MERGING INTERACTIVE, MODULAR, AND OBJECT-ORIENTED PROGRAMMING

Sho-Huan Simon Tung

Submitted to the faculty of the Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

This material is based on work supported by the Sandia National Laboratories under contract number 75-5466 and by the National Science Foundation under grant number CCR-8803432 and by the Motorola Inc.

Abstract

Interactive, modular, and object-oriented programming are three important programming paradigms. Interactive programming encourages experimental programming and fast prototyping and is most valuable for solving problems that are not well understood. Modular programming is indispensable for large-scale program development and is also useful for smaller programs. Object-oriented programming features classes, objects, and inheritance and is suitable for many real world applications.

This dissertation describes an approach of merging interactive, modular, and object-oriented programming by presenting the definition, design, and implementation of the λ_{imp} language, the IMP system, and the IMOOP system. The primary benefit of merging these three paradigms is that the programmer can use either paradigm where appropriate.

In order to merge interactive and modular programming, the programmer must be allowed to modify variable bindings and module interfaces during program development. Furthermore, the effects of these modifications must also propagate to affected modules automatically. The ability to modify one or more modules at a time is provided with a window-based user interface that strongly relates modules with files, edit windows, and a read-eval-print loop with multiple evaluation contexts. The semantics of these modifications is specified with the λ_{imp} language which extends λ-calculus to support interactive modular programming. This semantics gives a formal description of the notion of propagating variable definitions and module interfaces.

The λ_{imp} language supports recursive modules, that is, it allows the import/export relation of modules to contain cycles.

The design and implementation of IMP addresses many practical problems of supporting interactive modular programming for the Scheme programming language. These problems include efficient implementation of variable lookup, separate compilation, project compilation, using macros in the context of modules, and programming environment supports. The dissertation presents solutions to these problems.

IMOOP extends IMP with support for object-oriented programming. It uses IMP's ability to export variables, to rename imported variables, and to define recursive modules to provide an object system that offers many advantages. These advantages include a flexible and efficient slot-access mechanism with strong encapsulation, a simple approach for constructing generic functions, a flexible mechanism for handling conflicts resulting from multiple inheritance, and tight integration of modular programming and object-oriented programming.

Acknowledgements

I would like to thank my advisor, R. Kent Dybvig, for his support, assistance, and friendship. I am especially grateful for his suggestions on improving my writing skill as well as on many technical aspects of this research.

I would like to thank the other members in my committee. Chris Haynes encouraged me to pursue a Ph.D degree; Dennis Gannon supported me during a critical stage of my study; Bye Wynne provided advice for career directions.

I would like to thank Dan Friedman for various pieces of advice I have received from him.

I would like to thank my friends and fellow students in the Department of Computer Science. In particular, I would like to thank Carl Bruggeman, Hsianlin Dzeng, Robert Hieb, Jenq-Kuen Lee, Shinn-Der Lee, Shing-Shong Bruce Shei, and Chang-Yaw Wang for many activities shared among us.

I would like to thank my parents and my wife for their constant support throughout my entire graduate study.

Contents

List of Figures

Chapter 1

Introduction

Interactive, modular, and object-oriented programming are three important programming paradigms. These three paradigms have different strengths and weaknesses, and are suitable for solving problems of different natures. Designing language facilities, user interfaces, and programming systems that allow an application program to be developed jointly using these three paradigms is the theme of this dissertation.

1.1 Problems and Paradigms

Computers are problem solving tools. The advance of computer technologies, both hardware and software, are largely driven by the needs of problem domains and the needs of problem solving processes. For instance, fast parallel computers are invented to meet the demands of large and complex scientific applications, and CASE (Computer-Aided Software Engineering) tools are constructed to support the software development processes.

The advance of programming languages are no exception. Many programming languages including Cobol, Fortran and Ada are invented primarily for specific application domains. Many other programming languages are invented, however, to support different programming techniques or program development paradigms.

Interactive programming is an important programming technique that is most suitable for solving problems that are not well understood [80, 75]. An interactive programming system allows a programmer to enter programs or program fragments directly into the system and to receive the output from that program or fragment immediately, reducing the usual compile-link-execute step conceptually to a single evaluate step. This feature encourages experimental programming and fast prototyping which are effective approaches to understanding vaguely defined problems. In addition, interactive programming is a valuable debugging technique and can significantly improve the productivity of programmers [8].

Modular programming, however, is best suited for problems whose solutions are understood. Modular programming encourages solving a problem by decomposing the problem into modules. This process requires an understanding of the problem and the structure of its solution. Modular programming languages provide facilities for a module to hide its internal details and to export services to other modules. Other benefits of modular programming include separate compilation and enhanced reusability.

