# A Closer Look at Export and Import Statements

By

Matthias Felleisen
and
Daniel P. Friedman

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 184

A CLOSER LOOK AT
EXPORT AND IMPORT STATEMENTS

by

Matthias Felleisen & Daniel P. Friedman

November, 1985

This report to appear in the journal, *Computer Languages.*

# A Closer Look at Export and Import Statements

Matthias Felleisen, Daniel P. Friedman

Computer Science Department
Lindley Hall 101
Indiana University
Bloomington, Indiana 47405

*Abstract*

Export and import statements can be implemented as syntactic extensions. We first define their intuitive semantics in terms of Scheme programs. Then we show how export and import can be improved to allow for an arbitrary load-sequence of modules and to handle dynamic extensions of modules.

*Import Export Modular Programming*

## 1. Introduction

Modular programming has become an indispensable tool for the design of large software systems. Language designers have realized this fact and have incorporated facilities for modular programming. Simula 67 ([4]) started this trend with its class concept. Other languages, including Modula ([13]), Ada ([15]), Modula-2 ([14]), Y ([6]), and ML ([7]), have followed. Most of these languages support at least features for the declaration of modules, export, and import.

Scheme ([3],[12]), a modern programming language combining imperative and functional aspects, does not provide modules per se. As Scheme is an interactive language, one could think, on the other hand, that it is only suitable for programming-in-the-small, where modules are not required. However, we believe that modular programming techniques should be used for small programs for just the same reasons they are used for larger ones.

The conventional way to provide modular programming features in a language is to change the compiler so that it recognizes the appropriate statements. In Scheme we have an alternative available. Scheme has a small, but powerful set of semantic constructs which can be used to program new language facilities. In Scheme 84 ([5]) there is also a syntax preprocessor ([8]) which allows for syntactically abstracting new constructs. Put differently, if we wish to extend Scheme 84 with a

This material is based on work supported by the National Science Foundation under grants DCR 85-01277 and MCS 83-03325.

new construct, we first define an operational semantics, and then we syntactically abstract from the implementation details. As observed elsewhere ([2]), definitions of this kind also provide a starting point for a denotational semantics for these statements.

We will demonstrate in this paper how to incorporate modules into Scheme 84 with this technique. In the course of the presentation, three different versions of import will be defined. We call them import-by-value, import-by-need, and import-by-name. The latter turns out to be the most useful one, though it seems that no other language offers a similar construct. Export can also be defined in two different ways. The enhanced version allows for dynamic changes to a module and can thus be helpful for program debugging in conjunction with import-by-name. Again, most languages do not exploit this more flexible form of export.

The next section gives a brief description of Scheme 84 and its syntax preprocessor. In Section 3 we present the syntactic extensions for export, import, and for defining modules. In the last section we discuss more general aspects of the new facilities and an unresolved problem.

## 2. Scheme

Scheme is a descendant of Lisp and the λ-calculus. It is lexically scoped, and applicative order with functions as first-class values. This will be important for the design of our export mechanisms.

Figure 1 defines a subset of Scheme 84's syntax that is sufficient for the purposes of this paper. The superscripts $^*$ and $^+$ denote 0 or more, and 1 or more occurrences of the preceding form, respectively. *Square brackets are interchangeable with parentheses*, and are used for readability. `quote` expressions return the indicated literal object; `'⟨object⟩` is equivalent to `(quote ⟨object⟩)`. `lambda` expressions evaluate to first-class functional objects that statically bind their identifiers when invoked; the variation `(lambda ⟨identifier⟩ ⟨expression⟩`$^+$`)` binds `⟨identifier⟩` to the list of *all* actual parameters. `let` makes lexical bindings, and `letrec` makes several (mutually) recursive lexical bindings. `block` is like `let` except that the identifiers do not get initialized to a value, *i.e.*, it is equivalent to a block in Algol 60. `rec` establishes one recursive lexical binding. `if` evaluates its second expression if the first is true, and the third otherwise; `when` only evaluates the second expression if the first one is true. `select` evaluates the tag expression, and then returns the value of the first expression whose corresponding symbol matches the tag. If the last symbol is `else`, it always matches. `lambda`, `let`, `letrec`, `block`, and `select` evaluate the expression lists from left to right and return the value of the last one. `set!` changes a binding of a lexical identifier; `define` establishes a global definition. An application evaluates its expressions (in some unspecified order) and applies the functional value of the first expression to the values of the remaining expressions.

