Using closures for code generation

Marc Feeley
Guy Lapalme

Département d'informatique et de recherche opérationnelle (I.R.O.)
Université de Montréal
P.O.B. 6128, Station A
Montréal, Québec, H3C 3J7 (Canada)

ABSTRACT

This paper describes a new approach to compiling which is based on the extensive use of closures. In this method, a compiled expression is embodied by a closure whose application performs the evaluation of the given expression. For each primitive construct contained in the expression to compile, a closure is generated. As a whole, the compiled expression consists of a network of these closures. In a way, 'code generation' is replaced by 'closure generation'. This method, combined with an efficient closure implementation, produces compiled code which compares favorably (in execution time) with its interpreted counterpart. It can also be used to implement compilers for embedded languages and as it has been implemented in Scheme, it yields a straightforward metacircular compiler for Scheme.

Keywords: Code Generation, Closure, Compiling, Interpretation, Lisp, Scheme

1. Introduction

A computer language can be implemented in two ways: with a compiler or an interpreter. Each approach has its own advantages: usually an interpreter is easier to implement, gives better debugging tools and is more portable; a compiler can make programs run much more efficiently. This paper describes an original compiling technique which offers the advantages of an interpreter with the speed of compiled code. Code generation relies only on closure generation. Scheme [1], which is a dialect of Lisp, is used as the source and implementation language. As closures are already implemented in Scheme, it was thus possible to write an efficient and portable Scheme compiler in Scheme. The technique is not restricted to Scheme and can also be used for other languages as long as it is possible to create structures equivalent to closures.

We first give a short overview of Scheme and closures; we then describe how each primitive form of Scheme can be compiled. A complete example and improvements over the basic technique are then given. Finally, we explain how this technique can be applied to other languages and we analyse its performance.

2. Scheme and closures

In many lexically scoped dialects of Lisp, e.g. Scheme [28] [1] [2], T [24] [25] and Common Lisp [30], procedures are first class objects. They are defined by lambda-expressions of the form:

(**lambda** *formal-argument-list body*)

The formal argument list declares the variables that will contain the actual argument values when the procedure is later called and the body indicates the expression that will be evaluated to compute the result. The evaluation of a lambda-expression [†] returns a procedure that 'remembers' the current state of the

[†] lambda-expression evaluation differs from the invocation of the procedure it produces. In Common Lisp, lambda-expression evaluation is written as **(function (lambda ...))**.

environment (i.e. the set of current variable bindings). This operation is called closure. We speak of the resulting procedure as being a closure. In fact, we shall consider them to be synonymous in this paper.

Closures are a useful programming feature. They can be used to express data abstractions [1], to implement actors [27] [32] and also to provide classes and data protection [34]. The work of Atkinson and Morrison [6] discusses their usefulness in implementing modules, separate compilation and database views. Felleisen and Friedman [11] show how to incorporate modules in Scheme via syntactic extensions which make use of closures. Closures are used here to represent compiled code.

The possibility of naming computed values in order to ease their manipulation is a fundamental aspect of programming languages. This feature is provided by variables. A variable designates a location where a value can be stored. A variable's value can be accessed through the use of the variable reference and assignment operations. Certain constructs are used to create new variables and give them names. These are known as the binding constructs. In Scheme, the most fundamental binding construct is the **lambda** special form (i.e. lambda-expression). All other binding constructs, such as the **let, letrec, define** and **do** special forms, can be explained in terms of lambda-expressions. Without any loss of generality, we shall consider that the only binding construct available is the lambda-expression. The formal argument list indicates the names of the variables that are created. In accordance with lexical scoping rules, the only region of the program where these name-variable associations are effective is the body of the lambda-expression that declares them. Any use of a name, in a variable access, refers to the variable associated with this name in the innermost lambda-expression that binds the name and contains the use. Distinct variables can have the same name. However, at most one of these variables is accessible at a given time according to the previous rule.

The main operations permitted on closures are creation and invocation. Creation is obtained by the evaluation of a definition, which is a lambda-expression. The result of the evaluation is a closure which remembers the current state of the environment. Like other objects, closures can be stored in variables and data structures, passed as arguments to procedures, returned as the result of procedures, etc... Invocation is obtained by the evaluation of a combination. A combination is a list of the form (*operator operand1 ...*) whose first element is not the keyword of a special form †. The evaluation of the expression *operator* must result in a closure. The result of the invocation is the result of the evaluation of the body of the closure's definition in an environment consisting of the environment which was in effect when the closure was created augmented by new name-variable associations. Note that the environment in which the closure's body is evaluated has nothing to do with the environment in effect when the closure is invoked. This is not the case of dynamically scoped dialects of Lisp which, in a sense, allocate the formal argument variables in front of the environment in effect when the closure is invoked.

3. Overview of the compiler

The compiler is implemented as a one argument procedure named **compile**. It takes the list representation of a Scheme lambda-expression and returns a procedure that implements the given lambda-expression in the global environment. For example, the evaluation of the following two expressions (at top-level) result in equivalent procedures:

```
(lambda (x) (+ x 1))
(compile '(lambda (x) (+ x 1)))
```

The basic idea used in our compiling method is that for each primitive construct (of the source language) contained in the expression to compile, a procedure (i.e. a closure) is generated. Each generated closure has the property that when it is applied, it will perform the evaluation of the corresponding part of the original expression. Closures are used because, they can retain (through the use of closed variables) the values that are necessary to parameterize their behaviour. For example, the closure generated for the constant construct is parameterized by the value of the constant. Each closure is a procedure which accepts one argument, corresponding to the run-time environment in which the primitive construct is evaluated.

† Invocation can also be performed by using the **apply** procedure or, in Common Lisp, by the form (*funcall operator operand1 ...*).

In Scheme we have the following primitive constructs: constant, variable reference and assignment, conditional evaluation, procedure application and procedure definition (i.e. lambda-expression). Other constructs, such as **begin**, **cond**, **and**, **or**, **let**, **letrec** and others can be expressed with the primitive constructs [28] [29]. They are written as macros and are processed by the front-end of the compiler which is not discussed in this paper. Thus, we limit our discussion to these primitive constructs.

The heart of the compiler is a procedure of one argument named **gen**. It performs the recursive traversal of the expression to compile. The expression is classified according to the primitive construct and the corresponding code generation procedure is called. As stated above, these procedures return a closure that will perform the evaluation of the corresponding part of the original expression when it is applied. All subexpressions of the examined form which may need to be evaluated are compiled recursively by **gen**. This happens for the variable assignment, conditional evaluation, procedure application and procedure definition constructs. The closures generated by the compilation of the subexpressions are passed to the code generation procedure to parameterize the behaviour of the closure it generates. The compilation of an expression has the effect of constructing a network of closures which is reminiscent of the code generated for threaded languages [19] [21].

Not surprizingly, the compiler has a structure very similar to an interpreter. Both receive data structures representing expressions, classify them and recursively traverse the compound ones. The main difference is that the interpreter evaluates the expression as soon as it is recognized but the compiler generates the code that will perform the evaluation. In this latter case, the evaluation is done when the code is called. The interpreter must classify the expression each time it needs to be evaluated but the compiler does that only once at compile time. This fact explains why executing compiled code is more efficient than interpreting the source code. Moreover, a compiler can recognize at compile time special types of evaluations that can be handled more efficiently. This is not really useful when interpreting code because usually more time is spent recognizing a special case than is saved for its special handling.

We consider each primitive construct of the source language to see what must be done to compile it. The implementation of these functions depends in part on the representation of the run-time environments. For the sake of simplicity, we have chosen to represent the environment with an association list (of symbols and values) which is used like a stack. Later we will consider a more efficient approach and give the necessary modifications to be done to the code generation procedures.

