be nested: By issuing another `frm-popup` call in an event handler a new subdialog is displayed which has is own event handler. So the data structure stored is a stack of pairs of a closure (the event handler) and a continuation.

Another possibility is replacing the current dialog by another (instead of nesting it) by using `(frm-goto` *id handler*`)`. This discards the current event handler and stores no continuation, but simply installs the new handler. A subsequent `frm-return` activates the continuation of the last `frm-popup` call.

3.5 Application example

Assume a simple application to count items by tapping a button. It has two buttons, one to increment and one to reset the counter and a (read-only) text field displaying the current count. The dialog has resource id 1000, the text field 2000, the increment button 3000 and the reset button 3001. The LispMe code for this application is:

```
(define (run)
  (set-resdb "Sample resource")
  (frm-popup 1000 handler)) ; display dialog

(define handler
  (let ((count 0)) ; note: outside of lambda
    (lambda (event . args) ; note var. args
      (case event
        ((frm-open) ; dialog has been opened
          (fld-set-text 2000 count)) ; display counter
          ; LispMe automatically converts to a string
        ((ctl-select)  ; a button has been pressed
          (case (car args) ; which button?
            ((3000) ; increment button
              (set! count (+ count 1)))
            ((3001) ; reset button
              (set! count 0)))
          (fld-set-text 2000 count)) ; redisplay counter
        (else #f)))))  ; pass on other events
```

Note that `count` is in the environment stored with the handler closure, so it persists from one handler call to the next one.

Just like any other closure in Scheme, an event handler is called in the environment in effect when its lambda abstraction has been executed. This way one can write higher-order functions returning event handlers whose state is kept in the enclosing environment. Thus one can have several instances of the same dialog (but with different values) in an object-oriented style.

Another possibility is concatenating several event-handling procedures using higher-order combinators, e.g., to include common handling code for a particular menu into many different dialogs.

3.6 Summary and conclusion

In comparison with other languages Scheme, due to its lexical closures and higher-order procedures, is very suitable for coding handler procedures in an event-driven user interface. Integrating this paradigm within the Palm's GUI framework was quite straightforward.

4 Extending LispMe

4.1 Modules

An extension module is an object file which is statically linked to the LispMe executable[8]. It provides an array of primitive symbols as described in Section 2.9 together with their interpretation which can be:

1. An array of VM opcodes to be generated.
2. The type signature of the primitive operation and the address of a native C function to be called by the VM.
3. The address of a C function to compile the special form introduced by this keyword. New VM opcodes can also be put into the generated code which dispatch to a VM extension function.

An extension module can optionally define a *module control function*, which is called on several events to ensure proper module initialization:

- Application start and stop: Open and close system libraries, register hooks for foreign types (see Section 4.2).
- Session creation and deletion: Create and destroy databases or other resources specific to this module.
- Scheme heap initialization: Create variables, reset other resources.
- Session activation and deactivation: Cache heap-based variables, unpickle and pickle other resources.

4.2 Foreign types

Foreign types are defined in extension modules (see Section 4.1). To integrate foreign types seamlessly into the rest of LispMe, extension modules can register C functions to be called

- when printing a foreign value
- when comparing two foreign values (of the same type) with `eqv?`
- when marking a foreign value during garbage collection
- when destroying a foreign value, either due to garbage collection or session deletion
- when pickling or unpickling a foreign value

The last item is perhaps the most interesting. Since PalmOS is a single tasking OS, the LispMe application must be stopped and restarted when using other applications and thus save and restore data from the dynamic heap to more permanent memory. All kernel objects are stored in database memory, anyway (see Section 2.5), so copying them is not necessary, but foreign values often contain pointers to the dynamic heap.

In many cases, foreign values cannot be saved by simple binary copy. Instead they must be converted to an external representation, for example, open file handles are converted to file names and reopened when the session is reactivated.

[8]We are planning to provide dynamic (runtime) linking in future versions.

5 Design decisions

5.1 Internal data structures

Both LISPKIT and LispMe use lists in several places where other Scheme implementations use vectors for implementing environments and VM code sequences.

The reason is both historical and technical. LISPKIT does not provide vectors at all, and so LispMe originally did not include them, either. Vectors were added later to LispMe but the Pilot's memory architecture causes a slight (constant) overhead for vector accesses and a significant overhead for vector creation, so vectors were *not* used to re-implement environments.