Scheme 84 provides a syntax preprocessor ([8]) that examines the first object in each application. If the object is not a keyword, then it is assumed that the expression is an application; if the object is a syntactic extension keyword, then the

```
⟨expression⟩ ::=
    ⟨constant⟩
  | ⟨identifier⟩
  | ⟨syntactic extension⟩
  | (quote ⟨object⟩)
  | (lambda (⟨identifier⟩*) ⟨expression⟩+)
  | (lambda ⟨identifier⟩ ⟨expression⟩+)
  | (let ([⟨identifier⟩ ⟨value⟩]*) ⟨expression⟩+)
  | (letrec ([⟨identifier⟩ ⟨value⟩]*) ⟨expression⟩+)
  | (block (⟨identifier⟩*) ⟨expression⟩+)
  | (rec ⟨identifier⟩ ⟨expression⟩)
  | (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)
  | (when ⟨expression⟩ ⟨expression⟩)
  | (select ⟨tag⟩ [⟨symbol⟩ ⟨expression⟩+]+)
  | (set! ⟨identifier⟩ ⟨expression⟩)
  | (define ⟨identifier⟩ ⟨expression⟩)
  | ⟨application⟩
⟨value⟩, ⟨tag⟩, ⟨function⟩ ::= ⟨expression⟩
⟨syntactic extension⟩ ::= (⟨keyword⟩ ⟨object⟩*)
⟨application⟩ ::= (⟨function⟩ ⟨expression⟩*)
```

Figure 1 : Syntax of a Scheme 84 Subset

expression is replaced by an appropriately transformed expression. For an example, consider Scheme's **let** statement. With the syntax preprocessor it can be defined by the following equivalence:

```
(syntax
   (let ([id val] ...) exp ...)
   ((lambda (id ...) exp ...) val ...))
```

The syntactic pattern says that a **let**-expression is a list with **let** in its first position, a list of pairs "**([id val] ...)**" in the second followed by expressions "**exp ...**". The second line, or semantic pattern, specifies that a **let**-expression is equivalent to an application where "**exp ...**" are the function bodies, "**id ...**" are the formal parameters, and "**val ...**" are the actual arguments. The ellipsis "..." is a keyword for the syntax preprocessor. It specifies 0 or more occurrences of the preceding prototype.

The extent to which the syntax preprocessor enables a programmer to create new syntactic forms gives us the right to call this activity *syntactic programming.* It is interesting to note, that by virtue of the preprocessor Scheme's core could theoretically be reduced to seven types of expressions : ⟨**constant**⟩, ⟨**identifier**⟩, **quote**, **if**, **lambda**, and **set!** expressions, and ⟨**application**⟩.

We also require seven primitive functions. `cons`, `car`, and `cdr` are the conventional Lisp list constructor and selectors, respectively. `list` is also a list constructor: it takes a list of expressions and returns a list of their values. `set-cdr!` takes a (non-empty) list and an arbitrary object and replaces the cdr of the list with the object by a side-effect. `eq?` returns true if its arguments are the same literal objects. `null?` returns true if its argument is the null list (nil). `apply` takes a function and a list of arguments and returns the value of the application of the function to the list of arguments.

A typical example of a Scheme function is the following definition of the function `assq-sf`, which will be useful later in the paper:

```
(define assq-sf
   (lambda (a l succ fail)
      ((rec loop
          (lambda (l)
             (if (null? l) (fail)
                 (if (eq? a (car (car l))) (succ (car l))
                     (loop (cdr l))))))
       l)))
```

Intuitively, `assq-sf` searches an association list for a pair whose first component is eq? to its first argument and then applies the function `succ` to that pair; if unsuccessful, it invokes the function `fail` on no arguments.

## 3. Modular Programming Features for Scheme

In general a module is just a collection of function-, constant-, and variable-declarations. Some of these declarations are exported so that other modules may import and use them; the rest of the declarations remains hidden. In other words a module is like a block, part of whose definitions are visible to the outside world. The collection of exported items is usually referred to via the module identifier.

### 3.1 Export

The equivalent of a block in Scheme is a `let` or `letrec` expression. Each establishes a collection of new identifier-value bindings, *i.e.*, environments, and then each evaluates the statements in its body. The value of the last is the value of the entire expression. If a block is to represent a module, and a module in turn stands for the collection of exported items, it is quite natural to have an export expression as the last statement of a module–block. If we then bind the result of the export expression to the module's identifier, other modules and blocks may later refer to the exported items via the module identifier—just as desired. Figure 2 shows a typical module. Five functions are declared and four are exported.