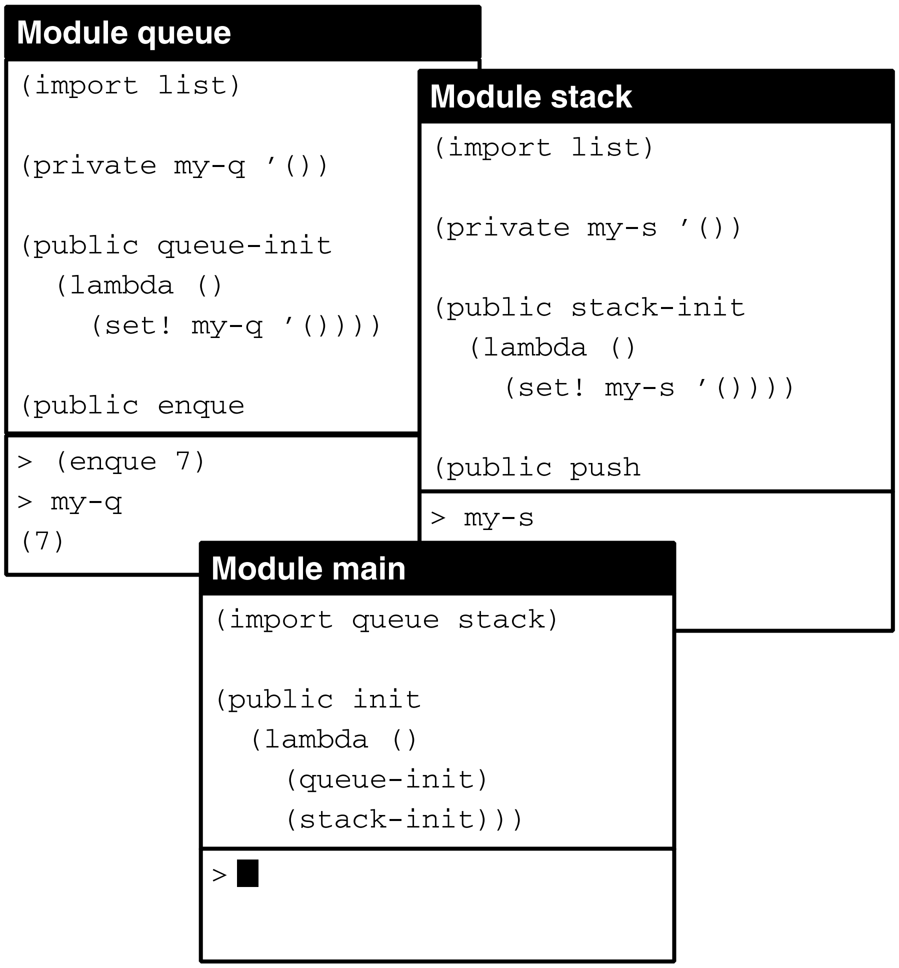
Object-oriented programming features classes, objects, and inheritance. These features allow existing code to be extended, specialized, or reused for different situations. Object-oriented programs are often written not only for providing solutions of problems but also for adopting the solutions to variants of the problems [12]. Many real world applications are developed using the object-oriented paradigm. Examples of these applications include graphical user interfaces, data base systems, visualization systems, and parallel or distributed computing systems.

Interactive, modular, and object-oriented programming all have features that are useful for problem solving at different areas in the entire problem space. However, these areas in the problem space are not disjoint: Many problems can be partly understood, partly not understood. Parts may be best organized with the object-oriented approach, and other parts not. Neither interactive, modular, nor object-oriented programming alone is best suited for solving these problems.

1.2 Multi-Paradigm Programming

This dissertation presents the semantics, design, user interfaces, and implementation techniques that merges interactive, modular, and object-oriented programming paradigms. The λ_{imp} language presents the basic semantic properties of an interactive modular programming system designed for a window-based user interface. The IMP system (Interactive Modular Programming) is an approach to supporting interactive modular programming for the Scheme programming language [58, 35, 20]. The IMOOP system (Interactive, Modular, and Object-Oriented Programming) extends IMP with supports for object-oriented programming. The most interesting feature of the IMOOP system is that it allows joint uses of the three paradigms to solve problems. Although most of the techniques described here are designed for Scheme, they should be applicable to other programming languages as well.

Interactive programming and modular programming are two seemingly incompatible programming paradigms. Interactive programming systems provide the ability to modify a program dynamically. Modular programming, on the other hand, typically requires a more static model for program development. In either case, a program must be written with an editor and subsequently saved in a file. A multi-window user interface that allows its user to modify modules in edit windows, to test the modification with module-sensitive read-eval-print loops, and to save the changes in files for future references seems to be a natural way of supporting interactive modular programming (see Figure 1). However, this natural and intuitive programming style



```
(import list)

(private my-q '())

(public queue-init
  (lambda ()
    (set! my-q '())))

(public enque
```

```
> (enque 7)
> my-q
(7)
```

```
(import list)

(private my-s '())

(public stack-init
  (lambda ()
    (set! my-s '())))

(public push
```

```
> my-s
```

```
(import queue stack)

(public init
  (lambda ()
    (queue-init)
    (stack-init)))
```

```
> █
```

Figure 1: A multiple window user interface

must also be associated with a well defined semantics and an efficient implementation. Developing the semantics and implementation techniques for this user-interface model are two challenging problems of this project.

Modular programming and object-oriented programming have many common objectives. These common objectives include encapsulation and modularity and should be supported by common rather than different language facilities. Most modular programming systems already provide facilities that support encapsulation and modularity. Extending the modular programming system to support object-oriented programming in a coherent manner is a third challenge of this project.

One contribution of this dissertation is the formal semantics of the λ_{imp} language that is intended to describe the meaning of various interactions supported by the window-based user-interface model. The semantics extends lexical scoping and allows modules to be used to determine the meaning of variable references. The most interesting aspect of the semantics is that it allows interactive modification of variable definitions as well as module interfaces and uses late binding semantics to determine the meanings of variable references. The late binding semantics is confined within regions defined by local definitions of a module and its imported modules. While lexically scoped languages allow programmers to use the block structure of a program to determine the meaning of lexical references in the program, the λ_{imp} language allows programmers to use local definitions and imported modules to determine the meaning of non-lexical references in a module. In addition, the λ_{imp} language also supports recursive modules.

Another contribution is the implementation of the IMP programming system for Scheme. IMP provides the convenience of interactive programming but does not compromise the discipline of modular programming. In addition, IMP supports separate compilation, delivered mode compilation, syntactic extension, and a simple method for maintaining the consistency of the state of the IMP system during interactive programming.

A third contribution of this dissertation is the design, definition, and implementation of the IMOOP programming system. IMOOP extends IMP with supports for object-oriented programming. IMOOP provides a flexible and efficient slot-access mechanism with strong encapsulation, permits simplified treatment of generic functions, supports flexible handling of conflicts resulting from multiple inheritance, and allows tight integration of modular programming and object-oriented programming.

1.3 Overview

Name space management techniques are often designed to facilitate modular or object-oriented programming. Programming environments are also often designed to facilitate interactive, modular, or object-oriented program development. Chapter 2 reviews previous work in name space management, modular programming languages, object-oriented programming languages, and language-centered programming environments.

Denotational semantics is an important tool for the design and implementation of programming language systems [69]. The denotational semantics of a programming language provides a formal and precise definition of the language and is useful for its users and its implementors [62, 76]. However, complete specifications of realistic programming languages in denotational semantics are often difficult to read. To overcome this difficulty, Chapter 3 presents the denotational semantics of a toy language designed to show the most interesting semantic properties of interactive modular programming. The language is text-oriented, however it is designed so that it can be mapped to the window-based user-interface model. An interpreter implemented by translating the semantics to Scheme is presented in Appendix A.