Usually argument lists and environment frames are small — especially when considering the Palm Pilot's limit of 16k Scheme heap cells — so the linear complexity of lists results in acceptable performance. Additionally, list-based implementations enable elegant translations of function calls and conditionals, as shown in Section 6.1 and allow an elegant optimization for accessing global variables, as shown in Section 6.8.

5.2 Top-level environment

Steele and Sussman discuss in *The Art of the Interpreter* [17] top level bindings versus referential transparency and come to the conclusion that free variables should be bound dynamically in the top-level environment to allow incremental program development.

However, for a small system like LispMe, another approach which was partly inspired by the Hugs Functional Language Environment [9] seemed practical.

5.3 LispMe's approach

LispMe handles top-level definitions differently from other Scheme implementations, i.e., it does not accumulate them into a single top-level environment. LispMe further requires each name to be bound statically and does not resolve top-level names dynamically.

This behaviour is similar to the Hugs Functional Language Environment [9] which strictly separates definition scripts from the expressions entered at top-level (which are evaluated using the definitions from scripts loaded earlier, but does not allow entering new ones). LispMe extends this system in three ways:

1. It allows definitions entered interactively and treats them like a single source file.
2. It allows "popping" the last loaded source file (or interactively entered definition) from the stack of definitions.
3. Like Hugs, it can show a list of all identifiers currently defined, but grouped by source file (unlike Hugs).

Both LispMe and Hugs require that all names must be defined in the current source or in a previously loaded one, so all variable addresses can be computed once and for all at compile-time, avoiding the difficulties with the mixture of lexical and by-name references described by Abelson and Sussman [1] in the section "Compiling `define` expressions"[9].

[9]On the other hand, this requirement allows removing the topmost environment frame later without invalidating lexical addresses.

One thing Hugs does and LispMe does not, is automatically partitioning a source file into cliques of mutual dependent definitions, though this is only necessary when a source file contains function definitions which are called while loading the source (to initialize some variables).

Each source file to be loaded is treated as a set of mutual recursive definitions and is thus transformed into a equivalent `letrec`-block with an empty body. On loading a source file, this `letrec`-block is compiled and executed. As usual, during execution of the block, the environment is extended by a frame containing all (top-level) bindings of the source. Now this environment frame is captured and pushed onto the global environment stack, so the net effect of loading a source file is extending the global environment by a frame containing the definitions from the source.

As an example, assume the following expressions entered sequentially into LispMe's top-level:

```
(define x 1)
(define f (lambda () x))
(define x 2)
(f)
```

The last expression evaluates to `1`, because it is interpreted as

```
(letrec ((x 1))
  (letrec ((f (lambda () x)))
   (letrec ((x 2))
     (f))))
```

so the result is `1` since `x` is lexically bound and cannot be rebound later. Of course, `x` can be modified with `set!` later to achieve the effect of a redefinition.

LispMe implicitly puts all definitions in a source file into a `begin` block to allow mutually recursive definitions.

5.4 Discussion

By requiring all used names to be defined earlier, LispMe encourages a bottom-up style of programming, where standard Scheme additionally offers the top-down approach by allowing unbound identifiers resolved at runtime.

Steele and Sussman's [17] motivation was to allow independent definition, replacement and debugging of parts of a (presumably) large program. All effects of redefinitions of top-level names can be achieved by assignment using `set!`[10], so the only thing missing in LispMe is the possibility to write definitions using not-yet-defined names. But this feature is more likely needed in larger programs not possible on the Palm Pilot anyway. In any case, one can still write dummy definitions for those names in the meantime.

On the other hand, since LispMe does not accumulate all definitions into a single top-level pool, but maintains a stack of loaded source files allowing removal of definitions, this provides the user with easy control over the code loaded. This is especially useful in a restricted system like the Palm. The positive response from LispMe users about its interface further assured us that this approach is appropriate.

6 The SECD virtual machine

The SECD machine model used by LispMe is largely based on the design presented by Henderson [8]. However, in LispMe it is extended in several ways.

[10] LispMe provides an option to transform re-definitions of already bound variables into assignments to avoid having to modify the Scheme source code.