The result of an export expression must represent the visible part of the module's declarations. Since declarations can be regarded as mappings from identifiers

```
(define stack-adt
   (letrec ([init (lambda () '(newstack))]
            [chk  (lambda (stk)
                     (when (eq? 'newstack (car stk))
                           (error _ _ _ )))]
            [push (lambda (ele stk) (cons ele stk))]
            [pop  (lambda (stk) (chk stk) (cdr stk))]
            [top  (lambda (stk) (chk stk) (car stk))])
      (export init push pop top)))
```

Figure 2: A Simple Module Declaration

to values, we have chosen to represent a module by a function object of the form[1]:

```
(lambda (msg)
   (select msg
      [name1 value1]
      [name2 value2]
      - - -
      [else  (error "bad export " msg)]))
```

where `value1, value2`, etc. refer to the values of the exported items `name1, name2`, etc. As this expression is evaluated in the module's environment, we can reference `value1` as `name1`, etc. Hence, the expression can be textually abstracted by the following syntax declaration:

```
(syntax
   (export name ...)
   (lambda (msg)
      (select msg
         [name name]
         ...
         [else  (error "bad export " msg)])))
```

Figure 3 shows the expanded form of the module in figure 2. The reader should convince himself that the identifier `stack-adt` is really bound to the approriate function.

[1] An alternative is to represent these finite functions with tables or association lists.

```
(define stack-adt
   (letrec ([init (lambda () '(newstack))]
            [chk  (lambda (stk)
                     (when (eq? 'newstack (car stk))
                           (error _ _ _ )))]
            [push (lambda (ele stk) (cons ele stk))]
            [pop  (lambda (stk) (chk stk) (cdr stk))]
            [top  (lambda (stk) (chk stk) (car stk))])
     (lambda (msg)
        (select msg
          [init init]
          [push push]
          [pop  pop]
          [top  top]
          [else (error "bad export " msg)]))))
```

Figure 3: An Expanded Module Definition

This first version of the export function is well-known ([1]). It suffices in most cases and is equivalent to the corresponding statements in traditional languages. Sometimes, however, more flexibility is needed. For example, it is often the case that during the debugging phase of a program a function in some module has to be added or changed. If the function to be changed is exported, a more dynamic export function can be used to alter the definition without re-loading and re-compiling the complete module.

The new version of export collects all the exported items in an association list, `table`. It then defines a function `*addf`, which can change or extend the bindings in `table`. The function is immediately used to insert itself into the table. The new syntactic abstraction is:

```
(syntax
   (export name ...)
   (let ([table (list (cons 'name name) ...)])
     (let ([*addf (lambda (newf newval)
                    (assq-sf newf table
                      (lambda (p) (set-cdr! p newval))
                      (lambda ()
                         (set! table
                               (cons (cons newf newval) table)))))])
       (*addf '*addf *addf)
       (lambda (msg)
          (assq-sf msg table cdr
             (lambda () (error "bad export " msg)))))))
```

As in the first definition of export, the result of an export expression is a closure which takes an identifier, looks up its binding—this time in a table—and returns the associated value. With `*addf` we can now dynamically extend and redefine bindings in modules. To do this, the user must pass the function name and the function body to the `*addf` function of the module which he wants to change. The following syntactic form hides the details:

```
(syntax
   (add-function-to-module mod-name func-name func-body)
   ((mod-name '*addf) 'func-name func-body))
```

Unfortunately, the technique does not work if the new function definition needs access to a lexical variable in the definition of the module. For example, we cannot dynamically add a function to a module which needs to side-effect a state-variable of that module. Neither can we add a function that calls locally defined functions.

## 3.2 Import

Import is a converse of export. It extends an environment to include new bindings. The values for these extensions come from exporting modules.

In Scheme the standard way to extend a lexical scope is the `let` statement. It lexically binds values to names during the evaluation of its body. In the case of an import from modules, values are obtained by sending their export names to the modules. A first approach towards an import statement could thus be:

```
(syntax
   (import from-module (name ...) exp ...)
   (let ([name (from-module 'name)] ...) exp ...))
```

This technique works for many programs but it is inflexible about the naming of imported items. Often the same name is used in different modules for different things, *e.g.*, `init` for a function that initializes stacks, queues, etc. An import statement should therefore at least allow for renaming of imported functions. This can easily be done by mimicking the let statement:

```
(syntax
   (import-by-value from-module ([im-name ex-name] ...) exp ...)
   (let ([im-name (from-module 'ex-name)] ...) exp ...))
```

The new version of import binds the objects called `ex-name`, etc. from the exporting module to the names `im-name`, etc., respectively. An example of its use is shown in figure 4.

