Interactive Modular Programming in Scheme *

Sho-Huan Simon Tung
Computer Science Department
Indiana University
Bloomington, Indiana 47405
email: stung@cs.indiana.edu

Abstract

This paper presents a module system and a programming environment designed to support interactive program development in Scheme. The module system extends lexical scoping while maintaining its flavor and benefits and supports mutually recursive modules. The programming environment supports dynamic linking, separate compilation, production code compilation, and a window-based user interface with multiple read-eval-print contexts.

1. Introduction

Interactive programming is an important technique for reducing program development time. An interactive programming system allows a programmer to enter a program or program fragment directly into the system and to receive the output from that program or fragment immediately, reducing the usual compile-link-execute process conceptually to a single evaluate step. Interactive programming is also valuable for experimental programming, rapid prototyping, and debugging.

Modular programming is an important programming paradigm for large-scale program development. Modular programs are easier to understand and maintain, thereby reducing overall program development cost. Many programming languages provide facilities for modular programming. These facilities give programmers direct control over the visibility of names among modules using import and export mechanisms. Other benefits of modular programming include separate compilation and enhanced reusability.

On the surface, it appears that modular programming and interactive programming are inherently incompatible. Interactive programming relies on the ability to make changes easily and dynamically. Modular programming, on the other hand, typically requires a more static model for program development. However, for some languages, these two programming paradigms can be merged gracefully while maintaining the benefits of both.

This paper presents a module system and a programming environment designed for Scheme [4, 10] that supports interactive modular programming. The module system has the following features:

1. It maintains the flavor and benefits of lexical scoping.

2. It allows flexible and responsive interactive programming. Interactive modifications of definitions and module interfaces are allowed without requiring recompilation.

3. It supports separate compilation, permits reuse of existing modules, and provides an incremental migration path to obtain production code.

The programming environment features a window-based user interface that associates each module with a file and an edit window. It also provides a module-sensitive read-eval-print loop for interactive program development.

We begin with an overview of related work. We then present the design of the module system and of the IMP (Interactive Modular Programming) programing environment. This is followed by a description of the implementation of the IMP system. Finally, we discuss issues of extending IMP to support object-oriented programming.

*This material is based on work supported in part by the National Science Foundation under grant number CCR-8803432 and by the Motorola Inc.

2. Related Work

Among many concrete proposals we have looked at, Felleisen and Friedman's module proposal is the only module system designed with consideration for interactive programming [5]. However, their system does not allow dynamically extended bindings to access lexical variables in a module. This restriction is too strong to be acceptable.

Some Scheme implementations support first-class environments [1]. A first-class environment captures the current lexical environment at the point where the first-class environment is created. When used with `eval` or `access`, first-class environments can be used to support a form of modular programming. This approach, however, disrupts the flavor and benefits of lexical scoping.

Curtis and Rauen recently proposed a module system designed for large scale programming in Scheme [3]. Their system supports more general interface specifications than IMP, but does not address interactive programming.

Common Lisp's [11] package system uses symbol tables to represent modules. Symbols defined as external in a package can be exported. Various mechanisms are available to access or to import exported symbols. The package system could be used as the low-level implementation for fully developed (static) modules. However, the major problem of the package system is that the association of symbols to packages is fixed at read time and can only be changed by rereading and recompiling all affected code, which is undesirable in an interactive programming system.

Queinnec and Padget designed a module system for Lisp [9], but their system binds imported identifiers early in the module definition phase. This requires the module dependency graph to be acyclic and defeats possibilities for flexible interaction.

Standard ML is a statically scoped programming language with a secure polymorphic type system and a module system [6]. However, its type system limits its flexibility as an interactive language. Modifying the value of an existing top-level binding has no effect on other bindings occurring before the modification. As a result, almost all modifications require that the entire program be reloaded.