In addition to clear and simple semantics, programming environment support and efficient implementation strategies are also important aspects of programming language research. Chapter 4 presents the design and implementation of IMP to demonstrate that the language features presented in Chapter 3 can be associated with a multi-window programming environment and implemented efficiently. In addition, many practical considerations such as separate compilation and syntactic extension are also discussed.

Chapter 5 presents the design and implementation of the IMOOP programming system. IMOOP extends the interactive modular programming system to support object-oriented programming. It provides improvements over existing Lisp-based object systems designed without modules. A complete description of IMOOP is presented in Appendix B.

Chapter 6 presents the conclusion of the dissertation and suggests some areas for further research.

Chapter 2

Related Work

This dissertation relates to three major areas of research in computer science. One area concerns name space management and its application on modular programming. Another area concerns object-oriented programming languages. A third area concerns language-centered programming environments. This chapter reviews related work in these areas.

2.1 Modules and Name Space Management

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 [23]. Interactive languages require late binding for flexible program development. By restricting exported values to procedures, they support various late binding techniques for defining modules and importing items. The ultimate goal is to allow arbitrary load orders for modules and to permit interactive extension and redefinition of bindings in modules. However, this goal is not entirely achieved, since dynamically extended bindings cannot 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 when the first class environment is created. When used with eval or access, which allow expressions or variables to be evaluated dynamically in a first class environment, some form of packaging facilities can be supported. However, first class environments suffer at least three limitations which make them inadequate to support interactive modular programming:

1. A first class environment captures indiscriminately the entire environment at the point when the first class environment is created. This feature makes it difficult to use first class environments to model export, since a module should not export all of its bindings.

2. Every first class environment shares the same initial environment and possibly some other common parent environments. These redundant information make it difficult to combine first class environments to model import, since it is unclear how the shared portion of different first class environment should be combined when imported in a module.

3. It is unclear how first class environments can be used to support recursive modules or to allow referencing a first class environment before it is created.

Common Lisp's [66] 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. However, packages are low level implementation concepts and provide much weaker expressive power for name space management than environments.

Symmetric Lisp [36] is a programming language based on environments. The map expression constructs an environment containing zero or more regions. Expressions associated with the regions are evaluated in parallel. Map evaluation is *non-strict*, meaning that the regions of a map may be accessed even while other regions in the same map continue to be evaluated. Many primitive operations on maps (environments) are available. The most interesting one is the kappa expression which creates an environment abstraction. This abstraction, when applied, generates a map with regions specified by the kappa expression. Symmetric Lisp explores many possible uses of environments in a parallel programming language.

Instead of the single environment domain used in Symmetric Lisp, Lamping's model of transparent parameterization [46] uses two environment domains to determine the meaning of an expression. The essence of his model is that expressions are parameterized over two kinds of variables: the values of lexical variables in an expression are determined by the *lexical* environment at the point of definition, and the values of data variables are determined by the *use* environment at the point where the expression is used. The lexical environment is extended with the let expression. The use environment is extended dynamically with the supply expression, which provides values for free data variables. Lamping's model is able to express a variety of language constructs, including module linkage. However, his model may be too dynamic to implement efficiently.

Queinnec and Padget have designed a module system for Lisp [56]. The design goals of their system are to support separate compilation and to control the visibility of resources. A module is available for use after it is defined and loaded. In the module definition phase, the module's top-level environment is established with imported bindings and free variables in the module. In the module loading phase, actual evaluation takes place. A program starts by executing the (start-module *module-name function-name arguments*) special form which applies the *function-name* to its *arguments*. In their system, imported identifiers are bound early in the module definition phase. This requires the module dependency to be acyclic and defeats possibilities for flexible interaction.

Curtis and Rauen recently proposed a module system designed for large scale programming in Scheme [14]. In their proposal, a module is an isolated scope which may be nested inside other modules. Modules are anonymous. They communicate with each other by sharing items specified in *interfaces*. Interfaces are named environments that may contain both syntax bindings and value bindings. They also proposed a syntactic extension facility in the context of modules based on syntactic environments [4]. However, they did not address the problem of fitting their system into Scheme's interactive programming style.

Standard ML is a statically scoped functional programming language with a secure polymorphic type system [29]. 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 of a program require that the entire program be reloaded. The module extension to Standard ML associates each module with two environments: a *signature* containing the interface of the module, and a *structure* containing the implementation of the module [50]. The types and values of a structure can be imported as a whole structure or as individual items. *Functors* are parameterized structures. Unlike Symmetric Lisp's Kappa form, which is parameterized over individual items in an environment, functors parameterize an environment (structure) over its sub-environments (sub-structures).

In addition to module facilities for interactive languages, we also looked at a few conventional modular languages such as CLU [49], Modula-2 [81], Modula-3 [9], and Ada [79]. 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. This facility allows exporting of different levels of detail of an implementation to different importing modules. Ada supports generic packages. Generic packages, however, can only be instantiated statically through declarations.

2.2 Object-Oriented Languages

This section reviews a few object-oriented languages. Implementation techniques of object-oriented languages are reviewed in Chapter 5.

Smalltalk-80 is a programming system integrated with an object-oriented language and an interactive programming environment [26, 25]. The Smalltalk-80 language is based on ideas of Simula-67. These ideas describe the basic concepts of object-oriented programming and are indicated by a few words - *object*, *message*, *class*, *method*, and *inheritance* [15, 26]. An object consists of some private memory and a set of methods. A message is a request for an object to carry out one of its methods. A class describes the implementation of a kind of similar objects. Smalltalk-80 supports single inheritance, meaning that a class can inherit variables and method from at most one direct super class. A simple denotational definition of its inheritance mechanism based on the concept of *closure* can be found in [57]. Smalltalk-80 is a uniformly object-oriented system. Objects are used to represents numbers, character strings, file directories, and even computer programs like compilers and text editors. Features of the Smalltalk-80 programming environment are presented in the next section.

C++ is an object-oriented extension of the C programming language [40, 70, 71]. C++ requires *virtual functions* to be declared explicitly for messages that have more than one method and provides a visibility control mechanism for variables and methods defined in a class. *Private* variables or methods are only accessible to functions or *friend functions* of the class. *Protected* variables or methods may be used by functions of the class and its subclasses. In addition, C++ supports *generic classes* which are parameterized classes. Like Ada's generic packages, generic classes must be instantiated statically before used. Recent versions of C++ also support multiple inheritance.