6.1 The original SECD machine

The SECD machine derives its name from the four main registers of the virtual machine (VM):

S the stack — used to hold intermediate values during evaluation

E the environment — used to hold variable values during evaluation

C the code — used to hold the machine code program being executed

D the dump — used as a stack to hold other registers while calling a function

Theoretically, **S** and **D** could be combined into a single register, but this would complicate the state transitions for the VM instructions. Each of the registers contains [a pointer to] an S-expression. To show the effect of a VM instruction, *state transitions* are used, which describe the contents of each register before and after one VM step.

$$s \quad e \quad c \quad d \quad \rightarrow \quad s' \quad e' \quad c' \quad d'$$

For example, the state transition for the VM instruction *LDC*[11] used to load a constant on top of the stack is

$$s \quad e \quad (LDC\ x.c) \quad d \quad \rightarrow \quad (x.s) \quad e \quad c \quad d$$

meaning that **E** and **D** have not changed, the code list in **C** has been shortened by the first two elements and the constant x appearing after the instruction *LDC* has been pushed onto the stack **S**.

All implemented VM instructions and the corresponding numeric codes are defined in the source file `vm.h`

The meaning of most instructions is quite straightforward, since many of them encapsulate operations on simple datatypes, which expect their arguments on the stack (**S**) and leave their result on the stack, too:

$$(a\ b.s) \quad e \quad (SUB.c) \quad d \quad \rightarrow \quad (b-a.s) \quad e \quad c \quad d$$

However, three mechanisms are noteworthy, both because they differ from other virtual machines and undergo further extensions in LispMe:

6.2 Conditionals

Executing different code sequences depending on a condition works differently from most other VM architectures, where usually jumps to compiler-generated machine code addresses are employed. The SECD machine exploits the fact that register contents are arbitrary trees, so an entire code sequence can occupy a single "machine code location".

To implement the `if` special form of Scheme, the *SEL* instruction is used, which activates one of two code sequences depending on the truth value on top of the stack:

$$(x.s) \quad e \quad (SEL\ c_t\ c_f.c) \quad d \quad \rightarrow \quad s \quad e \quad c_t \quad (c.d)$$

$$(\#f.s) \quad e \quad (SEL\ c_t\ c_f.c) \quad d \quad \rightarrow \quad s \quad e \quad c_f \quad (c.d)$$

[11] For performance reasons, the instructions are encoded within a 16 bit word.

In any case, the rest of the code (the "continuation") is saved on the dump **D**, whereas the environment **E** does not need to be saved.

Each of the two branches is expected to end with a *JOIN* instruction, which restores the continuation saved on the dump:

$$s \quad e \quad (JOIN) \quad (c.d) \quad \to \quad s \quad e \quad c \quad d$$

6.3 Function calls and environments

Each function call builds an association between the function parameter names and the actual arguments. Free variables are resolved in an outer (lexical) environment in the standard Scheme (or lambda calculus) fashion, so an environment is a stack of associations and implemented as a list of lists.

LISPKIT (and thus LispMe) separate names from values in their environment structures: there are in fact two isomorphic (two-level) trees. One contains the variable names (symbols) and the other one contains the associated values on corresponding positions. This has the advantage that variable names do not have to be stored at runtime and construction of a single environment frame at runtime is very simple: Just `cons` all arguments of a function call into a list and use it as the innermost environment frame. Steele and Sussman already used this technique in the original Scheme report [18] (see "This ain't A-lists").

All variables are addressed using a two-component address:

1. the (relative) frame number beginning with the current frame
2. the position of the value in this frame

The compiler transforms all variable references into these addresses and thus provides lexical scoping. Unbound variables cause a compile-time error.