We have also looked at conventional module-supporting languages such as CLU [8], Modula-2 [16], Modula-3 [2], and Ada [15]. CLU supports parameterized abstract data types. Modula-2 separates the definition of a module from its implementation, and it requires a one-to-one correspondence between the two components. Modula-3 differs from Modula-2 by allowing an implementation module to be associated with several interface specifications. Ada supports generic packages that must be instantiated statically through declarations. These languages, however, do not support interactive programming.

3. The Module System

The Revised^4 Report on Scheme [10] describes the structure of a Scheme program as consisting of a sequence of expressions and definitions. It describes the semantics of evaluating definitions as causing bindings to be created in the top-level environment, and the semantics of evaluating top-level expressions as executing them in order when the program is invoked or loaded and performing some kind of initialization. This kind of program structure is suitable for interactive programming. However, a single top-level environment is inadequate to support modular programming.

The design of our module system extends Scheme's program structure to contain a sequence of module expressions and module definitions and replaces the single top-level environment with a separate *module environment* for each module in a program. The syntax of the module system is presented in Figure 1.

The effect of evaluating a public or a private definition in a module is to cause a binding to be created in the public or private portion of the module environment. Public bindings are exported while private bindings are visible only within the module.

The **import** definition provides a flexible way to declare the imports of a module. As an example, the following definition:

(**import** *main* (*stack* (*queue* (*q-init init*) *enq deq*)))

declares that the public bindings of the module *stack* and the *init*, *enq*, and *deq* binding of the module *queue* are imported by the module *main* with *init* renamed as *q-init*. It is possible to import a module or a binding that is yet to be defined. However, actually using the binding before it is defined would cause an error.

A module environment consists of its imported bindings along with its private and public bindings. The **with** syntactic form evaluates an expression in the module environment of a designated module. Although the syntax of the module system is text-oriented, it is designed to be mapped into the window-based user interface. In particular, the module name *module* is assumed to be the name of the window, and explicit use of the **with** statement is no longer necessary with the help of a module-sensitive read-eval-print loop (see Section 4).

An important restriction of the system is that public definitions cannot be assigned (although they may be redefined interactively at top level during program development) and can only bind identifiers to procedures. No expressive power is lost as a result of these

$$
\begin{array}{rcll}
\textit{id, ex-id, local-id,} & & & \\
\textit{m-all, m-sel, module} & \in & \textit{Ide} & \text{Scheme identifiers} \\
\textit{expression} & \in & \textit{Exp} & \text{Scheme expressions} \\
P & \in & \textit{Pgm} & \text{programs} \\
MD & \in & \textit{MDef} & \text{module definitions} \\
ME & \in & \textit{MExp} & \text{module expressions}
\end{array}
$$

$$
\begin{array}{rcl}
P & ::= & \{MD \mid ME\}^* \\
MD & ::= & (\textbf{import}\ \textit{module Imports}) \\
 & \mid & (\textbf{public}\ \textit{module id expression}) \\
 & \mid & (\textbf{private}\ \textit{module id expression}) \\
ME & ::= & (\textbf{with}\ \textit{module expression}) \\
\textit{Imports} & ::= & (\{\textit{m-all} \mid \textit{Select}\}\ \ldots) \\
\textit{Select} & ::= & (\textit{m-sel}\ \{\textit{id} \mid (\textit{local-id ex-id})\}^+)
\end{array}
$$

Figure 1: Syntax of the Module System

restrictions. If we wish to export the value of a variable, say x, and to allow x to be assignable by other modules, we can simply export a "reference procedure," *e.g.*, (**lambda** () x) and an "assignment procedure," *e.g.*, (**lambda** (v) (**set!** x v)).

These two restrictions encourages programmers to write programs that are more easily analyzed by both the compiler and the programmer, since any code that can assign a variable is insulated within a single module. The consequence is that the programmer and the compiler can simply scan a module to determine whether a given variable is assigned, and can more often determine the types of values assigned to the variable when it is assigned. Furthermore, these restrictions naturally lead to the use of assignment procedures that ensure the new value is in the range of acceptable values. For example, if a variable must be assigned to positive integers, the assignment procedure could be written as:

```
(lambda (v)
  (if (and (integer? v) (> v 0))
      (set! x v)
      (error)))
```

which results in safer, more readable, and more easily analyzed code.