3.1. Constant

The following procedure is used to implement the constant construct which obviously is independent of the environment:

```
(define (gen-cst a) (lambda (env) a))
```

The application of **gen-cst** to a specific value returns a procedure which, when it is later applied, will return this value. For example:

```
(define code1 (gen-cst 123))
(code1 '()) => 123
```

3.2. Variable reference:

Variable reference consists of fetching, from the current environment, the value which is associated with a particular variable. Accessing a variable consists of searching, via the **assq** procedure, the environment list for a pair whose symbol is the same as the symbol representing the variable to access. Variable reference is implemented by the **gen-ref** procedure, as follows:

```
(define (gen-ref a) (lambda (env) (cdr (assq a env)))) .
```

The application of **gen-ref** to a specific symbol returns a procedure which, when it is later applied (to the run-time environment), will return the value associated with the variable it represents.

3.3. Variable assignment:

Variable assignment can be dealt with in a similar fashion. This construct is parameterized by the symbol that represents the variable which is assigned to and also by the code (i.e. the closure) used to compute the value to assign. Application of the closure that corresponds to the value to assign will be performed by the closure corresponding to the assignment construct. Variable assignment is implemented by the **gen-set** procedure, as follows:

```
(define (gen-set a b) (lambda (env) (set-cdr! (assq a env) (b env))))
```

The **b** argument of this procedure corresponds to the compiled form (i.e. the closure) which computes the value to assign. Here is an example of variable reference and assignment using **gen-ref** and **gen-set**:

```
(define code2 (gen-set 'b (gen-cst 78)))
(define code3 (gen-ref 'b))
(define env '((a . 12) (b . 34) (c . 56)))
(code2 env)
env => ((a . 12) (b . 78) (c . 56))
(code3 env) => 78 .
```

3.4. Conditional evaluation

The conditional evaluation construct corresponds to a two branched **if** of the form:

(if *condition consequent alternative***)**

This construct is parameterized by the code used to compute the *condition*, *consequent* and *alternative*. Depending on the result of the application of the closure corresponding to the condition, the conditional evaluation construct will perform the application of the closure corresponding to the consequent or alternative. The **gen-tst** procedure is used to implement conditional evaluation, as follows:

```
(define (gen-tst a b c) (lambda (env) (if (a env) (b env) (c env))))
```

3.5. Procedure application

The procedure application construct corresponds to the form:

(*operator operand1 ...***)**

where *operator* is not the keyword of a special form. This construct is parameterized by the code used to compute the operator and each of the operands. Since the number of operands is not fixed, we have chosen to implement this construct using several procedures each corresponding to an application with a specific number of operands †. The compiler will select the correct one when it compiles an application. The following procedures implement procedure application:

```
(define (gen-ap0 a)     (lambda (env) ((a env))))
(define (gen-ap1 a b)   (lambda (env) ((a env) (b env))))
(define (gen-ap2 a b c) (lambda (env) ((a env) (b env) (c env))))
...
```

3.6. Procedure definition

In Scheme, as in most other dialects of Lisp, lambda-expressions of the form

(lambda *formal-argument-list body***)**

define procedures. The evaluation of this construct returns a procedure that 'remembers' the current state of the environment. When this procedure is later applied, its body will be evaluated in an environment

† We can also use a single code generation procedure based on **apply** but it adds another kind of procedure application mechanism and we want to rely only on the primitive constructs (in order to have a metacircular compiler).

consisting of the retained environment augmented by the argument bindings. This construct is parameterized by the code used to evaluate the body of the procedure and the symbols that represent each variable of the formal argument list. Environment allocation is performed by adding symbol-value pairs to the front of the environment list. Since the number of variables in the formal argument list is not fixed, this construct is implemented by several procedures each corresponding to a lambda-expression with a specific number of arguments. The compiler will select the correct one when it compiles a lambda-expression. The following procedures implement procedure definition:

```
(define (gen-pr0 a) (lambda (env)
                      (lambda () (a env))))
(define (gen-pr1 a b) (lambda (env)
                        (lambda (x) (a (cons (cons b x) env)))))
(define (gen-pr2 a b c) (lambda (env)
                          (lambda (x y) (a (cons (cons b x)
                                                 (cons (cons c y) env))))))
...
```

The **a** argument of these procedures corresponds to the compiled form (i.e. the closure) which computes the body of the procedure [†].

4. A complete example

Consider the expression:

```
(define add1 (compile '(lambda (x) (+ x 1))))
```

The evaluation of this expression compiles the expression **(lambda (x) (+ x 1))** and binds the resulting procedure to the variable **add1** . The recursive traversal of this expression by the compiler is equivalent to the following expanded form:

```
(define add1 ((gen-pr1 (gen-ap2 (gen-ref '+) (gen-ref 'x) (gen-cst 1))
                       'x)
              *glo-env*)) .
```

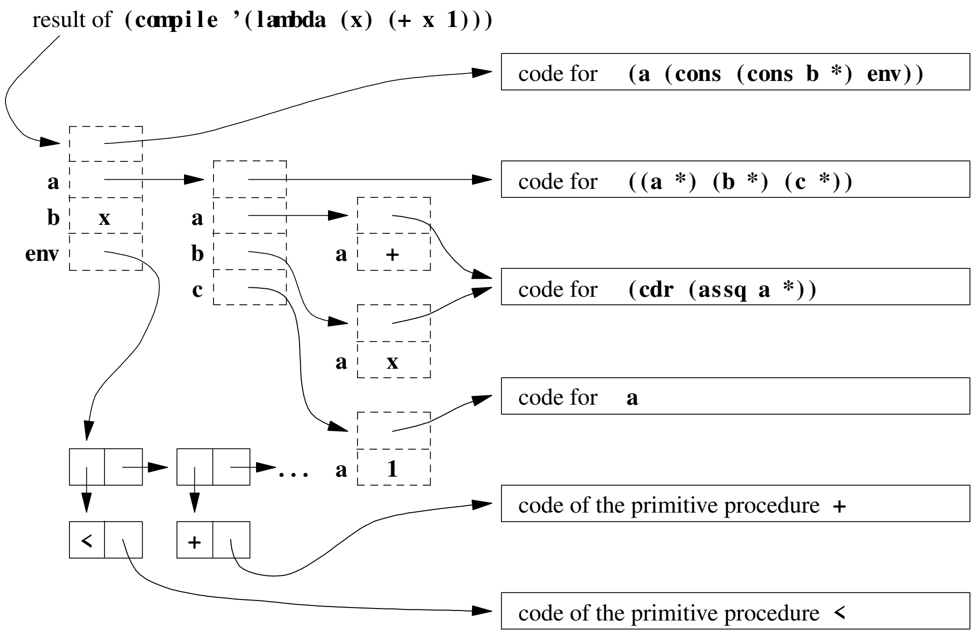
The value of the variable ***glo-env*** corresponds to the association list representing the global environment (it should at least contain the definition of the variable **+**). The code generated by the compiler is shown in Figure 1.

5. Discussion of the method

Overall, the compiler described is of the same size and complexity as an equivalent interpreter. This can be seen in the programs given in the appendices A and B. Yet programs compiled using this method execute faster than when they are interpreted. This point will be discussed in the "performance" section of this paper. The essential qualities of the interpreter, however, need not be lost. In particular, debugging capabilities and profile generation can be obtained by adding the necessary code to the body of the generated closures. A compiler switch can control the generation of those facilities thus providing the best of both worlds. Though operational and efficient, this method can be optimized to increase its performance.

Instead of using an association list to represent the run-time environment, a simple list of values can be used. The compiler computes the offset associated with a particular variable and fetches the value in the list without a search. Also, the value associated with a global variable can be a property of the symbol that represents the variable. Other representations for the environment are also possible such as a vector representing a stack or as a linked stack of heap allocated frames.

[†] Procedure definition can also be implemented by a single procedure:
(define (gen-pr a b) (lambda (env) (lambda l (a (alloc b l env)))))
The **b** argument corresponds to the formal argument list which is scanned (along with **l**) by the **alloc** procedure to perform environment allocation and verify that the procedure is called with the correct number of arguments.