The Self programming language is an object-oriented language that has no classes [78]. Object creation and inheritance are facilitated by *prototyping*. Any object in Self can be *cloned* or copied to create new objects, and any object can inherit behaviors from its parent objects. Self does not distinguish state from behavior and does not have variables either. Information about an object is stored in slots of the object and is accessed by passing a message to that object.

CLOS (Common Lisp Object System) [6] and its ancestors Flavors [53] and CommonLoops [5] are three object systems designed for Common Lisp. CLOS, Flavors and CommonLoops all support generic functions and multiple inheritance. A generic function is defined by methods whose definitions are distributed over a chain of inherited classes. A generic function can have different behavior depending on the arguments supplied to it. Calling a generic function serves the same purpose of sending a message to an object. Multiple inheritance allows a class to inherit from more than one direct super classes. CLOS adopts the concept of meta-classes from CommonLoops. Meta-classes specifies the behaviors of classes and allows the user to extend the object system itself. Both CLOS and Flavors support interactive programming. Methods can be defined independently from their classes. Both classes and methods can be modified interactively.

CommonObjects is another object system designed for Common Lisp [63]. CommonObjects differs most significantly from the previous three systems in its emphasis on encapsulation. Most object systems allow objects to be encapsulated with interfaces that define operations on objects. CommonObjects extends the notion of encapsulation to class inheritance: The designer of a class can decide whether the class should be implemented through inheritance or from scratch and change this decision later without affecting client code.

Several systems have been proposed to support object-oriented programming in Scheme. These systems are reviewed in the remainder of this section.

Adams and Rees's proposal [2] uses procedures to represent objects that respond to messages. Instance variables are implemented as lexical variables and can be accessed only by methods defined within an object. Their system does not explicitly support the notion of classes. Multiple inheritance is supported by delegating unsupported messages of an object to its parent objects. In addition, they have developed an efficient implementation technique for method invocation.

Dresher's Object Scheme [18] uses a generalized fluid binding mechanism to represent objects and to support inheritance. Like Adams and Rees's system, Object Scheme does not distinguish between classes and instances. This distinction, if needed, can be established by convention. Object Scheme concisely supports object-oriented programming with only four additional primitives. However, it is difficult to identify instance variables and methods of an Object Scheme object, since this information is not directly specified and must be inferred with bindings in the surrounding lexical environment and their side effects. This drawback makes Object Scheme code difficult to read.

Oaklisp [47] supports object-oriented programming by extending Scheme's function call semantics with support for generic functions and allowing the user to define first-class types. The use of first-class types allows programmers to manipulate the definition of Oaklisp itself and provides a tight integration of user-defined first-class types with existing data types of Scheme.

2.3 Integrated Languages and Environments

The Interlisp programming environment developed during the 1970's contains a set of programming tools for expert programmers [75]. The primary goal of Interlisp is to support experimental programming and structured growth. Some problems in computer science are not well understood and cannot be specified in advanced. The solutions for these problems must evolve through a series of experiments and enhancements. Interlisp supports this program "evolution" process through a set of integrated tools.

Symbolics Genera is a Lisp-based programming environment [80]. The goal of Genera is to support the methodology of evolutionary refinement for software development. Some of the interesting features of Genera are its data-level integration and its open system approach. Data-level integration allows multiple processes to share the same virtual address space and communicate with each other through shared objects in memory. The open system approach does not distinguish between the user and the system. A user program is able to access any routines available in the system. The benefit of the open system approach is enhanced software reusability and extensibility. The drawback of the open system approach is that it violates the information hiding principle [54]. The designers of Genera admit this:

> The system software has been designed with calling interfaces at various levels of abstraction. Unfortunately, the boundaries between the layers of abstraction are mostly invisible, with no formal definition. Both customers and system developers have difficulty choosing a level of abstraction for a problem and staying in its boundaries.

Eiffel is a language and environment for software development [51]. The language supports static type checking, object-oriented programming, exception handling, and systematic use of assertions and invariants. The type system is integrated with the object system by using the inheritance relation to represent the subtype relation. An Eiffel variable of one type may be assigned to a variable of another type if the former is a subtype of the later. Renaming is used to handle name conflicts in multiple inherited classes. In addition to inheritance, which allows information sharing, information hiding is supported by explicitly specifying exported variables of a class. The programming environment provides tools for automatic compilation management, class browsing, document generation, and debugging.

Programming in Smalltalk-80 is supported by a graphical, interactive programming environment. The Smalltalk-80 programming environment supports objects creation and manipulation, program development, and information storage and retrieval. Every object accessible by the user can be presented and manipulated in a meaningful way using a graphical user interface. The graphical user interface uses pointing devices to select objects and invoke messages on the selected objects. Smalltalk-80 supports interactive program development through recompilation. A class can be modified interactively with its subclass automatically recompiled.

Interactive programming environments have been traditionally based on dynamically typed languages such as Lisp or Smalltalk. However, Cedar is an interactive programming environment based on a strongly typed, compiler-oriented language [74, 73]. Cedar uses delayed type binding and a run-time type system to provide the flexibility required by interactive programming. A Cedar user can create multiple instances of UserExecutive interactively. Each UserExecutive has its own state and performs its own operations. One of the operations a UserExecutive can perform is to run an interpreter for interactive programming. Cedar is an open system. Unlike Genera and Interlisp which do not provide explicit layering, Cedar components are organized as layers of modules. Although the Cedar language supports modules, the UserExecutive does not take advantage of the module facility by associating each UserExecutive with an evaluation context of a module. Cedar also provides a sophisticated tree-structured document browsing and preparation tool.

Chapter 3

Semantic Foundations

This chapter describes basic requirements for interactive modular programming and presents the semantics of the λ_{imp} language that satisfies these requirements. The λ_{imp} language extends the λ-calculus by allowing modules to be used to determine the meaning of variable references. The design of λ_{imp} is intended to map to the multi-window user interface model for interactive modular programming.