Import, as defined above, determines the bindings for its blocks once and for all. It does this at the beginning of its evaluation, and is therefore called import-by-value. It immediately follows that a programmer would not be able to use the dynamic extension facility as defined in the preceding section. Another problem associated with this first version of import is more serious. In an interactive programming environment like Scheme one should not be forced to load and compile modules in a certain order. Also, one would like to have the freedom of redefining modules without having to re-link the complete program. Unfortunately, import-by-value is incapable of allowing for either of these possibilities.

```
(define queue-of-stacks
   (import stack ([NewS init]
                  [Sfront top])
      (import queue ([NewQ init]
                     [Qfront front]
                     [add enq])
         (let ([search   _ _ _ ]
               [ins-item _ _ _ ])
            (export search ins-item)))))
```

Figure 4: Re-naming Import from Several Modules

The problem with the current version of import is that it immediately evaluates the value which is imported. This determines the binding for the rest of the execution. However, if imported values were restricted to functions, then we could delay the real import until the function was needed. This can be accomplished by binding another, knowledgeable function to the import names. Let us assume for the moment that we are only dealing with importing and exporting functions. Furthermore, let us only consider modules that do not use imported functions at the time they are defined. We will show in the next section how to remove this latter restriction. The impact of the former one will be discussed in the last section.

An improved version of import delays the import, *i.e.*, the evaluation of `(from-module 'ex-name)` until the function named `im-name` is needed. Evaluation of an expression can be delayed by packaging it up in a function. As it is unknown how many arguments the imported function takes, the delay function must take an indefinite number of arguments. For this purpose Scheme provides a form of lambda which binds the list of all its arguments to a single parameter name. Having obtained the list of arguments for the imported function, the delay function evaluates `(from-module 'ex-name)` and applies the result to its list of arguments with the help of `apply`:

```
(syntax
   (import-by-name from-module ([im-name ex-name] ...) exp ...)
   (let ([im-name (lambda args (apply (from-module 'ex-name) args))] ...)
      exp ...))
```

Import-by-name imports a function whenever this function is used just as call-by-name evaluates the form associated with a parameter whenever the corresponding identifier is referenced. And just like call-by-name, import-by-name is an expensive operation. Whenever the function is invoked, it must obtain the imported function by evaluating `(from-module 'ex-name)`. This is probably reasonable for the debugging phase because a programmer can make use of this extra flexibility. However, the price is in general prohibitive. Frequently, we can assume that if a program works correctly, it does not change its user-defined functions—at least not the exported ones. Based on this assumption, import-by-name can be improved. After the first import the delay function can redefine itself to the result of the import for all future references. More precisely, the delay function redefines the binding for the imported function to the result of the import. It is interesting that because of this need for redefinition of the delay function, the binding has to be recursive, although the function does not call itself recursively. We must finally guarantee that it returns the proper result on its first invocation. The definition of import-by-need summarizes this optimization:

```
(syntax
   (import-by-need from-module ([im-name ex-name] ...) exp ...)
   (letrec ([im-name (lambda 1st-time-args
                        (set! im-name (from-module 'ex-name))
                        (apply im-name 1st-time-args))]
            ...)
      exp ...))
```

Import-by-need resembles call-by-need in that it evaluates its import expression only the first time it is used determining the result for the remainder of the program execution.

Both, import-by-name and import-by-need, allow the programmer to load and compile modules in an arbitrary order as long as these modules satisfy the above mentioned restrictions. Furthermore, import-by-name offers two advantages. First, a programmer can redefine a module after a test and re-run the program without having to re-link all the other modules. And second, import-by-name can make use of the enhanced export technique which allows for dynamically extending modules with functions.