Note: * represents the argument passed to the closure

Fig. 1. Sample code generated by the compiler.

Another inefficiency comes from passing the run-time environment argument to the closures generated when they are evaluated. Instead, the environment can be bound to a global variable and used freely by all the closures generated. This global 'environment' variable is saved and updated when entering a procedure and is restored to its original value upon exit [†]. Thus, the closures generated are zero argument procedures and we can expect their application to be faster.

Another source of optimization stems from the fact that certain forms are used more frequently than others (e.g. small numeric constants, procedure application using a global variable as operator, etc...). Instead of using the general procedure to generate the code corresponding to these forms, a specific procedure (which is less parameterized) can be used. For example, to generate the code for the evaluation of the constants **1** or **#!true**, the compiler can call the following procedures (instead of the more general procedure **gen-cst**) :

```
(define (gen-1)     (lambda (env) 1))
(define (gen-true) (lambda (env) #!true))
```

As a specialized code generation procedure is less parameterized than the corresponding general one, less space is needed for the code. This method is much like the classical strength reduction optimization technique [3] which consists of using less general but more efficient code instead of the usual form. In our case, we are actually constructing "custom-made" closures and tayloring them to each particular case. One must take care that the number of special cases does not grow too rapidly so this optimization is only used for the more frequent ones.

Also, space can be saved by sharing pieces of code. Instead of generating new closures for each expression (and subexpression) compiled, the compiler can reuse a previously generated closure if it corresponds to the same expression. This is possible because when a closure is called, it receives also a continuation giving where execution must go after the code in this closure is executed. Each program sharing some

---

[†] If this is done, care must be taken to preserve the tail-recursive semantics of Scheme. If the procedure's body is a procedure application (or one of the branches of a conditional evaluation containing a procedure application), environment restoration should be performed after the evaluation of the arguments of the application and before the actual jump to the called procedure.

code gives its own continuation to the closure. So instead of generating a new closure each time, the compiler can check if a closure has the same parameters as a previously generated one and give it back if this is the case. This operation can be easily integrated in the code generation fonctions. For example **gen-ref** can be rewritten as follows:

```
(define *ref* ())

(define (gen-ref a)
   (if (not (assq a *ref*))
     (set! *ref*
       (cons (cons a (lambda (env) (cdr (assq a env))))
             *ref*)))
   (cdr (assq a *ref*)))
```

In this example, a global variable ***ref*** keeps in an association list the previously created closures and their parameters. This list is checked on entering **gen-ref** and, if a corresponding closure is found, then it is given back otherwise it is added to the list and returned. This technique is similar to hash consing [5] [15] [33] but it applies to closures instead of pairs.

This method could also be integrated in the closure generation mecanism. A hash table could be used to keep track of the closures generated and checked to see if a similar closure has already been generated before creating a new one. Common sub-expressions would be automaticaly detected and only one piece of code would be generated for them.

Even if this method can lower the space requirement for the code, we can surely worry that the space needed for keeping track of the closures might be larger than the space saved. This would not be important if it was not the case that the compilation and execution phases are disjoint but Scheme is an interactive language and so the compiler must be "active" all the time. A reasonable tradeoff would be to keep the more frequent ones and ignore the rest; this could be implemented with a fixed length table where entries are entered and discarded using a "least-recently-used" policy. To simplify further, we could only keep the closures associated with the current expression.

Since the number of different kinds of closures generated by the compiler is relatively low and that the body of these closures is shared (see Figure 1), it is reasonable to hand code the body of these procedures in assembler in the same way that the primitive procedures, e.g. **car**, **cons**, **+**, are usually coded in a lower level language.

The execution of the generated code consists mainly of closure applications, thus, an efficient closure implementation will improve performance. Such a method is described in [9] [10]. In fact that method has been used in conjunction with our compilation method to implement the compiler for the MC68000 using MC68000 assembly language as the implementation language. Its performance is discussed in section 7.

The optimizations that have been described always dealt with closure generation but any other optimizations could also be used, for example: invariant removal, common subexpression elimination and expression simplifications [4] [16] [29]. Data and control flow analysis could also be computed [17] [18]. Many of these optimizations can be seen as source to source transformations as was done in the RABBIT Scheme compiler [29]. It would also be interesting to link this technique with the Orbit compiler [20] which has proven itself to be very efficient.

6. Application to other languages

This code generation can also be used for other languages than Scheme. For example, it would be possible to write a Pascal compiler given an appropriate representation for each primitive construct and given the forms of the closure to be generated. We would also need to write the primitive functions and a parser.

But a more appropriate use of this technique would be for "embedded languages" within Scheme. These languages are often designed in artificial intelligence (e.g. MICRO-PLANNER [31], CONNIVER [22], OPS5 [12] and LCF [14]) and are usually implemented with an interpreter written in Lisp. Using the technique described in this paper we could easily compile those languages and the programs would run much faster.

The technique could also be implemented in another language than Scheme as long as the language enables the creation of closures or their equivalent. Many lexically scoped dialects of Lisp (e.g. T [24] or Common Lisp [30]) give closures as a primitive construct. But it can also be used within a language like Simula 67 [7] if closures are implemented with class instances. Feeley [9] gives the details of this implementation and a small example is given in appendix E. In fact, most object oriented language allowing for the dynamic creation of objects can be used for the implementation language.

7. Performance

We have conducted some benchmarks to measure the performance of the code generated by our method. Table 1 shows the run times for the evaluation of **(fib 20)**, **(tak 18 12 6)** [13] which are the "classical tests" involving many recursive calls and integer arithmetic and **(sort '(3 1 ...))** a selection sort of 70 numbers allocating many pairs and using tail recursion for iterating. These programs can be found in the appendix D.

Relative times are shown in parenthesis. We tested our implementation on two Scheme interpreters:

- MIT C Scheme (version 6.1) [23] written in C, generating pseudo-code (S-code) and running on a SUN-2/50;
- MacScheme (version 1.11)[26] written in MC68000 assembly language, generating pseudo-code (byte-code) and running on a Macintosh Plus.

Four methods of evaluating the expressions were tried: directly in the implementation language (i.e. in MIT C Scheme or MacScheme), with the interpreter (given in appendix A), and with the optimizing and non-optimizing compilers (given in appendix B and C).

The optimizing version of the compiler has the following features:

- it represents environments as a list of values in which the variables are accessed via an offset
- the environment is kept in a global variable which is saved and restored at each procedure call
- two versions of each closure are used for taking into account the tail-recursive nature of the Scheme programs
- other small but often used improvements consist of generating special efficient closures for:
  - constants **1**,**2** and **#!null**
  - accesses to global variables and the first three local variables
  - applications with a global variable as an operator.

The optimizing compiler is given in appendix C and is about three times as long as the non-optimizing compiler.

implementation call	implementation language	optimizing compiler	non-optimizing compiler	interpreter
MIT C Scheme				
(fib 20)	190 (1.0)	540 (2.0)	1700 (8.9)	3900 (20.5)
(tak 18 12 6)	430 (1.0)	1800 (4.1)	9700 (22.5)	16000 (37.2)
(sort '(3 1 ..))	32 (1.0)	110 (3.4)	850 (26.6)	1300 (40.6)
MacScheme				
(fib 20)	53 (1.0)	120 (2.3)	190 (3.6)	440 (8.3)
(tak 18 12 6)	130 (1.0)	370 (2.8)	640 (4.9)	1400 (10.8)
(sort '(3 1 ..))	9 (1.0)	23 (2.6)	44 (4.9)	98 (10.9)

Table 1: CPU time (in seconds) for the implementation with closures in Scheme

Programs compiled by the non-optimizing compiler execute 30% to 50% faster than when they are interpreted. They are four to twenty-five times slower than the implementation language. So this is very good compared to the interpreter taking into account that the interpreter and the non-optimizing compiler are almost of the same length and complexity.