To apply a function, LISPKIT expects two values on the stack:

1. Topmost: the closure (or continuation, see later) to apply
2. Next to top: a list of all arguments

Given that, the *AP* instruction saves all registers on the dump (**D**) installs the closure's code in **C** and builds a new environment from the function arguments and the environment part of the closure (*not* the current environment to provide proper lexical scoping) and executes **C** with an empty stack:

$$((c'.e')v.s) \quad e \quad (AP.c) \quad d \quad \to \quad () \quad (v.e') \quad c' \quad (s\,e\,c.d)$$

When a function application has finished, it executes the *RTN* instruction, which restores the registers saved on the dump previously and pushes the return value of the function:

$$(x) \quad e' \quad (RTN) \quad (s\,e\,c.d) \quad \to \quad (x.s) \quad e \quad c \quad d$$

6.4 Recursion

To be able to refer by name to the function being applied, the closure's environment must contain a reference to the closure itself. To implement this cyclic structure, LISPKIT uses 2 new VM instructions: *DUM* creates a new dummy environment frame Ω implemented by a special tag. The transition for *DUM* is

$$s \quad e \quad (DUM.c) \quad d \quad \to \quad s \quad (\Omega.e) \quad c \quad d$$

Trying to access this frame results in LispMe's *Sucked into black hole* error message, which is not a problem, since the dummy frame will be replaced later, anyway.

RAP ("recursive apply") is much like *AP*, but instead of extending the environment, it expects the top-most environment frame to be Ω and replaces it using `set-car!`:

$$((c'.e')v.s) \quad (\Omega.e) \quad (RAP.c) \quad d \quad \to \quad () \quad e'' \quad c' \quad (s\,e\,c.d)$$

where e'' is the cell obtained by `(set-car! e' v)`. Because of the construction of the code list, *RAP* is always executed when $e' = (\Omega.e)$ and thus creates the cycle described above.

6.5 Extending closures

To provide error checking, we have extended the representation of closures. Henderson's closures are simple pairs of the code list and the environment, which inhibits error checking. LispMe adds a type tag to the closure and further includes the arity a in the closure object, leading to the representation $([clos]\,a\,c.e)$

To denote variable numbers of parameters, negative arities are used as follows: -1 indicates any number of parameters (created by `(lambda x ...)`), -2 one required and any number of optional parameters (created by `(lambda (x . rest) ...)`) and so on.

The compiler always generates the same code for a function call, disregarding the arity of the closure. Instead, arity check is done at runtime and in the case of variable-arity closures [7], the argument list is transformed by creating a new `cons` cell and thus "pushing the remaining arguments one level deeper".

The state transition for this new "apply checked" instruction is almost the same as *AP*:

$$(([clos]\,n\,c'.e')v.s) \quad e \quad (APC.c) \quad d \quad \to$$
$$() \quad (v.e') \quad c' \quad (s\,e\,c.d)$$

The original opcodes *LDF* and *AP* are still supported and are used for `let`-expressions where no arity and type checks are necessary.

6.6 Tail recursion

Tail calls do not have to save the current state onto the dump **D**, so the following transition

$$(([clos]\,n\,c'.e')v.s) \quad e \quad (TAPC) \quad d \quad \to \quad s \quad (v.e') \quad c' \quad d$$

yields the desired result.[12]

[12] Note that due to its construction, *TAPC* is always the last instruction in a code sequence.

To provide proper tail recursion semantics, several other instructions do not extend the dump, for example *SELR*:

$$(x.s) \quad e \quad (SELR\, c_t\, c_f) \quad d \quad \to \quad s \quad e \quad c_t \quad d$$

$$(\#f.s) \quad e \quad (SELR\, c_t\, c_f) \quad d \quad \to \quad s \quad e \quad c_f \quad d$$

Of course, none of the branches c_t and c_f must end with a *JOIN* instruction now — a condition which is ensured by the compiler.

6.7 Continuations

Since in the SECD machine there is no processor stack involved and all registers are cell references, implementing continuations is straightforward. A continuation object is a list of all 4 VM registers, tagged with a special value ($[cont]\, s\, e\, c.d$) to allow runtime type checking.

The new instruction *LDCT* captures the current continuation and pushes it as a singleton list[13] onto the stack:

$$s \quad e \quad (LDCT\ c'.c) \quad d \quad \to \quad ((([cont]\, s\, e\, c'.d)).s) \quad e \quad c \quad d$$

The application instructions *APC* and *TAPC* have been extended to accept continuations, too: They push the argument given to the continuation invoked onto the stack after restoring all registers from the continuation.

6.8 Optimizing access to global variables

Since the lexical address of a variable never changes during its extent and global variables always occupy the same environment frame, a global variable always refers to the same storage cell.