The first section of this chapter describes basic requirements for interactive modular programming. Section 2 gives an overview of the λ-calculus and its denotational semantics [69, 62]. Sections 3 and 4 illustrate the semantics and present programming examples demonstrating the flexibility of the λ_{imp} language.

3.1 Semantic Requirements

Modular programming provides at least two important functionalities. The first one is to support the information hiding principle. The information hiding principle states that a module should provide to its users all the required information to use the module, and nothing more [54]. The second one is to facilitate the design and organization of programs. A program should be allowed to be decomposed into modules with each module providing services to and receiving services from other modules.

Most modular programming languages support these two functionalities by allowing a module to be defined in terms of three subcomponents: an import specification, public variables and their values, and private variables and their values. A module hides details about its implementation using private variables, provides services by exporting its public variables, and receives services by binding free variable references within the module to imported variables or variables defined in the module.

Interactive programming systems provide flexible ways for their users to develop, test, and modify programs interactively. This flexibility is primarily supported by allowing top-level variables to be redefined interactively without requiring separate steps of recompiling or relinking procedures that use the redefined variables. Interactive programming systems also allow forward references to variables as long as their values are available at run time.

An interactive modular programming system should support both modular programming and interactive programming by:

1. permitting and encouraging information hiding,
2. assisting the design and organization of programs, and
3. allowing private variables, public variables, and imports of modules to be modified interactively.

In addition, an interactive modular programming system should also support forward module references, *i.e.*, allow a module to refer to modules before their definitions. Forward module reference allows recursive module linkage and permits more flexible module structures than the traditional tree-like module structure supported by most modular programming languages.

3.2 The λ-Calculus and its Semantics

The λ-calculus is a language defined with notations and conversion rules that allow the evaluation of lambda expressions denoting first-class anonymous functions [69]. The syntax of the λ-calculus is presented in Figure 2.

The λ-calculus conversion/reduction rules allow λ-expressions to be reduced or "evaluated" to simpler λ-expressions. The β-reduction rule corresponds intuitively to applying a function to its arguments. The β-reduction substitutes bound variables in the body of a function for the corresponding arguments. The following gives an example of β-reduction:

$$((\mathbf{lambda}\ (x)\ (+\ one\ x))\ two) \Rightarrow (+\ one\ two)$$

The bound variable x in the abstraction is substituted with the variable *two* after "evaluating" the application.

Associated with the β rule are the definitions of free and bound:

- Definition of free identifiers:
  1. x occurs free in x but not in any other identifier;
  2. x occurs free in $(e_0\ e_1)$ if it occurs free in e_0 or e_1 (or both);
  3. x occurs free in $(\mathbf{lambda}\ (y)\ e)$ if x and y are different identifier and x occurs free in e.
- Definition of bound identifiers:
  1. No identifier occurs bound in an expression consisting just of a single identifier;
  2. x occurs bound in $(e_0\ e_1)$ if it occurs bound in e_0 or e_1 (or both);

3. x occurs bound in (**lambda** (y) e) if x and y are the same identifier or if x occurs bound in e.

For example, consider the following expression:

$$((\textbf{lambda}\ (x)\ (x\ \ y))\ x)$$

Both x and y are free in $(x\ \ y)$. However, x is bound and y is free in (**lambda** (x) $(x\ \ y)$). For the complete expression, x occurs both free (in the rightmost occurrence) and bound (in the remaining two occurrences) [69]. Note that in an expression an identifier can be both free and bound. However, each occurrence of an identifier is either free or bound but not both.

Another important concept behind the β rule is the notion of substituting bound identifiers in the body of an abstraction to the argument expression when the abstraction is applied. Naive substitution may result in name clashes among existing bound variables. The following example illustrates an incorrect substitution:

$$\begin{aligned}&((\textbf{lambda}\ (x)\ (\textbf{lambda}\ (\mathit{two})\ (+\ x\ \ \mathit{two})))\ \mathit{two})\\&\Longrightarrow (\textbf{lambda}\ (\mathit{two})\ (+\ \mathit{two}\ \ \mathit{two}))\end{aligned}$$

The α-conversion rule, which corresponds intuitively to consistently renaming the formal parameters of a function in most programming languages, allows identifiers to be renamed to avoid name clashes. The correct substitution of the previous example requires changing the bound variable *two* to some other name to avoid name clashes:

$$\begin{aligned}&((\textbf{lambda}\ (x)\ (\textbf{lambda}\ (\mathit{two})\ (+\ x\ \ \mathit{two})))\ \mathit{two})\\&\overset{\alpha}{\Longrightarrow} ((\textbf{lambda}\ (x)\ (\textbf{lambda}\ (y)\ (+\ x\ \ y)))\ \mathit{two})\\&\overset{\beta}{\Longrightarrow} (\textbf{lambda}\ (y)\ (+\ \mathit{two}\ \ y))\end{aligned}$$

A λ-expression may be reduced using different sequences of reductions. *Normal order* reduction reduces the arguments of a function *after* substituting bound variables in the body of the function with the arguments (β-reduction). *Applicative order* reduction reduces the arguments of a function *before* β-reduction. The following expression does not reduce to normal form with applicative order reduction. However, it terminates with normal order reduction.

$$\begin{array}{l}((\textbf{lambda}\ (x)\ (\textbf{lambda}\ (y)\ y))\\ \quad((\textbf{lambda}\ (x)\ (x\ x))\ (\textbf{lambda}\ (x)\ (x\ x))))\end{array}$$

Applicative order reduction closely resembles call-by-value evaluation of many programming languages. The Church-Rosser Theorem guarantees that different reduction sequences produce the same result as long as they terminate [69].

The λ-calculus was originally invented to study the behavior of functions. However, the λ-calculus by itself is merely a syntactic system defined with transformation rules and does not define semantics. That is, it is unclear what abstract value a λ-expression denotes [69].

Fortunately, the techniques of denotational semantics allow the semantics of most programming languages, including the semantics of the λ-calculus, to be expressed in a formal manner. The *denotational semantics* of a programming language is composed of the syntax, domains, and semantic functions of the language. The syntax of the language describes the appearance of syntactic forms and allows their components to be extracted. Domains are special kinds of sets constructed with certain rules for mathematical consistency. Semantic functions map syntax to values in semantic domains. Semantic functions may be defined by a set of mutually recursive definitions.