When a few simple optimisations are added, execution times drop by 40% to 70% which is only two to four times slower than the implementation language.

This technique was also embedded in a compiler for Scheme generating assembly language; it is described in [9] and table 2 gives the times for that implementation and for ExperLisp[8] which is also a native code compiler. We hand coded two functions in assembly language using the same call conventions as our compiler just to give us an idea of the "optimal" code.

call	our compiler	ExperLisp	Assembly language
(fib 20)	5.0 (1.0)	13.0 (2.6)	2.64
(tak 18 12 6)	17.0 (1.0)	44.0 (2.6)	7.72
(sort '(3 1 ...))	1.1 (1.0)	4.0 (3.6)	

Table 2: CPU Times (in seconds) with assembly code

So we can see that this technique can be used to achieve good efficiency within a "real" compiler while keeping the advantages of the interpreter. It is surprising indeed that this compiler gives programs running less than three times slower than assembly language.

8. Conclusion

We have shown that it is feasible to write a Scheme compiler in Scheme with the use of closures. The generated code takes the form of a network of closures whose application performs the desired computations. A compiler relatively independent of the target machine can be designed using this method. Optimizations to the basic method can be applied in order to gain efficiency and provide a usable system.

The method discussed can also be used to implement imbedded languages in languages that possess closures or their equivalent (e.g. Scheme, T, Common Lisp, SIMULA 67). While the compiler is portable and fairly efficient compared to the classical interpreter method, the main qualities of the interpreter (e.g. debugging capabilities, profile information, etc...) are not lost.

9. References

[1] Abelson H., Sussman G. J., Sussman J., *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, (1985).

[2] Abelson H., Adams N., Bartley D., Brooks G., Clinger W., Friedman D., Halstead R., Hanson C., Haynes C., Kohlbecker E., Oxley D., Pitman K., Rees J., Rozas B., Sussman G. J., Wand M., *The Revised Revised Report on Scheme or an UnCommon Lisp.* MIT Artificial Intelligence Memo 848, Cambridge, Massachusetts, (1985).

[3] Aho A. V., Ullman J. D., *Principles of Compiler Design.* Addison-Wesley, Reading, Massachusetts, (1977).

[4] Allen F. E., *Bibliography on program optimization.* IBM Research Report RC-5767, Technical Journal Watson Research Center, Yorktown Heights, New York, (1975).

[5] Allen J., *Anatomy of Lisp.* McGraw-Hill, New York, New York, (1978).

[6] Atkinson M. P., Morrison R., Procedures as Persistent Data Objects. *ACM Transactions on Programming Languages and Systems, 7*, no. 4, 539-559, (1985).

[7] Dahl O.-J., Myhrhaug B., Nygaard K., *SIMULA 67 Common Base Language.* Norwegian Computing Center Report 725, (1982).

[8] ExperTelligence, *ExperLisp Reference Manual*. Santa Barbara, California, (1984).

[9] Feeley M., *Deux approches a l'implantation du langage Scheme*, These de Maitrise, Document de travail #183, Département d'informatique et recherche opérationnelle, Université de Montréal,

(1986).

[10] Feeley M., Lapalme G., *Closure generation based on viewing LAMBDA as EPSILON plus COMPILE* . Submitted for publication, (1986).

[11] Felleisen M., Friedman D.P., *A closer look at export and import statements* , Computer Language, *11* , 29-37 (1986).

[12] Forgy C. L., *The OPS5 User's Manual* . Technical Report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, (1981).

[13] Gabriel R. P., Masinter L. M., Performance of Lisp Systems. *Conference record of the 1982 ACM Symposium on Lisp and Functional Programming* , Pittsburgh, Pennsylvania, 123-142, (1982).

[14] Gordon M., Milner R., Wadsworth C., *Edinburgh LCF* , Lecture Notes in Computer Science, *78* , Springer-Verlag, New York, New York, (1979).

[15] Goto E., *Monocopy and Associative Algorithms in an Extended Lisp* . University of Tokyo, Japan, (1974).

[16] Haraldsson A., *A Partial Evaluator, and Its Use for Compiling Iterative Statements in Lisp* . Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, 195-202, (1978).

[17] Hecht M. S., *Data Flow Analysis of Computer Programs* . American Elsevier, New York, New York, (1977).

[18] Jones N. D., Muchnick S. S., Flow Analysis and Optimisation of LISP-like Structures. *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages* , 244-256, San Antonio, Texas, (1979).

[19] Klint P., Interpretation Techniques. *Software-Practice and Experience* , *11* , 963-973, (1981).

[20] Kranz D., Kelsey R., Rees J., Hudak P., Philbin J., Adams N., Orbit: an Optimizing Compiler for Scheme, *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction* , (1986).

[21] Loeliger R. G., *Threaded Interpretive Languages* . Byte Books, Peterborough, New Hampshire, (1981).

[22] McDermott D., Sussman G. J., *The CONNIVER Reference Manual* . MIT Artificial Intelligence Memo 259A, Cambridge, Massachusetts, (1973).

[23] *MIT Scheme Manual* . Seventh Edition, Cambridge, Massachusetts, (1984).

[24] Rees J. A., Adams N. I., T: A Dialect of Lisp or, LAMBDA: The Ultimate Software Tool. *Conference record of the 1982 ACM Symposium on Lisp and Functional Programming* , Pittsburgh, Pennsylvania, 114-122, (1982).

[25] Rees J. A., Adams N. I., Meehan J. R., *The T Manual* . Computer Science Department, Yale University, New Haven, Connecticut, (1984).

[26] Semantic Microsystems, *MacScheme Reference Manual* . Sausalito, California, (1985).

[27] Steele G. L., *Lambda: the ultimate declarative* . MIT Artificial Intelligence Memo 379, Cambridge, Massachusetts, (1976).

[28] Steele G. L., Sussman G. J., *The Revised Report on Scheme: A Dialect of Lisp* . MIT Artificial Intelligence Memo 452, Cambridge, Massachusetts, (1978).

[29] Steele G. L., *Rabbit: a compiler for Scheme* . MIT Artificial Intelligence Memo 474, Cambridge, Massachusetts, (1978).

[30] Steele G. L., *Common Lisp: the Language* . Digital Press, (1984).

[31] Sussman G. J., Winograd T., Charniak E., *MICRO-PLANNER Reference Manual* . MIT Artificial Intelligence Memo 203A, Cambridge, Massachusetts, (1971).

[32] Sussman G. J., Steele G. L., *Scheme: An Interpreter for extended Lambda Calculus* . MIT Artificial Intelligence Memo 349, Cambridge, Massachusetts, (1975).

[33] Terashima M., *Algorithms Used in an Implementation of HLISP* . Information Sciences Laboratory Technical Report 75-03, University of Tokyo, Japan, (1975).

[34] Wand M., Continuation-Based Multiprocessing. *Conference record of the 1984 ACM Symposium on Lisp and Functional Programming* , Stanford, California, 19-28, (1980).