Figure 2 presents an example containing two modules that import from and export to each other. Both of the modules import *scheme*, which exports standard Scheme primitives. From the user's point of view, the semantics of evaluating an expression in the context of a module is essentially the same as the semantics of evaluating the expression in Scheme except that a local module environment is used as the top-level environment. For example, while evaluating (*odd?* 3) in *even-module*, the free variable *odd?* is found in *even-module*'s module environment since *even-module* imports *odd-module* which exports *odd?*. However, the *odd?* procedure itself is defined in *odd-module*; the free variables =, −, and *even?* should therefore find their values using *odd-module*'s module environment.

```
(import even-module (odd-module scheme))
(public even-module even?
  (lambda (x)
    (if (= 0 x)
        #t
        (odd? (- x 1)))))

(import odd-module (even-module scheme))
(public odd-module odd?
  (lambda (x)
    (if (= 0 x)
        #f
        (even? (- x 1)))))

(with even-module (odd? 3)) => #t
(with even-module (even? 2)) => #t
```

Figure 2: Recursive modules

In order to provide a more precise description of the informal semantics described above, we present the denotational semantics of the system in Figure 3 using an

$lookup$: $MDB \rightarrow Ide \rightarrow Env \rightarrow Ide \rightarrow E$
$lookupM$: $MDB \rightarrow Ide^* \rightarrow Ide \rightarrow E$

$$
\begin{array}{l}
\quad lookup\ \mu\ m\ \rho\ i = \\
\quad\quad \textbf{if}\ \rho\ \mathrm{i} = \text{undefined}\ \textbf{than} \\
\quad\quad\quad \textbf{if}\ (\mu m \downarrow 1)\ i\ = \text{undefined}\ \textbf{then} \\
\quad\quad\quad\quad \textbf{if}\ (\mu m \downarrow 2)\ i\ = \text{undefined}\ \textbf{then} \\
\quad\quad\quad\quad\quad lookupM\ \mu\ (\mu m \downarrow 3)\ i \\
\quad\quad\quad\quad \textbf{else}\ (\mu m \downarrow 2)\ i \\
\quad\quad\quad \textbf{else}\ (\mu m \downarrow 1)\ i \\
\quad\quad \textbf{else}\ \rho\ i
\end{array}
$$

$$
\begin{array}{l}
\quad lookupM\ \mu\ [\![]\!]\ i = \text{undefined} \\
\quad lookupM\ \mu\ [\![first\ rest^*]\!]\ i = \\
\quad\quad \textbf{if}\ (\mu first \downarrow 2)\ i\ = \text{undefined}\ \textbf{then} \\
\quad\quad\quad lookupM\ \mu\ rest^*\ i \\
\quad\quad \textbf{else}\ (\mu first \downarrow 2)\ i
\end{array}
$$

Figure 4: The *lookup* function

extended subset of Scheme [12]. To simplify the presentation, support for renaming imported identifiers has been omitted.

In addition to the straightforward tasks performed by **import**, **private**, and **public** to maintain the module database, the following aspects of the semantics are essential for the module system to preserve lexical scoping and to support interactive programming and recursive modules:

1. The name of the module where the **lambda** expression appears is closed along with the lexical environment in the returned closure.

2. The module database is provided dynamically as an implicit argument when the closure is applied.

3. The interpretation of the textual description of a module's imports is delayed until a variable lookup is performed.

The first item above and an extended variable lookup mechanism preserves the essence of lexical scoping, allowing the programmer to use the static program text to determine a variable's binding. The extended variable lookup mechanism is described with the function *lookup* in Figure 4. The *lookup* function uses the following precedence rule to find the value of a variable in a module: lexical variables have precedence over locally defined private or public variables, and locally defined private or public variables have precedence over imported variables. The second and the third items support interactive programming and recursive modules.