But again, both techniques do not generalize to modules which use imported functions at definition time. Consider the example in figure 5. Here the state variable `msgQ` is immediately initialized with the imported function `init`. If `queue` is not defined at the time `msg-handler` is defined the evaluation results in an error. No import technique can solve this problem in a reasonable way. The problem is caused by our usage of define for module declarations. We obviously need a special defining form for modules.

```
(define msg-handler
   (import queue ([init init][addm enq][first front][get deq])
      (let ([msgQ (init)])
         (let ([send! (lambda (msg)
                        (set! msgQ (addm msg msgQ))
                        'ok)]
               [get!  (lambda ()
                        (let ([fst-element (first msgQ)])
                           (set! msgQ (get msgQ))
                           fst-element))])
            (export send! get! first)))))
```

Figure 5: Module with State Variables

## 3.3 Defining Modules

For the definition of modules we have so far relied on Scheme's generally available definition techniques, *i.e.*, `define` and `letrec`. As we have seen in the preceding section, `define` is too weak to allow a most general import/export technique. The same is true for `letrec`. Consider the example in figure 6 which displays the declaration of two local modules. The way `letrec` evaluates its expressions, `procQ` would try to import functions from `process` in an environment where `process` is not yet bound to a module.

```
(letrec ([process (let ([time-pt  _ _ _ ]
                        [msgQ-pt  _ _ _ ]
                        [state-pt _ _ _ ]
                        [name-pt  _ _ _ ]
                        [init     _ _ _ ])
                    (export time-pt msgQ-pt state-pt name-pt init))]
         [procQ   (import queue ([initQ init][addQ add][remQ remove])
                    (import process ([Pname name-pt][initP init])
                      (let ([init _ _ _ ]
                            [addP _ _ _ ] _ _ _ )
                        (export init addP _ _ _ ))))]
         _ _ _ )
  _ _ _ )
```

Figure 6: Local Modules

The major problem with `define` and `letrec` is immediate evaluation. In both cases the result of the delayed expression is an export function. For definitions with `define` we delay the evaluation of the module body, or in other words, the delay function is wrapped around the body. The delay function assures that the first call to the export function works correctly, *i.e.*, it takes the message sent to the module and passes it on. The definition of `def-module` parallels `import-by-need`.

```
(syntax
  (def-module name module-exp)
  (define name
    (lambda (msg)
      (set! name module-exp)
      (name msg))))
```

The problem of `letrec` can be solved by a simple redefinition. `letrec` is equivalent to:

```
(syntax
  (letrec ([id val] ...) exp ...)
  (block (id ...) (set! id val) ...  exp ...)).
```

So we just have to replace the `set!` by `def-module` in order to make `letrec` work for module definitions:

```
(syntax
   (m-letrec ([id val] ...) exp)
   (block (id ...) (def-module id val) ... exp ...)).
```

But `letrec`'s problem is also present when mutually recursive closures—not necessarily representing modules—are defined. Consider for example the expression `(letrec ([f g][g cons]) g)`. Depending on the sequencing of `[f g]` and `[g cons]` this program will result in an error. Therefore, we define a general `letrec` for closures. The difference between the new `c-letrec` and `m-letrec` is small. As we do not know how many arguments a function declared in `c-letrec` takes, `c-letrec` uses a `lambda`-form which binds all the arguments to one single parameter:

```
(syntax
   (c-letrec ([name body] ...) exp ...)
   (block (name ...)
      (set! name (lambda x (set! name body) (apply name x)))
      ...
      exp ...))
```

The expansion of `c-letrec` first establishes a block with the new variables. Then each variable is set to its corresponding delay function. As this is done in the scope of the newly created environment, the bindings can be mutually recursive. When the functions are finally invoked, they reset themselves to the intended function. The names get set and reset only once.

## 4. Discussion

In the preceding sections we have shown how modular programming facilities can be implemented as syntactic extensions to Scheme. This observation is not new. Reynolds ([11]) had already pointed out that Simula 67's classes are syntactic sugar, and even Landin had foreseen this with his explanation of own variables in Algol 60 ([9]). Furthermore, we have shown several alternatives for the implementation and semantics of export and import statements for modules.

Whereas export has two rather straightforward implementations, import can be realized in at least three different ways. Import-by-value is the easiest one to implement but it restricts the programmer in many ways. He cannot load modules in an arbitrary order, and it is also impossible to interactively debug modules. As our proposal is aimed towards an interactive language, this is indeed a severe restriction. Interactive languages require late binding for program development. We do this by delaying the import until the function is used. Then we know the function is needed, and it must then be defined. The resulting import-by-name has the desired attributes. It allows for an arbitrary load order and makes debugging in many cases easier. Import-by-name is, however, expensive. Fortunately it can be replaced by import-by-need under rather weak assumptions. We finally want to point out that our import and export statements are programs, *i.e.*, that they are executed at run-time. This differentiates them from their syntactic counterparts which are used at compile- and link-time.