APPENDIX A

```
;-----------------------------------------------------------------------------
;
;                Scheme interpreter written in Scheme
;
; Note: Only the 'quote', 'set!', 'if' and 'lambda' special forms, and the
;       constant reference, variable reference and procedure application
;       constructs are handled by the interpreter.
;-----------------------------------------------------------------------------

(define (interpret expr)
  (int expr *glo-env*))

(define (int expr env)
  (cond ((symbol? expr)
         (int-ref expr env))
        ((not (pair? expr))
         (int-cst expr env))
        ((eq? (car expr) 'quote)
         (int-cst (cadr expr) env))
        ((eq? (car expr) 'set!)
         (int-set (cadr expr) (caddr expr) env))
        ((eq? (car expr) 'if)
         (int-tst (cadr expr) (caddr expr) (cadddr expr) env))
        ((eq? (car expr) 'lambda)
         (let ((p (cadr expr)))
           (cond ((null? p)
                  (int-prc0      (caddr expr) env))
                 ((symbol? p)
                  (int-prc1/rest (caddr expr) p env))
                 ((null? (cdr p))
                  (int-prc1      (caddr expr) (car p) env))
                 ((symbol? (cdr p))
                  (int-prc2/rest (caddr expr) (car p) (cdr p) env))
                 ((null? (cddr p))
                  (int-prc2      (caddr expr) (car p) (cadr p) env))
                 ((symbol? (cddr p))
                  (int-prc3/rest (caddr expr) (car p) (cadr p) (cddr p) env))
                 ((null? (cdddr p))
                  (int-prc3      (caddr expr) (car p) (cadr p) (caddr p) env))
                 (else
                  (error "too many parameters")))))
        ((null? (cdr expr))
         (int-ap0 (car expr) env))
        ((null? (cddr expr))
         (int-ap1 (car expr) (cadr expr) env))
        ((null? (cdddr expr))
         (int-ap2 (car expr) (cadr expr) (caddr expr) env))
        ((null? (cddddr expr))
         (int-ap3 (car expr) (cadr expr) (caddr expr) (cadddr expr) env))
        (else
         (error "too many arguments"))))

; --- interpretation of constants ---

(define (int-cst a env)
  a)

; --- interpretation of variable references ---

(define (int-ref a env)
  (cdr (assq a env)))
```

```
; --- interpretation of assignments ---

(define (int-set a b env)
  (set-cdr! (assq a env) (int b env)))

; --- interpretation of 'if' special form ---

(define (int-tst a b c env)
  (if (int a env) (int b env) (int c env)))

; --- interpretation of procedure application ---

(define (int-ap0 a env)
  ((int a env)))

(define (int-ap1 a b env)
  ((int a env) (int b env)))

(define (int-ap2 a b c env)
  ((int a env) (int b env) (int c env)))

(define (int-ap3 a b c d env)
  ((int a env) (int b env) (int c env) (int d env)))

; --- interpretation of 'lambda' special form ---

(define (int-prc0 a env)
  (lambda ()
    (int a env)))

(define (int-prc1 a b env)
  (lambda (x)
    (int a (cons (cons b x) env))))

(define (int-prc2 a b c env)
  (lambda (x y)
    (int a (cons (cons b x) (cons (cons c y) env)))))

(define (int-prc3 a b c d env)
  (lambda (x y z)
    (int a (cons (cons b x) (cons (cons c y) (cons (cons d z) env))))))

(define (int-prc1/rest a b env)
  (lambda x
    (int a (cons (cons b x) env))))

(define (int-prc2/rest a b c env)
  (lambda (x . y)
    (int a (cons (cons b x) (cons (cons c y) env)))))

(define (int-prc3/rest a b c d env)
  (lambda (x y . z)
    (int a (cons (cons b x) (cons (cons c y) (cons (cons d z) env))))))

; --- global variable definition ---

(define (define-global var val)
  (if (assq var *glo-env*)
    (set-cdr! (assq var *glo-env*) val)
    (begin
      (set-cdr! *glo-env* (cons (car *glo-env*) (cdr *glo-env*)))
      (set-car! *glo-env* (cons var val)))))

(define *glo-env* (list (cons 'define define-global)))
(define-global 'cons  cons )
(define-global 'car   car  )
(define-global 'cdr   cdr  )
(define-global 'null? null?)
(define-global 'not   not  )
(define-global '<     <    )
(define-global '+     +    )
(define-global '-     -    )
```

```
; --- to evaluate an expression we call the interpreter ---

(define (evaluate expr)
  (interpret expr))

;-----------------------------------------------------------------------------
;
; Sample use of the compiler:
;
; To evaluate:                  one should enter:
;
;   (define fib                   (evaluate '(define 'fib
;     (lambda (x)                              (lambda (x)
;       (if (< x 2)                              (if (< x 2)
;         x                                        x
;         (+ (fib (- x 1))                         (+ (fib (- x 1))
;            (fib (- x 2))))))                        (fib (- x 2)))))))
;
;   (fib 20)                      (evaluate '(fib 20))
```

APPENDIX B

```
;-----------------------------------------------------------------------------
;
;                Non-optimizing Scheme compiler written in Scheme
;
; Note: Only the 'quote', 'set!', 'if' and 'lambda' special forms, and the
;       constant reference, variable reference and procedure application
;       constructs are handled by the compiler.
;-----------------------------------------------------------------------------

(define (compile expr)
  ((gen expr) *glo-env*))

(define (gen expr)
  (cond ((symbol? expr)
         (gen-ref expr))
        ((not (pair? expr))
         (gen-cst expr))
        ((eq? (car expr) 'quote)
         (gen-cst (cadr expr)))
        ((eq? (car expr) 'set!)
         (gen-set (cadr expr) (gen (caddr expr))))
        ((eq? (car expr) 'if)
         (gen-tst (gen (cadr expr)) (gen (caddr expr)) (gen (cadddr expr))))
        ((eq? (car expr) 'lambda)
         (let ((p (cadr expr)))
           (cond ((null? p)
                  (gen-prc0      (gen (caddr expr))))
                 ((symbol? p)
                  (gen-prc1/rest (gen (caddr expr)) p))
                 ((null? (cdr p))
                  (gen-prc1      (gen (caddr expr)) (car p)))
                 ((symbol? (cdr p))
                  (gen-prc2/rest (gen (caddr expr)) (car p) (cdr p)))
                 ((null? (cddr p))
                  (gen-prc2      (gen (caddr expr)) (car p) (cadr p)))
                 ((symbol? (cddr p))
                  (gen-prc3/rest (gen (caddr expr)) (car p) (cadr p) (cddr p)))
                 ((null? (cdddr p))
                  (gen-prc3      (gen (caddr expr)) (car p) (cadr p) (caddr p)))
                 (else
                  (error "too many parameters")))))
        ((null? (cdr expr))
         (gen-ap0 (gen (car expr))))
        ((null? (cddr expr))
         (gen-ap1 (gen (car expr)) (gen (cadr expr))))
        ((null? (cdddr expr))
         (gen-ap2 (gen (car expr)) (gen (cadr expr)) (gen (caddr expr))))
        ((null? (cddddr expr))
         (gen-ap3 (gen (car expr)) (gen (cadr expr)) (gen (caddr expr))
```

```
                                                      (gen (cadddr expr))))
        (else
         (error "too many arguments"))))

; --- code generation for constants ---

(define (gen-cst a)
  (lambda (env) a))

; --- code generation for variable references ---

(define (gen-ref a)
  (lambda (env) (cdr (assq a env))))

; --- code generation for assignments ---

(define (gen-set a b)
  (lambda (env) (set-cdr! (assq a env) (b env))))

; --- code generation for 'if' special form ---

(define (gen-tst a b c)
  (lambda (env) (if (a env) (b env) (c env))))

; --- code generation for procedure application ---

(define (gen-ap0 a)
  (lambda (env) ((a env))))

(define (gen-ap1 a b)
  (lambda (env) ((a env) (b env))))

(define (gen-ap2 a b c)
  (lambda (env) ((a env) (b env) (c env))))

(define (gen-ap3 a b c d)
  (lambda (env) ((a env) (b env) (c env) (d env))))

; --- code generation for 'lambda' special form ---

(define (gen-prc0 a)
  (lambda (env) (lambda ()
                  (a env))))

(define (gen-prc1 a b)
  (lambda (env) (lambda (x)
                  (a (cons (cons b x) env)))))

(define (gen-prc2 a b c)
  (lambda (env) (lambda (x y)
                  (a (cons (cons b x) (cons (cons c y) env))))))

(define (gen-prc3 a b c d)
  (lambda (env) (lambda (x y z)
                  (a (cons (cons b x) (cons (cons c y) (cons (cons d z) env)))))))

(define (gen-prc1/rest a b)
  (lambda (env) (lambda x
                  (a (cons (cons b x) env)))))

(define (gen-prc2/rest a b c)
  (lambda (env) (lambda (x . y)
                  (a (cons (cons b x) (cons (cons c y) env))))))

(define (gen-prc3/rest a b c d)
  (lambda (env) (lambda (x y . z)
                  (a (cons (cons b x) (cons (cons c y) (cons (cons d z) env)))))))

; --- global variable definition ---

(define (define-global var val)
  (if (assq var *glo-env*)
```

```
    (set-cdr! (assq var *glo-env*) val)
    (begin
      (set-cdr! *glo-env* (cons (car *glo-env*) (cdr *glo-env*)))
      (set-car! *glo-env* (cons var val)))))

(define *glo-env* (list (cons 'define define-global)))
(define-global 'cons  cons )
(define-global 'car   car  )
(define-global 'cdr   cdr  )
(define-global 'null? null?)
(define-global 'not   not  )
(define-global '<     <    )
(define-global '+     +    )
(define-global '-     -    )

; --- to evaluate an expression we compile it and then call the result ---

(define (evaluate expr)
  ((compile (list 'lambda '() expr))))

;-----------------------------------------------------------------------------
;
; Sample use of the compiler:
;
; To evaluate:                   one should enter:
;
;   (define fib                    (evaluate '(define 'fib
;     (lambda (x)                                (lambda (x)
;       (if (< x 2)                                (if (< x 2)
;         x                                          x
;         (+ (fib (- x 1))                           (+ (fib (- x 1))
;            (fib (- x 2))))))                          (fib (- x 2)))))))
;
;   (fib 20)                       (evaluate '(fib 20))
```