Figure 2 presents the semantics of the λ-calculus in the style of denotational semantics. The $\rightarrow$ notation is the function domain constructor. The domain of expressed values contains functions only. The domain of environments maps identifiers to expressed values. The purpose of environments is for determining the meaning of identifiers in expressions. An environment may be constructed by extending an

Syntactic Categories:

i	$\in$	*Ide*	identifiers
e	$\in$	*Exp*	expressions

Abstract Syntax:

$$\begin{array}{lcll} e & ::= & i & \text{identifiers} \\ & \mid & (\textbf{lambda}\ (i)\ e) & \text{abstractions} \\ & \mid & (e\ e) & \text{applications} \end{array}$$

Semantic Domains:

ρ	$\in$	$Env = Ide \rightarrow E$	environments
ϵ	$\in$	$E = (E \rightarrow E)$	expressed values

Semantic Functions:

$$\mathcal{E}\colon Exp \rightarrow Env \rightarrow E$$

$$\mathcal{E}\,[\![i]\!]\rho = \rho\,[\![i]\!]$$
$$\mathcal{E}\,[\![(\textbf{lambda}\ (i)\ e)]\!]\rho = \lambda\epsilon.\mathcal{E}\,[\![e]\!](\rho[i \leftarrow \epsilon])$$
$$\mathcal{E}\,[\![(e_0\ e_1)]\!]\rho = (\mathcal{E}\,[\![e_0]\!]\rho)(\mathcal{E}\,[\![e_1]\!]\rho)$$

Figure 2: Syntax and semantics of the λ-calculus

existing environment. The notation $\rho[i \leftarrow \epsilon]$ constructs a new environment. The new environment is the same as ρ except that it maps i to ϵ. Environments help capture the meaning of β-reduction.

The semantic function $\mathcal{E}$ is a function that maps expressions to a function that maps environments to expressed values. The value of an identifier is obtained by applying the identifier to an environment. The value of an abstraction is a function. The result of applying the function to any argument is obtained by evaluating the body of the abstraction in an environment constructed by extending the definition-time environment with a binding that associates the bound variable of the abstraction with the argument value. The value of an application is obtained by applying the value of its operator to the value of its operand.

3.3 The λ_{imp} Language

The λ_{imp} language extends the λ-calculus with a set of predefined constants and facilities for interactive modular programming. The evaluation order of λ_{imp} is applicative order, since it is intended to be used as the semantic foundation for interactive modular programming with call-by-value programming languages. As a consequence of the extension, λ_{imp} preserves lexical scoping while providing flexible interactive capabilities. Unlike the specifications of most programming languages, which define the semantics of complete and syntactically correct programs, the λ_{imp} language is designed to describe the semantics of complete or partially complete programs composed of statements interactively entered during program development.

Figure 3 presents the syntax of the λ_{imp} language. The syntax of λ_{imp} can be divided into two groups. The first group consists of **module**, **private**, **public** and **with** statements. The first three statements are used to support interactions that a user might perform to define bindings in modules and relations among modules.

The **with** statement allows a user to test a program by evaluating expressions within environments associated with a module. The * indicates zero or more occurrences of the preceding form.

The second group consists of constants, identifiers, abstractions, and applications. These expressions are the λ-calculus expressions and are used to describe the actual computation performed in a program.

The syntax of λ_{imp} is simple. It can be further simplified in a window-based programming environment with multiple read-eval-print contexts. In particular, the module name m 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.

Figure 4 presents the domains of the λ_{imp} language. These domains include environments, module environments, constant values, and expressed values. Environments are functions that map identifiers to expressed values. Module environments are functions that map identifiers (module names) to modules. Each module has three components: a private environment, a public environment, and its imports. The imports of a module consists of zero or more identifiers which are the names of other modules. Expressed values are the union of constant values and function values. Unlike function values of the λ-calculus, which are functions of a single argument, the function values of the λ_{imp} language take a module environment as an additional implicit argument.

The concept of free and bound identifiers is essential to the λ-calculus. Free and bound identifiers help determine meanings of identifiers occurring in expressions. Before illustrating the semantics of the λ_{imp} language, the definition of free and bound identifiers for the λ_{imp} language is first presented as follows:

- Definition of free and bound identifiers:

$$
\begin{array}{rcll}
c & \in & \mathit{Con} \quad = \{\text{undefined}, +, -, \ldots, 0, 1, \ldots\} & \text{constants} \\
m,\ i & \in & \mathit{Ide} & \text{identifiers} \\
s & \in & \mathit{Stmt} & \text{statements} \\
e & \in & \mathit{Exp} & \text{expressions}
\end{array}
$$

$$
\begin{array}{rcl}
s & ::= & (\textbf{module}\ m\ \textbf{import}\ m^*) \\
 & | & (\textbf{public}\ m\ i\ e) \\
 & | & (\textbf{private}\ m\ i\ e) \\
 & | & (\textbf{with}\ m\ e) \\
 & | & s\ s \\
e & ::= & c \\
 & | & i \\
 & | & (\textbf{lambda}\ (i)\ e) \\
 & | & (e\ e)
\end{array}
$$

Figure 3: Syntax of λ_{imp}

$$
\begin{array}{rcll}
\rho & \in & \mathit{Env} = \mathit{Ide} \rightarrow E & \text{environments} \\
\mu & \in & \mathit{MEnv} = \mathit{Ide} \rightarrow (\mathit{Env} \times \mathit{Env} \times \mathit{Ide}^*) & \text{module environments} \\
 & & C & \text{constant values} \\
\epsilon & \in & E = C + (\mathit{MEnv} \rightarrow E \rightarrow E) & \text{expressed values}
\end{array}
$$

Figure 4: Semantic domains of λ_{imp}

1. The **module** statements do not evaluate expressions and have neither free nor bound identifiers.
2. If an identifier occurs free in the expression part of a **private**, **public**, or **with** statement then it occurs free in the statement as a whole. If an identifier occurs bound in the expression part of a **private**, **public**, or **with** statement then it occurs bound in the statement as a whole. The identifier in the second syntactic position of any of these statements is neither free nor bound, and the identifier in the third syntactic position of the **public** or **private** statement is neither free nor bound.
3. The rules for determining free and bound occurrences of identifiers in expressions is the same as the rules defined in the λ-calculus.