By using special instructions *LDG* and *STG* and embedding the storage cell *itself* — instead of the lexical address — in the code list, we reduced the complexity of access to global variables from $O(n)$ to $O(1)$.

$$s \quad e \quad (LDG\ (v.e').c) \quad d \quad \to \quad (v.s) \quad e \quad c \quad d$$

There is a small difficulty when constructing the cyclic environment while loading mutual recursive definitions, since the topmost environment frame has not been constructed yet when compiling variable accesses. We solved this problem by generating another set of special instructions (*LDU* and *STU*) using ordinary lexical addresses, and by modifying the VM code on first access to the variable:

$$s \quad e \quad (LDU\ la.c) \quad d \quad \to \quad s \quad e \quad (LDG\ (v.e').c) \quad d$$

where $(v.e')$ is the cell obtained by $locate(la, e)$.

6.9 Summary and conclusion

The SECD machine is a very suitable base for implementing Scheme within limited resources. Additionally, it allows straightforward extensions like continuations.

[13] The receiver procedure this continuation will be passed to expects one argument, so creating this argument list within the *LDCT* instruction saves one instruction.

7 The compiler

7.1 Henderson's approach

Henderson [8] describes a function $comp(expr, names, cont)$ returning a code sequence *code* with the property

$$eval(expr) = exec(comp(expr, (), (STOP)), ())$$

where $exec(code, env)$ is the SECD interpreter function started in the state $\langle (), env, code, () \rangle$.

The LISPKIT expression to be compiled is *expr*, *names* is the compile-time environment and *cont* is an accumulation parameter to avoid using `append` on the intermediate code lists. But of course *cont* is in fact the continuation (though Henderson never uses this word in this context) of *expr*, which is exploited in the extensions introduced by LispMe.

Here are some sample compilation equations:

$$comp(var, n, c) \quad = \quad list{*}(LD, location(var, n), c)$$

where *location* computes the position of the variable *var* in the compile-time environment e

$$comp(\texttt{(+}\ e_1\ e_2\texttt{)}, n, c) \quad = \quad comp(e_1, n, comp(e_2, n, cons(ADD, c)))$$

$$\begin{aligned} comp(\texttt{(if}\ e_?\ e_t\ e_f\texttt{)}, n, c) &= comp(e_?, n, list{*}(SEL, c_t, c_f, c)) \\ \textbf{where}\ \ c_t &= comp(e_t, n, (JOIN)) \\ \textbf{and}\ \ c_f &= comp(e_f, n, (JOIN)) \end{aligned}$$

$$\begin{aligned} comp(\texttt{(}e\ e_1 \ldots e_k\texttt{)}, n, c) &= append((LDC\ NIL), c_k, \ldots, c_1, p, c) \\ \textbf{where}\ \ p &= comp(e, n, (AP)) \\ \textbf{and}\ \ c_i &= comp(e_i, n, (CONS)) \\ & \quad \forall i \in \{1, \ldots, k\} \end{aligned}$$

$$\begin{aligned} comp(\texttt{(lambda}\ p\ e\texttt{)}, n, c) &= list{*}(LDF, body, c) \\ \textbf{where}\ \ body &= comp(e, cons(p, n), (RTN)) \end{aligned}$$

The last equation shows the extension of the compile-time environment by a frame containing the function's parameter names.

7.2 The LispMe compiler

Henderson [8] uses bootstrapping for the compiler written in LISPKIT code. Though this approach is instructive in an introductory textbook, it was impractical for LispMe considering the memory constraints. To allow as much usable Scheme heap as possible, we decided to implement the compiler entirely in C.

7.3 Optimized parameter passing

The original LISPKIT is somewhat inefficient when building the argument list for a function call. Each argument is pushed onto the stack **S** and afterwards consed to the partial argument list already on the stack, resulting in $2n$ `cons` calls and $n+1$ additional VM instructions for each function call with n arguments.

Since the stack is internally represented as a list, it is an obvious optimization to just push all arguments onto the stack without intervening *CONS* instructions and to directly use the top-most n stack cells as the argument list. We introduced a new instruction *LST* for this purpose:

$$(v_1 \ldots v_n.s) \quad e \quad (LST\ n.c) \quad d \quad \rightarrow \quad ((v_1 \ldots v_n).s) \quad e \quad c \quad d$$

Together with the new translation rule

$$\begin{aligned} comp((e\ e_1 \ldots e_k), n, c) &= append(c_k, \ldots, c_1, (LST\ k), p, c) \\ \textbf{where}\ p &= comp(e, n, (AP)) \\ \textbf{and}\ c_i &= comp(e_i, n, ()) \\ & \quad \forall i \in \{1, \ldots, k\} \end{aligned}$$

replacing the one from Section 7.1, a function call with n arguments now needs only n `cons` calls and 1 additional VM instruction.