4. The IMP System

We have implemented a prototype user interface for the IMP system. The user interface relates modules with files, editing windows, and a multiple-context read-eval-print loop. It is based on GNU Emacs and can also be used with Epoch, which is a variant of GNU Emacs supporting true multi-window editing under the X window system [7]. Figure 5 presents IMP's user interface.

The syntax of the module system requires a module name to be specified for every definition and expression within the module. The IMP system removes this inconvenience by implicitly using the name of the file or the window with which the module is associated. The implicit name is then used to obtain the evaluation context for definitions and expressions in the module. The **import** definition should appear at the beginning of the file in order to prepare the evaluation context for the rest of the file. The *scheme* module is implicitly imported by every module.

Unlike traditional read-eval-print loops providing a single evaluation context, IMP's read-eval-print loop can be associated with several evaluation contexts. The prompt of the read-eval-print loop represents the name of a module. Expressions subsequently entered after the prompt would be evaluated in the module environment of the module. The user can explicitly change the current module to some other module. An expression can also be sent from an editing window directly to the read-eval-print loop; the module associated with the editing window is then used to determine the evaluation context.

An IMP program is organized as a *project*. A project is composed of zero or more modules. The project records the load order of the modules and the directories from which to load the modules. IMP supports two kinds of modules: *developing modules* and *developed modules*. Definitions in developing modules can be changed or deleted. Developed modules cannot change their definitions and are not allowed to import from developing modules. Developed modules can be compiled into more efficient code than developing modules. If an IMP project contains only developed modules then the entire project can be compiled into efficient Scheme code without any of the module statements used during the developing phase.

5. Implementation

The implementation of the IMP system is best described by techniques used in implementing dynamic linking, separate compilation, and production code compilation.

Syntactic Categories:

c	$\in$	Con	$= \{\text{undefined}, +, ...\}$	constants
$m,\ i$	$\in$	Ide		identifiers
s	$\in$	$Stmt$		statements
e	$\in$	Exp		expressions

Semantic Domains:

ρ	$\in$	$Env = Ide \to E$	environments
μ	$\in$	$MDB = Ide \to MEnv$	module database
		$MEnv = (\text{Env} \times \text{Env} \times Ide^*)$	module environments
		C	constant values
ϵ	$\in$	$E = C + (MEnv \to E \to E)$	expressed values

Semantic functions:

$\mathcal{S}$: $Stmt \to MDB \to (MDB \times E)$
$\mathcal{K}$: $Con \to E$
$\mathcal{E}$: $Exp \to MDB \to Ide \to Env \to E$

$$\mathcal{S}\,[\![\textbf{import}\ m_0\ m^*)]\!]\,\mu = \langle \mu[m_0 \leftarrow \langle \mu m_0 \downarrow 1, \mu m_0 \downarrow 2, m^* \rangle], \text{undefined} \rangle$$

$$\mathcal{S}\,[\![(\textbf{private}\ m\ i\ e)]\!]\,\mu = \langle \mu[m \leftarrow \langle (\mu m \downarrow 1)[i \leftarrow \mathcal{E}\,[\![e]\!]\mu m \rho_0], \mu m \downarrow 2, \mu m \downarrow 3 \rangle], \text{undefined} \rangle$$

$$\mathcal{S}\,[\![(\textbf{public}\ m\ i\ e)]\!]\,\mu = \langle \mu[m \leftarrow \langle \mu m \downarrow 1, (\mu m \downarrow 2)[i \leftarrow \mathcal{E}\,[\![e]\!]\mu m \rho_0], \mu m \downarrow 3 \rangle], \text{undefined} \rangle$$

$$\mathcal{S}\,[\![(\textbf{with}\ m\ e)]\!]\,\mu = \quad \langle \mu, \mathcal{E}\,[\![e]\!]\mu m \rho_0 \rangle$$

$$\mathcal{S}\,[\![s_0\ s_1]\!]\,\mu = \quad \mathcal{S}\,[\![s_1]\!]\,(\mathcal{S}\,[\![s_0]\!]\,\mu) \downarrow 1$$