Figure 5 presents the semantic functions of the λ_{imp} language. We assume that an initial environment (represented by ρ_0) maps every identifier to the undefined value, and initially a module environment maps every identifier to a triple consisting of an initial environment for private bindings, an initial environment for public bindings, and an empty string indicating no imported modules. The **module** statement maps a module name to the module's private bindings, public bindings, and the newly specified imports. A **private** or **public** statement evaluates its expressions and return a new module environment that contains the new binding in the corresponding module's private or public environment. These three statements all return a sequence consisting of a new module environment and the "undefined" value.

The **with** statement evaluates its expression using a module environment, a module name, and an initial lexical environment. It returns a sequence consisting of the unchanged module environment and an expressed value. Consecutive statements are evaluated by passing the module environment returned from the previous statement to the next statement.

Notations:

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

Semantic functions:

$\mathcal{S}$: *Stmt* $\rightarrow$ *MEnv* $\rightarrow$ *(MEnv* $\times$ *E)*
$\mathcal{K}$: *Con* $\rightarrow E$
$\mathcal{E}$: *Exp* $\rightarrow$ *Menv* $\rightarrow$ *Ide* $\rightarrow$ *Env* $\rightarrow E$

$$\mathcal{S}\,[\![(\textbf{module}\ m_0\ \textbf{import}\ m^*)]\!]\ \mu =$$
$$\langle \mu[m_0 \leftarrow \langle \mu m_0 \downarrow 1, \mu m_0 \downarrow 2, m^* \rangle], \text{undefined} \rangle$$
$$\mathcal{S}\,[\![(\textbf{private}\ m\ i\ e)]\!]\ \mu =$$
$$\langle \mu[m \leftarrow \langle (\mu m \downarrow 1)[i \leftarrow \mathcal{E}\,[\![e]\!]\mu m \rho_0], \mu m \downarrow 2, \mu m \downarrow 3 \rangle], \text{undefined} \rangle$$
$$\mathcal{S}\,[\![(\textbf{public}\ m\ i\ e)]\!]\ \mu =$$
$$\langle \mu[m \leftarrow \langle \mu m \downarrow 1, (\mu m \downarrow 2)[i \leftarrow \mathcal{E}\,[\![e]\!]\mu m \rho_0], \mu m \downarrow 3 \rangle], \text{undefined} \rangle$$
$$\mathcal{S}\,[\![(\textbf{with}\ m\ e)]\!]\ \mu = \quad \langle \mu, \mathcal{E}\,[\![e]\!]\mu m \rho_0 \rangle$$
$$\mathcal{S}\,[\![s_0\ s_1]\!]\ \mu = \quad \mathcal{S}\,[\![s_1]\!]\,(\mathcal{S}\,[\![s_0]\!]\ \mu) \downarrow 1$$

$$\mathcal{E}\,[\![c]\!]\mu m \rho = \mathcal{K}\,[\![c]\!]$$
$$\mathcal{E}\,[\![i]\!]\mu m \rho = lookup\ \mu\ m\ \rho\ i$$
$$\mathcal{E}\,[\![(\textbf{lambda}\ (i)\ e)]\!]\mu_0 m \rho =$$
$$\lambda \mu_1.[strict(\lambda \epsilon.\mathcal{E}\,[\![e]\!]\mu_1 m(\rho[i \leftarrow \epsilon]))]\ in\ E$$
$$\mathcal{E}\,[\![(e_0\ e_1)]\!]\mu m \rho =$$
$$(\mathcal{E}\,[\![e_0]\!]\mu m \rho \mid (MEnv \rightarrow E \rightarrow E))\mu(\mathcal{E}\,[\![e_1]\!]\mu m \rho)$$

Figure 5: Semantic functions of λ_{imp}

In addition to modifying a module's private bindings, public bindings, and imports directly, a few features of the semantics of **lambda** expressions, applications, and identifiers are essential to the semantics of the λ_{imp} language:

1. The semantics requires call-by-value evaluation. This is indicated by the use of the *strict* function in the semantic function of abstractions.

2. The lexical environment is closed in the closure returned from an abstraction.

3. The name of the module in which the abstraction is evaluated is also closed in the closure returned from evaluating the abstraction.

4. The module environment is *not* closed and is dynamically provided as an implicit argument when closures are applied.

5. The value of an identifier is determined with a lexical environment, a module name, and a module environment. The module name may be closed previously and may not be the current module name. The search for an identifier's value starts from the lexical environment, followed by the private and public bindings of the module corresponding to the module name, and then by the public bindings of modules imported by the module.

The essence of the semantics is that bindings of free occurrences of identifiers in a function defined with statements are determined dynamically using the use-time module environment. However, the dynamic search is confined to the three use-time components (private environment, public environment, and imports) of the module corresponding to the module name closed at the time when the function is defined. Since the user can modify the bindings and imports of each module in a program, free occurrences of identifiers in a module are sensitive to user's interactions. In addition, bound occurrences of identifiers are determined with the lexical environment.

```
$lookup$: $MEnv \rightarrow Ide \rightarrow Env \rightarrow Ide \rightarrow E$
$lookupM$: $MEnv \rightarrow Ide^* \rightarrow Ide \rightarrow E$

$lookup\ \mu\ m\ \rho\ i =$
    **if** $\rho$ i = undefined **than**
        **if** $(\mu m \downarrow 1)\ i$ = undefined **then**
            **if** $(\mu m \downarrow 2)\ i$ = undefined **then**
                $lookupM\ \mu\ (\mu m \downarrow 3)\ i$
            **else** $(\mu m \downarrow 2)\ i$
        **else** $(\mu m \downarrow 1)\ i$
    **else** $\rho\ i$

$lookupM\ \mu\ [\![]\!]\ i$ = undefined
$lookupM\ \mu\ [\![first\ rest^*]\!]\ i =$
    **if** $(\mu first \downarrow 2)\ i$ = undefined **then**
        $lookupM\ \mu\ rest^*\ i$
    **else** $(\mu first \downarrow 2)\ i$
```

Figure 6: Auxiliary functions

The resulting semantics preserves lexical scoping while providing flexible interactive capabilities.