Unfortunately, this optimization is not possible when `apply`ing a procedure to a list. Since the called procedure might modify its parameters, the argument list cannot be shared and must be copied. The compiler emits the instructions *APY* or *TAPY* in this case.

7.4 Tail calls

Tail calls can be recognized in the compiled code sequences by the pattern `(... APC RTN)`, so replacing this sequence by `(... TAPC)` yields the desired result. Thus *TAPC* is always the last instruction in a code sequence as mentioned earlier.

But matters get more difficult when one remembers the non-linear nature of SECD machine code: A simple conditional separates the *APC* from the *RTN* instruction like this:

```
(define (exp2 n acc)
  (if (eqv? n 0) acc (exp2 (- n 1) (* 2 acc))))
```

compiles to

```
(LDC 0 LD (0 . 0) EQV SEL
  (LD (0 . 1) JOIN)
  (LD (0 . 1) LDC 2 MUL LDC 1 LD (0 . 0) SUB LST 2
   LD (1 . 0) APC JOIN)
  RTN)
```

In this case, one can move *RTN* into both branches of the *SEL* instruction, where *RTN* replaces *JOIN* (both branches) and creates the reducible sequence `(... APC RTN)` in the second branch. Since there is no *JOIN* anymore, we do not need to save the rest of the code onto the dump (**D**) like in the *SEL* instruction. Instead, we invent a new instruction *SELR* behaving like *SEL*, but without saving the continuation:

$$(x.s) \quad e \quad (SELR\ c_t\ c_f) \quad d \quad \rightarrow \quad s \quad e \quad c_t \quad d$$

$$(\#f.s) \quad e \quad (SELR\ c_t\ c_f) \quad d \quad \rightarrow \quad s \quad e \quad c_f \quad d$$

with the additional compilation rule

$$\begin{aligned} comp((\texttt{if}\ e_?\ e_t\ e_f), n, (RTN)) &= comp(e_?, n, list(SELR, c_t, c_f)) \\ \textbf{where}\ c_t &= comp(e_t, n, (RTN)) \\ \textbf{and}\ c_f &= comp(e_f, n, (RTN)) \end{aligned}$$

So altogether, the previous example compiles to

```
(LDC 0 LD (0 . 0) EQV SELR
  (LD (0 . 1) RTN)
  (LD (0 . 1) LDC 2 MUL LDC 1 LD (0 . 0) SUB LST 2
   LD (1 . 0) TAPC))
```

with tail call elimination. Similar code merging is done for other instructions when the continuation is *RTN*.

7.5 Continuations

As mentioned earlier, supporting continuation with a heap-based SECD machine is simple. The only addition are these new compilation rules:

$$\begin{aligned} comp((\texttt{call/cc}\ e), n, c) &= list{*}(LDCT, list(c), cont) \\ \textbf{where}\ cont &= comp(e, n, cons(APC, c)) \\ comp((\texttt{call/cc}\ e), n, (RTN)) &= list{*}(LDCT, (RTN), cont) \\ \textbf{where}\ cont &= comp(e, n, (TAPC)) \end{aligned}$$

Thus the original code continuation c is used twice, once in the usual way and additionally as an argument to the *LDCT* instruction, which creates a Scheme continuation object at runtime as described in 6.7.

7.6 (Non-)Macros

The first release of LispMe did not allow macros, so we implemented almost all non-core syntactic forms like `and`, `case`, `let`, `delay`, `quasiquote` natively in the compiler and we often used special VM instructions yielding better optimized code than is possible with the well-known standard macros. For example, compiling `let` uses the original SECD instructions *LDF* and *AP*, which create and apply an unchecked closure, whereas using a macro which expands `let` to the equivalent `lambda`-application would create *LDFC* and *APC* instructions, performing superfluous type and arity checks.