$$\mathcal{E}\,[\![c]\!]\mu m \rho = \mathcal{K}\,[\![c]\!]$$

$$\mathcal{E}\,[\![i]\!]\mu m \rho = lookup\ \mu\ m\ \rho\ i$$

$$\mathcal{E}\,[\![(\textbf{lambda}\ (i)\ e)]\!]\mu_0 m \rho = [strict\ \lambda \mu_1 \epsilon . \mathcal{E}\,[\![e]\!]\mu_1 m(\rho[i \leftarrow \epsilon])]\ in\ E$$

$$\mathcal{E}\,[\![(e_0\ e_1)]\!]\mu m \rho = (\mathcal{E}\,[\![e_0]\!]\mu m \rho \mid (MEnv \to E \to E))\mu(\mathcal{E}\,[\![e_1]\!]\mu m \rho)$$

Notations:

$\langle \ldots \rangle$	sequence formation
$s \downarrow k$	kth member of the seqnence s $(1-\text{based})$
$(\rho[i \leftarrow x])\ i'$	$(i' = i) \to x,\ \rho\ i'$
$x\ in\ D$	injection of x into domain D
$x \mid D$	projection of x to domain D

Figure 3: Denotational Semantics of Modules

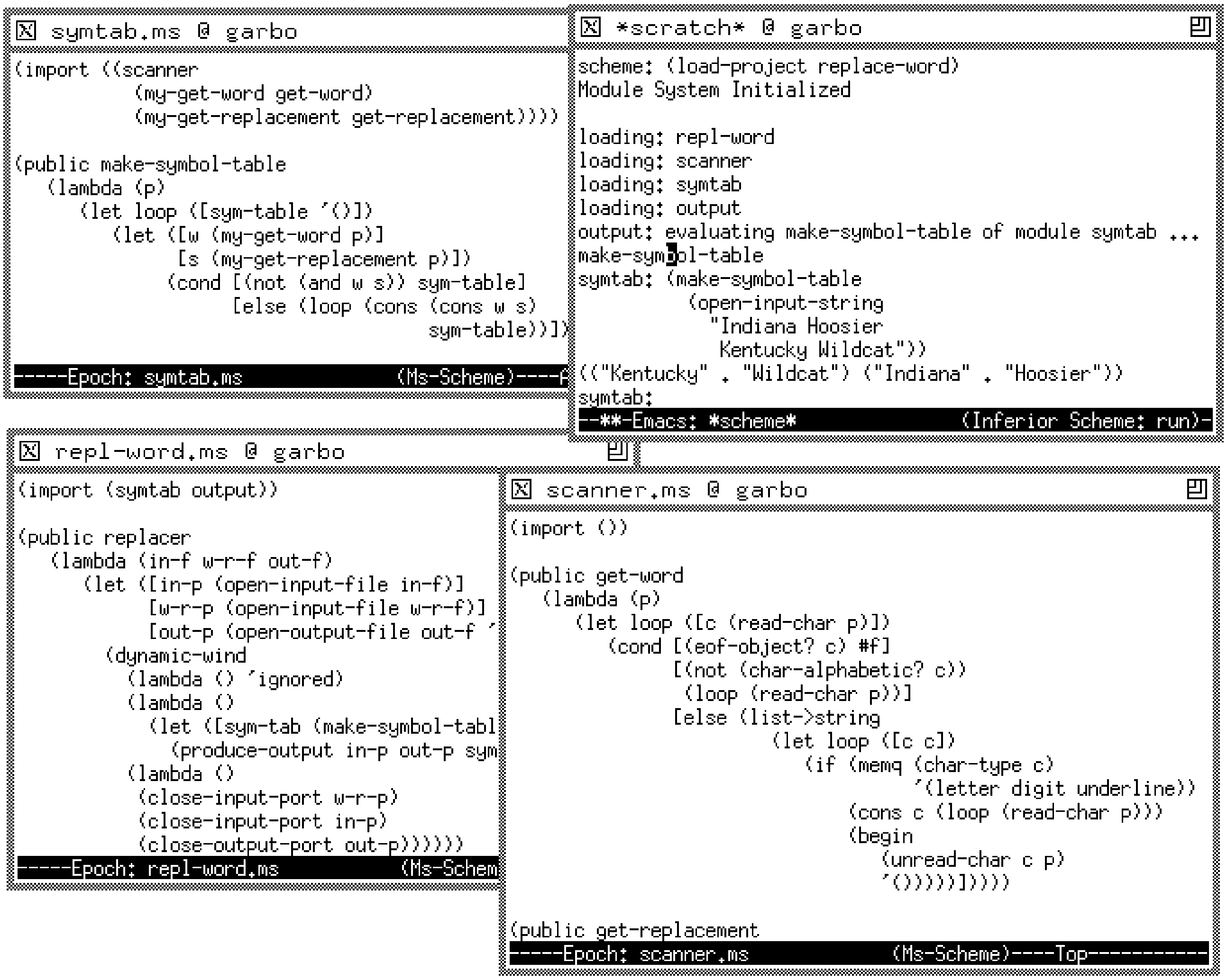

Figure 5: User interface

```
Module m1:

(import (m2))

(private a 1)
(public set-a!
  (lambda (n)
    (set! a n)))

(public foo
  (lambda () bar))
```

```
Module m2:

(import (m1))

(private b 2)
(public bar
  (lambda ()
    (foo)
    (set-a! 3)))
```

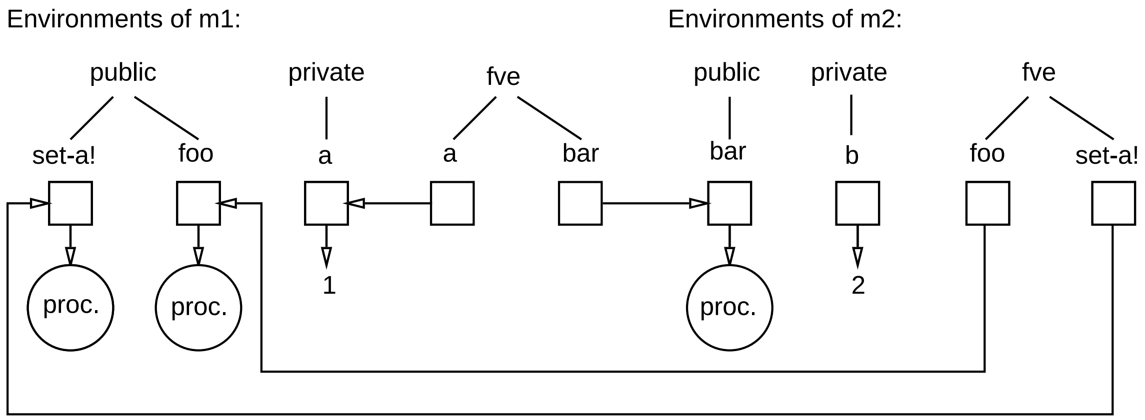

Figure 6: Module linkage

5.1 Dynamic Linking

The semantics of our module system requires the bindings of free variables in a top-level expression to be determined dynamically at run time. This property provides the necessary flexibility for interactive programming. However, a naive implementation of the semantics can result in unacceptable performance. The implementation presented here keeps track of the import/export relations among modules and uses double indirections with an implicit incremental link step after each interactive modification to resolve the bindings of free variables.

The IMP system keeps track of variable bindings for all modules. Each module has environments for public variables, private variables, and free variables. The public and private environments associate identifiers with the locations that contain their values. The free-variable environments (FVE) associate free variables with locations that contain pointers to the locations of local variables or public variables imported from other modules. In consequence, variable references and assignments require at most one or two memory references. A binding in the public or private environment of a module is created when a public or private definition is evaluated. A binding in the free-variable environment of a module is created when an expression containing the free variable is compiled. Before evaluating the compiled expression, an implicit link step is performed that associates bindings in the free-variable environment with the locations of the variables defined in the module or imported from other modules. A run-time error is signaled if a free variable is used during evaluation for which no binding has been established. In addition, the system also provides a command that lists the identifiers that are currently unbound in a module. Figure 6 depicts the implementation using two modules that import from and export to each other. Note that the private variable a also appears in *m1*'s free-variable environment, since a occurs free in the expression defined by *set-a!*.