APPENDIX C

```
;-----------------------------------------------------------------------------
;
;                Optimizing Scheme compiler written in Scheme
;
; Note: Only the 'quote', 'set!', 'if' and 'lambda' special forms, and the
;       constant reference, variable reference and procedure application
;       constructs are handled by the compiler.
;-----------------------------------------------------------------------------

(define (compile expr)
  ((gen expr '() #f)))

(define (gen expr env term)
  (cond ((symbol? expr)
         (ref (variable expr env) term))
        ((not (pair? expr))
         (cst expr term))
        ((eq? (car expr) 'quote)
         (cst (cadr expr) term))
        ((eq? (car expr) 'set!)
         (set (variable (cadr expr) env) (gen (caddr expr) env #f) term))
        ((eq? (car expr) 'if)
         (gen-tst (gen (cadr expr) env #f)
                  (gen (caddr expr) env term)
                  (gen (cadddr expr) env term)))
        ((eq? (car expr) 'lambda)
         (let ((p (cadr expr)))
           (prc p (gen (caddr expr) (allocate p env) #t) term)))
        (else
         (let ((args (map (lambda (x) (gen x env #f)) (cdr expr))))
           (let ((var (and (symbol? (car expr)) (variable (car expr) env))))
             (if (global? var)
               (app (cons var args) #t term)
               (app (cons (gen (car expr) env #f) args) #f term)))))))
```

```
(define (allocate parms env)
  (cond ((null? parms)   env)
        ((symbol? parms) (cons parms env))
        (else            (cons (car parms) (allocate (cdr parms) env)))))

(define (variable symb env)
  (let ((x (memq symb env)))
    (if x
      (- (length env) (length x))
      (begin
        (if (not (assq symb *glo-env*)) (define-global symb 'undefined))
        (assq symb *glo-env*)))))

(define (global? var)
  (pair? var))

(define (cst val term)
  (cond ((eqv? val 1)   ((if term gen-1*    gen-1   )    ))
        ((eqv? val 2)   ((if term gen-2*    gen-2   )    ))
        ((eqv? val '()) ((if term gen-null* gen-null)    ))
        (else           ((if term gen-cst*  gen-cst ) val))))

(define (ref var term)
  (cond ((global? var) ((if term gen-ref-glo*   gen-ref-glo  ) var))
        ((= var 0)     ((if term gen-ref-loc-1* gen-ref-loc-1)    ))
        ((= var 1)     ((if term gen-ref-loc-2* gen-ref-loc-2)    ))
        ((= var 2)     ((if term gen-ref-loc-3* gen-ref-loc-3)    ))
        (else          ((if term gen-ref*       gen-ref      ) var))))

(define (set var val term)
  (cond ((global? var) ((if term gen-set-glo*   gen-set-glo  ) var val))
        ((= var 0)     ((if term gen-set-loc-1* gen-set-loc-1)     val))
        ((= var 1)     ((if term gen-set-loc-2* gen-set-loc-2)     val))
        ((= var 2)     ((if term gen-set-loc-3* gen-set-loc-3)     val))
        (else          ((if term gen-set*       gen-set      ) var val))))

(define (prc parms body term)
  ((cond ((null? parms)          (if term gen-prc0*      gen-prc0     ))
         ((symbol? parms)        (if term gen-prc1/rest* gen-prc1/rest))
         ((null? (cdr parms))    (if term gen-prc1*      gen-prc1     ))
         ((symbol? (cdr parms))  (if term gen-prc2/rest* gen-prc2/rest))
         ((null? (cddr parms))   (if term gen-prc2*      gen-prc2     ))
         ((symbol? (cddr parms)) (if term gen-prc3/rest* gen-prc3/rest))
         ((null? (cdddr parms))  (if term gen-prc3*      gen-prc3     ))
         (else                   (error "too many parameters")))
   body))

(define (app vals glo term)
  (apply (case (length vals)
           ((1) (if glo (if term gen-ap0-glo* gen-ap0-glo)
                        (if term gen-ap0*     gen-ap0)))
           ((2) (if glo (if term gen-ap1-glo* gen-ap1-glo)
                        (if term gen-ap1*     gen-ap1)))
           ((3) (if glo (if term gen-ap2-glo* gen-ap2-glo)
                        (if term gen-ap2*     gen-ap2)))
           ((4) (if glo (if term gen-ap3-glo* gen-ap3-glo)
                        (if term gen-ap3*     gen-ap3)))
           (else (error "too many arguments")))
         vals))

; --- code generation procedures for non-terminal evaluations ---

; --- code generation for constants ---

(define (gen-cst a) ; any constant
  (lambda () a))

(define (gen-1)     ; for constant 1
  (lambda () 1))

(define (gen-2)     ; for constant 2
  (lambda () 2))
```

```
(define (gen-null)  ; for constant ()
  (lambda () '()))

; --- code generation for variable references ---

(define (gen-ref-glo a) ; for a global variable
  (lambda () (cdr a)))

(define (gen-ref a)     ; for any non-global variable
  (lambda () (do ((i 0 (+ i 1)) (env (cdr *env*) (cdr env)))
                 ((= i a) (car env)))))

(define (gen-ref-loc-1) ; for first local variable
  (lambda () (cadr *env*)))

(define (gen-ref-loc-2) ; for second local variable
  (lambda () (caddr *env*)))

(define (gen-ref-loc-3) ; for third local variable
  (lambda () (cadddr *env*)))

; --- code generation for assignments ---

(define (gen-set-glo a b) ; for a global variable
  (lambda () (set-cdr! a (b))))

(define (gen-set a b)     ; for any non-global variable
  (lambda () (do ((i 0 (+ i 1)) (env (cdr *env*) (cdr env)))
                 ((= i a) (set-car! env (b))))))

(define (gen-set-loc-1 a) ; for first local variable
  (lambda () (set-car! (cdr *env*) (a))))

(define (gen-set-loc-2 a) ; for second local variable
  (lambda () (set-car! (cddr *env*) (a))))

(define (gen-set-loc-3 a) ; for third local variable
  (lambda () (set-car! (cdddr *env*) (a))))

; --- code generation for 'if' special form ---

(define (gen-tst a b c)
  (lambda () (if (a) (b) (c))))

; --- code generation for procedure application ---

(define (gen-ap0 a)     ; any application (of 0 to 3 arguments)
  (lambda () ((a))))

(define (gen-ap1 a b)
  (lambda () ((a) (b))))

(define (gen-ap2 a b c)
  (lambda () ((a) (b) (c))))

(define (gen-ap3 a b c d)
  (lambda () ((a) (b) (c) (d))))

(define (gen-ap0-glo a) ; application with global variable as operator
  (lambda () ((cdr a))))

(define (gen-ap1-glo a b)
  (lambda () ((cdr a) (b))))

(define (gen-ap2-glo a b c)
  (lambda () ((cdr a) (b) (c))))

(define (gen-ap3-glo a b c d)
  (lambda () ((cdr a) (b) (c) (d))))

; --- code generation for 'lambda' special form ---
```

```
(define (gen-prc0 a)      ; no rest parameter (0 to 3 parameters)
  (lambda () (let ((def (cdr *env*)))
               (lambda ()
                 (set! *env* (cons *env* def))
                 (a)))))

(define (gen-prc1 a)
  (lambda () (let ((def (cdr *env*)))
               (lambda (x)
                 (set! *env* (cons *env* (cons x def)))
                 (a)))))

(define (gen-prc2 a)