Scheme's `define` and `letrec` turned out to be insufficient for the definition of modules: they evaluate their expressions too early. The new forms, `def-module` and `c-letrec`, delay the evaluation until the newly defined variables are dereferenced. Metaphorically, they are assignment-by-need and letrec-by-need. Together with import and export they form the core of our proposal for Scheme's module structure.

Identifiers bound to non-functional values, *i.e.*, variables in the conventional sense, present a problem with these techniques. Import-by-value only evaluates the declaration once, *i.e.*, is incapable of reflecting changes in a variable's value; the other techniques assume that functions alone are exported. This is a restriction, but we argue on the following grounds that it is not a stringent one. To begin with, variables should not be exported at all: they are an explicit part of the implementation of modules and should for modularity and security reasons be hidden from the user of a module. Observer functions can and should equally well take their place ([10]). Second, Scheme is a call-by-value, not a call-by-reference, system. That means that identifiers are directly bound to objects, not to references. A natural consequence of this is that modules should export and import values, because identifiers themselves cannot be accessed beyond their lexical scope.

Another problem that we have not addressed at all is the issue of type checking. In conventional approaches to module structures import and export also exchange type information in order to prove the type correctness of programs. Scheme programs usually do not contain type information. However, type information can be added and used for compile-time type checking. Whether our proposals for module structures with late binding is compatible with type checking, must be explored.

**Acknowledgement.** We wish to thank Gary Brooks, Bruce Duba, Chris Haynes, John Nienart, Eugene Kohlbecker, and Mitch Wand for many useful comments on earlier drafts of this paper. We are especially indebted to Chris who stimulated the original discussion and provided several important observations.

## Bibliography

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

[2] Clinger, W.D., D.P. Friedman, M. Wand, A scheme for a higher-level semantic algebra, in *Algebraic Methods in Semantics*, J. Reynolds, M.Nivat (Eds)., 237–250, (1985).

[3] Clinger, W.D., The Revised Revised Report on Scheme, *Joint Technical Report Indiana University and MIT Laboratory for Computer Science*, (1985).

[4] Dahl, O.J. B.Myrhaug, U.Nygaard, *Simula67 Common Base Language*, Norwegian Computing Center (S-22), (1970).

[5] Friedman, D.P., C.T. Haynes, E. Kohlbecker, M. Wand, Scheme84 Interim Reference Manual, *Tech. Rep. No. 153*, Indiana Univeristy, Computer Science Department, (1985).

[6] Hanson, D.R., The Y programming language, *SIGPLAN Notices* **16**, 59–68, (1981).

[7] MacQueen D., Modules for Standard ML, *Conf. Rec. 1984 ACM Symposium on Lisp and Functional Programming*, 198 – 207, (1984).

[8] Kohlbecker, E., *Syntactic Extensions in a Lexically Scoped Language*, Ph.D. dissertation in progress, Indiana University, (1985).

[9] Landin, P.J., A formal description of ALGOL 60, in *Formal Description Languages for Computer Programming*, T.B. Steel (Ed.), (1965).

[10] Parnas D.L., On the criteria to be used in decomposing systems into modules, *Comm. ACM* **15**, 1053–1058, (1972).

[11] Reynolds, J., Syntactic control of interference, *Conf. Rec. 5th ACM Symposium on Principles of Programmig Languages*, 33 – 46, (1978).

[12] Sussman G.J., G. Steele, Scheme: An interpreter for the extended lambda calculus, *Memo 349*, MIT Artificial Intelligence Laboratory, (1975).

[13] Wirth N., Modula : A language for modular multi-programming, *Softw. pract. exp.* **7**, 3–35 (1970).

[14] Wirth N., *Programming in Modula-2*, Springer Verlag, (1983).

[15] US Department of Defense, *The Programming Language Ada – Reference Manual*, LNCS 106, Springer Verlag, (1981).

**About the Author**—Matthias Felleisen is currently working on his Ph.D. He received his M.S. in computer science from the University of Arizona, Tucson in 1981 and his Dipl. WIng. from Universität Karlsruhe, W. Germany in 1983. His interests include programming languages and their impact on software engineering.

**About the Author**—Daniel P. Friedman received his Ph.D. from The University of Texas at Austin in 1973. Since then he has been professor in

the Computer Science Department at Indiana University. He is primarily interested in programming languages and has published many papers in the field.