IMP allows interfaces among modules to be modified freely. Possible modifications include removing a module, defining a module, changing a module's imports, deleting a binding, adding a binding, and modifying an existing binding. To support these modifications, the free-variable environments of affected modules must be updated after every user interaction that changes the dependencies among modules. For example, if *m1* imports *bar* from another module *m3*, the entry *bar* in *m1*'s FVE must be changed to point to the *bar* in the public environment of *m3*. In extreme circumstances, the amount of relinking required could be high. In practice, however, this does not appear to be a problem.

When a public or a private binding is removed from a module, the free variables used in the removed binding should be removed from the free-variable environment if not referenced elsewhere in the module. Referring to Figure 6, if the public procedure foo is removed from module *m1*, the binding *bar* in *m1*'s FVE should also be removed. The system uses reference counts to determine whether free variables should be removed.

5.2 Separate Compilation

A separate compilation mechanism for developed modules must satisfy the following requirements:

1. Developed modules must integrate well with developing modules.

2. In order to support mutually recursive modules, the system must allow loading a developed module even though the developed module imports some items that may not be available at load time.

3. The efficiency overhead for variable access present when developing must be eliminated.

The first requirement can be satisfied by having the developed module establish the environments for its public variables. The second requirement requires a "delayed linking" mechanism for bindings that are not available at load time. The third requirement is satisfied by accessing free variables in a developed module without using indirection, which we can do since developed modules cannot import from developing modules.

Figure 7 shows the code generated for the module *m1* in Figure 6. Procedures *do-import*, *do-binds*, and *do-public* are provided by the IMP system[1]. The procedure *do-import* checks whether imported modules are also developed. The procedure *do-binds* initializes the public environment of the module. The procedure *do-public* puts the value of a variable definition into the public environment. Note that the private variables need not be initialized, since they are not visible outside the module.

The most interesting part of Figure 7 is the **letrec** expression, which is used to establish bindings for free variables in *m1*. Private variables are simply allocated with ordinary bindings. Any free variable that refers to a locally defined public variable or an imported public variable is bound to an **if** expression that returns a procedure if the public variable's value is available at load time or a *delay procedure* that postpones the reference of the variable until the delay procedure is invoked. The **set!** expression changes the delay procedure to an imported procedure when it is first invoked. This link-by-need mechanism allows mutually recursive modules to be loaded into the system. The procedure *value-getable?* checks whether the value of a variable is available. The procedure *get-value* returns the value of a variable. The restriction that public variables can be bound only to procedures simplifies this link-by-need mechanism.

Except for the cost associated with the link-by-need mechanism, the code generated for a developed module is as efficient as the corresponding Scheme program.

5.3 Project Compilation

The goal of project compilation is to eliminate all overhead associated with developing or developed modules. Because bindings of free variables are available at compile time and exported variables are not assignable, this goal can be achieved.

The project compiler first translates the entire project by consistently renaming every free variable name, say *v1*, with a name of the form *m-v1* where *v1* is defined in module *m*, or with a name of the form *m-v2*, if *v2* is the original name and is renamed to *v1* in an importing module. The project compiler is free to open-code any public procedure. The translated code is at least as efficient as an equivalent Scheme program written without modules.

6. Conclusion

One of the most important programming paradigms that Scheme systems support is interactive programming. We believe that modular programming facilities are also important. Unfortunately, traditional module systems are inherently at odds with interactive programming. Adding a traditional module system to Scheme would subvert most of Scheme's interactive capabilities.

Scheme is a lexically scoped language. The static nature of lexical scoping makes programs easier to understand and allows compilers to generate more efficient code. A module system designed for Scheme should not lose the benefits of lexical scoping.