  (lambda () (let ((def (cdr *env*)))
               (lambda (x y)
                 (set! *env* (cons *env* (cons x (cons y def))))
                 (a)))))

(define (gen-prc3 a)
  (lambda () (let ((def (cdr *env*)))
               (lambda (x y z)
                 (set! *env* (cons *env* (cons x (cons y (cons z def)))))
                 (a)))))

(define (gen-prc1/rest a) ; when a rest parameter is present
  (lambda () (let ((def (cdr *env*)))
               (lambda x
                 (set! *env* (cons *env* (cons x def)))
                 (a)))))

(define (gen-prc2/rest a)
  (lambda () (let ((def (cdr *env*)))
               (lambda (x . y)
                 (set! *env* (cons *env* (cons x (cons y def))))
                 (a)))))

(define (gen-prc3/rest a)
  (lambda () (let ((def (cdr *env*)))
               (lambda (x y . z)
                 (set! *env* (cons *env* (cons x (cons y (cons z def)))))
                 (a)))))

; --- code generation procedures for terminal evaluations ---

; --- code generation for constants ---

(define (gen-cst* a) ; any constant
  (lambda () (set! *env* (car *env*)) a))

(define (gen-1*)     ; for constant 1
  (lambda () (set! *env* (car *env*)) 1))

(define (gen-2*)     ; for constant 2
  (lambda () (set! *env* (car *env*)) 2))

(define (gen-null*)  ; for constant ()
  (lambda () (set! *env* (car *env*)) '()))

; --- code generation for variable references ---

(define (gen-ref-glo* a) ; for a global variable
  (lambda () (set! *env* (car *env*)) (cdr a)))

(define (gen-ref* a)     ; for any non-global variable
  (lambda () (do ((i 0 (+ i 1)) (env (cdr *env*) (cdr env)))
                 ((= i a) (set! *env* (car *env*)) (car env)))))

(define (gen-ref-loc-1*) ; for first local variable
  (lambda () (let ((val (cadr *env*))) (set! *env* (car *env*)) val)))

(define (gen-ref-loc-2*) ; for second local variable
  (lambda () (let ((val (caddr *env*))) (set! *env* (car *env*)) val)))
```

```
(define (gen-ref-loc-3*) ; for third local variable
  (lambda () (let ((val (cadddr *env*))) (set! *env* (car *env*)) val)))

; --- code generation for assignments ---

(define (gen-set-glo* a b) ; for a global variable
  (lambda () (set! *env* (car *env*)) (set-cdr! a (b))))

(define (gen-set* a b)     ; for any non-global variable
  (lambda () (do ((i 0 (+ i 1)) (env (cdr *env*) (cdr env)))
                 ((= i a) (set-car! env (b)) (set! *env* (car *env*))))))

(define (gen-set-loc-1* a) ; for first local variable
  (lambda () (set-car! (cdr *env*) (a)) (set! *env* (car *env*))))

(define (gen-set-loc-2* a) ; for second local variable
  (lambda () (set-car! (cddr *env*) (a)) (set! *env* (car *env*))))

(define (gen-set-loc-3* a) ; for third local variable
  (lambda () (set-car! (cdddr *env*) (a)) (set! *env* (car *env*))))

; --- code generation for procedure application ---

(define (gen-ap0* a)     ; any application (of 0 to 3 arguments)
  (lambda () (let ((w (a)))
               (set! *env* (car *env*))
               (w))))

(define (gen-ap1* a b)
  (lambda () (let ((w (a)) (x (b)))
               (set! *env* (car *env*))
               (w x))))

(define (gen-ap2* a b c)
  (lambda () (let ((w (a)) (x (b)) (y (c)))
               (set! *env* (car *env*))
               (w x y))))

(define (gen-ap3* a b c d)
  (lambda () (let ((w (a)) (x (b)) (y (c)) (z (d)))
               (set! *env* (car *env*))
               (w x y z))))

(define (gen-ap0-glo* a) ; application with global variable as operator
  (lambda ()
    (set! *env* (car *env*))
    ((cdr a))))

(define (gen-ap1-glo* a b)
  (lambda () (let ((x (b)))
               (set! *env* (car *env*))
               ((cdr a) x))))

(define (gen-ap2-glo* a b c)
  (lambda () (let ((x (b)) (y (c)))
               (set! *env* (car *env*))
               ((cdr a) x y))))

(define (gen-ap3-glo* a b c d)
  (lambda () (let ((x (b)) (y (c)) (z (d)))
               (set! *env* (car *env*))
               ((cdr a) x y z))))

; --- code generation for 'lambda' special form ---

(define (gen-prc0* a)      ; no rest parameter (0 to 3 parameters)
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda ()
                 (set! *env* (cons *env* def))
                 (a)))))
```

```
(define (gen-prc1* a)
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda (x)
                 (set! *env* (cons *env* (cons x def)))
                 (a)))))

(define (gen-prc2* a)
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda (x y)
                 (set! *env* (cons *env* (cons x (cons y def))))
                 (a)))))

(define (gen-prc3* a)
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda (x y z)
                 (set! *env* (cons *env* (cons x (cons y (cons z def)))))
                 (a)))))

(define (gen-prc1/rest* a) ; when a rest parameter is present
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda x
                 (set! *env* (cons *env* (cons x def)))
                 (a)))))

(define (gen-prc2/rest* a)
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda (x . y)
                 (set! *env* (cons *env* (cons x (cons y def))))
                 (a)))))

(define (gen-prc3/rest* a)
  (lambda () (let ((def (cdr *env*)))
               (set! *env* (car *env*))
               (lambda (x y . z)
                 (set! *env* (cons *env* (cons x (cons y (cons z def)))))
                 (a)))))

; --- global variable definition ---

(define (define-global var val)
  (if (assq var *glo-env*)
    (set-cdr! (assq var *glo-env*) val)
    (set! *glo-env* (cons (cons var val) *glo-env*))))

(define *glo-env* (list (cons 'define define-global)))
(define-global 'cons  cons )
(define-global 'car   car  )
(define-global 'cdr   cdr  )
(define-global 'null? null?)
(define-global 'not   not  )
(define-global '<     <    )
(define-global '+     +    )
(define-global '-     -    )

; --- to evaluate an expression we compile it and then call the result ---

(define (evaluate expr)
  ((compile (list 'lambda '() expr))))

(define *env* '(dummy)) ; current environment

;------------------------------------------------------------------------------
;
; Sample use of the compiler:
;
; To evaluate:                   one should enter:
;
```

```
;   (define fib                      (evaluate '(define 'fib
;     (lambda (x)                                 (lambda (x)
;       (if (< x 2)                                 (if (< x 2)
;         x                                           x
;         (+ (fib (- x 1))                            (+ (fib (- x 1))
;            (fib (- x 2))))))                           (fib (- x 2)))))))
;
;   (fib 20)                         (evaluate '(fib 20))
```