For this reason (and to avoid having to load an init file defining those syntactic forms) we left all these forms and the additional instructions in LispMe even after introducing the macro facility.

7.7 Primitive procedures

All primitive procedures are inlined (if the symbol has not been rebound), so they do not require any heap space.

When `apply`ing a primitive procedure, the compiler is invoked[14] to generate the code on the fly and prefix it to the **C** register.

$$\begin{gathered} ([prim\,p]\ v.s) \quad e \quad (APC.c) \quad d \quad \rightarrow \\ append(v, s) \quad e \quad (compile\_builtin([prim\,p]).c) \quad d \end{gathered}$$

$$\begin{gathered} ([prim\,p]\ v.s) \quad e \quad (TAPC) \quad d \quad \rightarrow \\ append(v, s) \quad e \quad (compile\_builtin([prim\,p])\ RTN) \quad d \end{gathered}$$

7.8 Summary and conclusion

Though the basic structure of the LispMe compiler is very similar to Henderson's compiler, we have extended it in various ways, showing the versatility of Henderson's approach. Implementing the compiler in C instead of Scheme was important to retain a maximal Scheme heap within the Palm's limited resources.

[14] in a special mode to avoid generating code for pushing the arguments onto the stack **S**

8 Conclusions

Since its infancy in autumn 1997, LispMe has been extended with most of the features of the R^4RS. This growth has been paralleled by a goal shift from "proof of concept" to the status of a useful tool, partly because of users response, and partly because of improvements in PalmOS and tools for overcoming memory restrictions.

LispMe is now becoming a general-purpose language for Palm development *on* the Palm. Since currently only a subset of the Palm API is implemented, extending support for Palm Pilot primitives will be the focus of LispMe's further development, especially for the various communication functions, like infrared and telephony.

Other activities planned are better conformance to R^5RS, including arbitrary precision integers and rationals, and perhaps the generation of fully stand-alone executables.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* McGraw-Hill Book Company, Cambridge, MA, 1985.

[2] D. B. Bartley and J. C. Jensen. The implementation of PC Scheme. In W. L. Scherlis and J. H. Williams, editors, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 86–93, Cambridge, Massachusetts, Aug. 1986. ACM Press.

[3] F. Bayer. Lispme home page, 2000. Contains executables, source code and documentation. Available online at `http://www.lispme.de/lispme/`.

[4] W. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation strategies for continuations. In R. C. Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 124–131, Snowbird, Utah, July 1988. ACM Press.

[5] W. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999. Extended version of [4].

[6] W. Clinger and J. Rees, editors. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[7] R. K. Dybvig and R. Hieb. A variable-arity procedural interface. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 106–115, Nice, France, June 1990. ACM Press.

[8] P. Henderson. *Functional Programming, Application and Implementation.* Prentice-Hall International, London, 1980.

[9] M. P. Jones. Hugs 1.3, the haskell user's gofer system: User manual. Technical report, Department of Computer Science, University of Nottingham, 1996. Available online at `http://www.haskell.org/hugs/`.

[10] R. Kelsey, W. Clinger, and J. Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998. Available online at `http://www.brics.dk/~hosc/11-1/`.

[11] R. A. Kelsey and J. A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–336, 1994.

[12] B. W. Kernighan and D. M. Ritchie. *The C programming language (2nd edition).* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[13] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[14] Palm Computing. *Palm OS SDK Reference.* Santa Clara, CA, 2000. Available online at `http://www.palm.com/devzone/`.

[15] F. Rouaix. Caml Light for PalmOS, 1998. An implementation of the Caml Light runtime environment for the Palm. Available online at `http://cristal.inria.fr/~rouaix/pilot/cl.html`.

[16] G. Springer and D. P. Friedman. *Scheme and the Art of Programming.* The MIT Press, Cambridge, MA, 1989.

[17] G. L. Steele Jr. and G. J. Sussman. The art of the interpreter or, the modularity complex (parts zero, one, and two). AI Memo 453, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[18] G. J. Sussman and G. L. Steele Jr. Scheme: An interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.

[19] S. Williams. *68030 Assembly Language Reference.* Addison-Wesley, Reading, MA, 1989.

[20] M. Winikoff. Palm software development — alternatives to C, 2001. An overview of language implementations for the Palm. Available online at `http://www.winikoff.net/palm/dev.html`.