The significant accomplishments of the system presented in this paper are that it supports interactive modular programming and maintains the flavor and benefits of lexical scoping. Efficient implementation techniques are provided for dynamic linking, separate

[1] To aid the presentation, the original names of these procedures are used in the generated code. In reality, these and other system procedures are bound to otherwise inaccessible names to avoid being captured by user-defined variables.

```
(do-import 'm1 '(m2))

(do-binds 'm1 '((pubind foo) (pubind set-a!)))

(letrec ([a #f]
         [bar
          (if (value-getable? 'm2 'bar)
              (get-value 'm2 'bar)
              (lambda args
                (if (value-getable? 'm2 'bar)
                    (begin (set! bar (get-value 'm2 'bar))
                           (apply bar args))
                    (error '()
                      "Variable ~s imported from ~s to ~s is not bound"
                      'bar 'm2 'm1))))])
  (set! a 1)
  (do-public 'm1 'set-a! (lambda (n) (set! a n)))
  (do-public 'm1 'foo (lambda () bar)))
```

Figure 7: Separate Compilation

compilation, and project compilation. In addition, we have also presented the design of a programming environment that provides a window-based user interface with multiple read-eval-print contexts and an incremental migration path for a programming project to obtain production code.

Modular programming and object-oriented programming are two programming paradigms that share many common objectives. These common objectives include encapsulation and modularity and should be supported by common rather than different language facilities. We have extended IMP to support object-oriented programming [13, 14].

Several macro systems supporting hygienic macros have been proposed for Scheme. These proposals provide different facilities for writing low-level macros. We are considering integrating and extending the system proposed by Hieb and Dybvig to support exported macros [10].

Acknowledgements

I would like to thank Kent Dybvig, Bob Hieb, Michael Ashley, Carl Bruggeman, Dan Friedman, and Shinn-Der Lee for many useful insights and suggestions on the design of the system and on earlier drafts of this paper. The Scheme programs in this paper were typeset using Dorai Sitaram's SLaTeX.

References

[1] Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1984.

[2] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report. Technical Report 31, DEC Systems Research Center, 1988.

[3] Pavel Curtis and James Rauen. A module system for Scheme. In *Conference Record of the 1990 ACM Lisp and Functional Programming*, 1990.

[4] R. Kent Dybvig. *The Scheme Programming Language.* Prentice-Hall, 1987.

[5] Daniel P. Friedman and Matthias Felleisen. A closer look at export and import statements. *Computer Language*, 11(1):29–37, 1986.

[6] Robert Harper, Robin Milner, and Mads Tofte. The definition of Standard ML. Technical Report ECS-LFCS-89-81, Department of Computer Science, University of Edinburgh, 1989.

[7] Simon Kaplan, Alan M. Carroll, Christopher Love, and Daniel M. LaLiberte. *Epoch - GNU Emacs for the X Window System.* Department of Computer Science, University of Illinois at Urbana-Champaign, 1990.

[8] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in

CLU. *Communications of the ACM*, 20(8):564–576, 1977.

[9] Christian Queinnec and Julian Padget. A detailed summary of a deterministic model of modules and macros for Lisp. Technical Report LIX/RR/90/01, Ecole Polytechnique, Laboratoire d'Informatique, 91128 Palaiseau Cedex (France), July-December 1989.

[10] Jonathan Rees and William Clinger. (Editors), Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, to appear.

[11] Guy L. Steele Jr. *Common Lisp.* Digital Press, 1990. Second Edition.

[12] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Mass., 1977.

[13] Sho-Huan Simon Tung. *Merging Interactive, Modular, and Object-Oriented Programming.* PhD thesis, Indiana University, Bloomington, 1992.

[14] Sho-Huan Simon Tung and R. Kent Dybvig. Object-oriented programming with interactive modules. in preparation.

[15] US Government - Department of Defense. The programming language ADA - reference manual. *Lecture Notes in Computer Science, Vol. 106*, 1981.

[16] Niklaus Wirth. *Programming in Modula-2.* Springer Verlag, 1983.