APPENDIX D

```
; fib

(define fib
  (lambda (x)
    (if (< x 2)
      x
      (+ (fib (- x 1))
         (fib (- x 2))))))

(fib 20)

; tak

(define tak
  (lambda (x y z)
    (if (not (< y x))
      z
      (tak (tak (- x 1) y z)
           (tak (- y 1) z x)
           (tak (- z 1) x y)))))

(tak 18 12 6)

; sort

(define sort
  (lambda (lst)
    (if (null? lst)
      '()
      (sort-aux (cdr lst) '() (car lst)))))

(define sort-aux
  (lambda (lst rest min)
    (if (null? lst)
      (cons min (sort rest))
      (if (< (car lst) min)
        (sort-aux (cdr lst) (cons min rest) (car lst))
        (sort-aux (cdr lst) (cons (car lst) rest) min)))))

(sort '(3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 4
        3 3 8 3 2 7 9 5 0 2 8     2 7 1 8 2 8 1 8 2 8 4
        5 9 0 4 5 2 3 5 3 6 0 2 8 7 4 7 1 3 5 2 6 6 2 4)) ; pi and e
```

APPENDIX E

```
!----------------------------------------------------------------------------;
!                                                                            ;
!                  Code generation example in SIMULA 67                      ;
!                                                                            ;
!----------------------------------------------------------------------------;

BEGIN

  ! --- data types (environments, integers, booleans and functions) --- ;

  CLASS DATA;
  BEGIN
  END;
```

```
DATA CLASS ENVIRONMENT(VAR,VAL,ENV);
  CHARACTER VAR;
  REF(DATA) VAL;
  REF(ENVIRONMENT) ENV;
BEGIN

  REF(DATA) PROCEDURE LOOKUP(X); CHARACTER X;
    LOOKUP :- IF X = VAR THEN VAL ELSE ENV.LOOKUP(X);

  PROCEDURE ASSIGN(X,Y); CHARACTER X; REF(DATA) Y;
    IF X = VAR THEN VAL :- Y ELSE ENV.ASSIGN(X,Y);

END;

DATA CLASS INTG(VAL); INTEGER VAL;
BEGIN
END;

DATA CLASS BOOL(VAL); BOOLEAN VAL;
BEGIN
END;

DATA CLASS FUNC;
  VIRTUAL :
    PROCEDURE APPLY1 IS
      REF(DATA) PROCEDURE APPLY1(X); REF(DATA) X;;
    PROCEDURE APPLY2 IS
      REF(DATA) PROCEDURE APPLY2(X,Y); REF(DATA) X, Y;;
BEGIN
END;

! --- code generation procedures --- ;

FUNC CLASS GEN_CST(A); REF(DATA) A;
BEGIN
  REF(DATA) PROCEDURE APPLY1(ENV); REF(ENVIRONMENT) ENV;
    APPLY1 :- A;
END;

FUNC CLASS GEN_REF(A); CHARACTER A;
BEGIN
  REF(DATA) PROCEDURE APPLY1(ENV); REF(ENVIRONMENT) ENV;
    APPLY1 :- ENV.LOOKUP(A);
END;

FUNC CLASS GEN_TST(A,B,C); REF(FUNC) A,B,C;
BEGIN
  REF(DATA) PROCEDURE APPLY1(ENV); REF(ENVIRONMENT) ENV;
    APPLY1 :- IF A.APPLY1(ENV) QUA BOOL.VAL THEN B.APPLY1(ENV)
                                            ELSE C.APPLY1(ENV);
END;

FUNC CLASS GEN_AP1(A,B); REF(FUNC) A,B;
BEGIN
  REF(DATA) PROCEDURE APPLY1(ENV); REF(ENVIRONMENT) ENV;
    APPLY1 :- A.APPLY1(ENV) QUA FUNC.APPLY1( B.APPLY1(ENV) );
END;

FUNC CLASS GEN_AP2(A,B,C); REF(FUNC) A,B,C;
BEGIN
  REF(DATA) PROCEDURE APPLY1(ENV); REF(ENVIRONMENT) ENV;
    APPLY1 :- A.APPLY1(ENV) QUA FUNC.APPLY2( B.APPLY1(ENV),
                                             C.APPLY1(ENV) );
END;

FUNC CLASS GEN_FN1(A,B); REF(FUNC) A; CHARACTER B;
BEGIN
  REF(DATA) PROCEDURE APPLY1(ENV); REF(ENVIRONMENT) ENV;
    APPLY1 :- NEW FN1( A, B, ENV );
END;

GEN_FN1 CLASS FN1(ENV); REF(ENVIRONMENT) ENV;
```

```
  BEGIN
    REF(DATA) PROCEDURE APPLY1(X); REF(DATA) X;
      APPLY1 :- A.APPLY1( NEW ENVIRONMENT(B,X,ENV) );
  END;

  ! --- predefined functions --- ;

  FUNC CLASS SMALLER;
  BEGIN
    REF(DATA) PROCEDURE APPLY2(X,Y); REF(INTG) X,Y;
      APPLY2 :- NEW BOOL( X.VAL < Y.VAL );
  END;

  FUNC CLASS ADD;
  BEGIN
    REF(DATA) PROCEDURE APPLY2(X,Y); REF(INTG) X,Y;
      APPLY2 :- NEW INTG( X.VAL + Y.VAL );
  END;

  FUNC CLASS SUB;
  BEGIN
    REF(DATA) PROCEDURE APPLY2(X,Y); REF(INTG) X,Y;
      APPLY2 :- NEW INTG( X.VAL - Y.VAL );
  END;

  REF(ENVIRONMENT) GLO_ENV;  ! global environment ;

! --- initialize global environment --- ;

GLO_ENV :- NEW ENVIRONMENT( 'F', NONE,
           NEW ENVIRONMENT( '<', NEW SMALLER,
           NEW ENVIRONMENT( '+', NEW ADD,
           NEW ENVIRONMENT( '-', NEW SUB, NONE ) ) ) );

! --- code generation for the fibonacci function:            --- ;
! --- (lambda (x) (if (< x 2) x (+ (f (- x 1)) (f (- x 2))))) --- ;

GLO_ENV.ASSIGN( 'F',
        NEW GEN_FN1(
            NEW GEN_TST(
                NEW GEN_AP2(
                    NEW GEN_REF( '<' ),
                    NEW GEN_REF( 'X' ),
                    NEW GEN_CST( NEW INTG(2) ) ),
                NEW GEN_REF( 'X' ),
                NEW GEN_AP2(
                    NEW GEN_REF( '+' ),
                    NEW GEN_AP1(
                        NEW GEN_REF( 'F' ),
                        NEW GEN_AP2(
                            NEW GEN_REF( '-' ),
                            NEW GEN_REF( 'X' ),
                            NEW GEN_CST( NEW INTG(1) ) ) ),
                    NEW GEN_AP1(
                        NEW GEN_REF( 'F' ),
                        NEW GEN_AP2(
                            NEW GEN_REF( '-' ),
                            NEW GEN_REF( 'X' ),
                            NEW GEN_CST( NEW INTG(2) ) ) ) ) ) ),
            'X' ).APPLY1( GLO_ENV ) );

! --- computation of fib(10) --- ;

OUTINT( GLO_ENV.LOOKUP('F') QUA
        FUNC.APPLY1( NEW INTG(10) ) QUA
        INTG.VAL, 4 );

OUTIMAGE;   ! writes: 55 ;

END;
```