Figure 6 presents the auxiliary functions *lookup* and *lookupM*. Although their definitions seem inefficient, a technique is presented in Chapter 4 that shows that the variable lookup semantics of the λ_{imp} language can be implemented efficiently.

One possible way to extend the λ_{imp} language is to allow the **with** statement to be used as an expression as well. Unfortunately, nesting **lambda** and **with** expressions would not preserve the essence of lexical scoping. However, λ_{imp} can be extended by allowing (**with-var** $m\ i$) as another kind of expressions that allows an identifier to be searched in a designated module. Chapter 4 presents the use of this extension to support hygienic macro expansion in IMP.

```
(module A import)
(public A square
  (lambda (x) (+ x x)))

(module B import A)
(public B distance
  (lambda (x y)
    (sqrt (+ (square x) (square y)))))

(with B (distance 3 4)) ⟹ 3.74
(public A square
  (lambda (x) (* x x)))
(with B (distance 3 4)) ⟹ 5
```

Figure 7: Example 1

3.4 Examples

The remainder of this section presents example programs demonstrating interactive use of the λ_{imp} language. These examples have been tested by an interpreter implementing an extended version of the λ_{imp} language that supports functions with zero or more arguments, conditional expressions, and some additional Scheme data types. The interpreter is presented in Appendix A.

The first example is presented in Figure 7. This example involves two modules. The module *A* exports the *square* function. The module *B* imports module *A* and uses the *square* function to calculate the distance between a point and the origin. The user can test individual functions using the **with** statement. Upon realizing that the definition of *square* is incorrect, the user redefines the *square* function and tests it immediately. This example demonstrates that a procedure may be redefined interactively with its new definition bound automatically to free identifiers in functions that use the procedure.

```
(module triangle import)
(public triangle area
  (lambda (x y)
    (/ (* x y) 2)))

(module rectangle import)
(public rectangle area
  (lambda (x y)
    (* x y)))

(module area import triangle)
(private area compute
  (lambda (x y)
    (area x y)))

(with area (compute 3 4)) ⟹ 6
(module area import rectangle)
(with area (compute 3 4)) ⟹ 12
```

Figure 8: Example 2

```
(module A import)
(public A square
  (lambda (x) (* x x)))

(module B import A)
(public B distance
  (lambda (x y)
    (sqrt (+ (square x) (square y)))))

(module compare import B)
(public compare less-than-4?
  (lambda (x y)
    (< (distance x y) 4)))
(private compare square
  (lambda (x) (+ x x)))

(with compare (less-than-4? 3 4)) ==> #f
```

Figure 9: Example 3

The second example is presented in Figure 8. This example involves three modules. Both the *triangle* and the *rectangle* modules export an *area* procedure. The *area* module can import an *area* procedure from either the *triangle* module or the *rectangle* module using the **module** statement. This example demonstrates that a module may interactively modify its imports and that the same identifier may be used as a module name and as a variable name. It also illustrates a potential problem regarding name conflicts with imported variables: The *area* module cannot use the two different *area* procedures of the *triangle* module and the *rectangle* module simultaneously. This problem must be addressed in a practical interactive modular programming system. We will return to this point later.

The third example is presented in Figure 9. This example uses the module A and the module B from Example 1 and adds the *compare* module that exports the

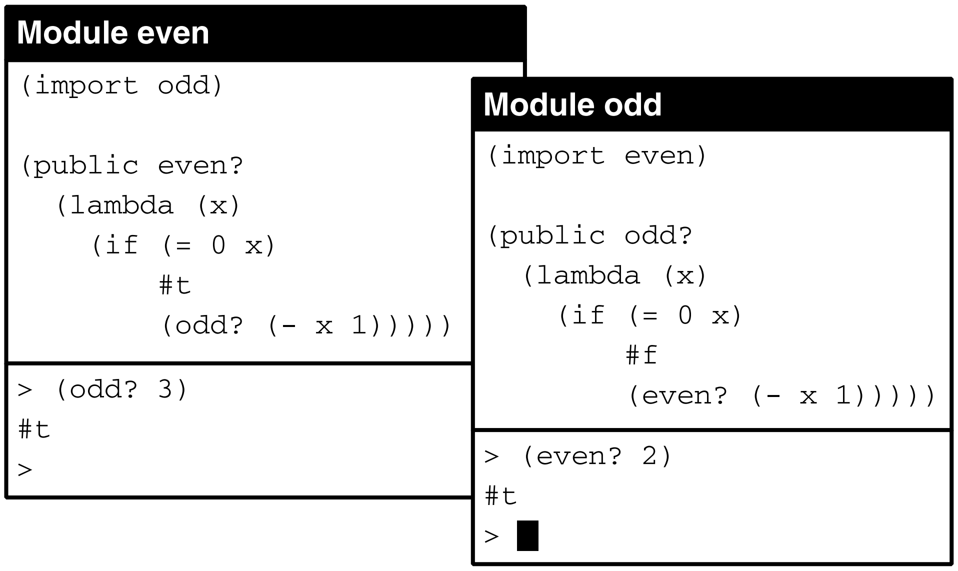
```
(module even import odd)
(public even even?
   (lambda (x)
      (if (= 0 x)
          #t
          (odd? (− x 1)))))

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

(with even (odd? 3)) ⟹ #t
(with odd (even? 2)) ⟹ #t
```

Figure 10: Recursive modules

less-than-4? function which computes whether the distance of a point from the origin is less than 4. The call to the *distance* function in *less-than-4?* indirectly calls the *square* function defined in module A rather than the *square* function locally defined in the *compare* module. This example demonstrates the importance of using the definition-time (closed) module name to look up free identifiers. If the use-time module name were used to look up free identifiers in exported functions, the call to *less-than-4?* would return #t!

The last example is presented in Figure 10. This example uses two recursively-defined modules that import and export to each other to implement the *even?* and *odd?* functions. Note that the **with** statement can also refer to imported identifiers in addition to identifiers locally defined in a module. This example demonstrates that recursive modules can be supported by the λ_{imp} language.

All the examples presented in this section can be mapped to a window-based



```
(import odd)

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

```
> (odd? 3)
#t
>
```

```
(import even)

(public odd?
  (lambda (x)
    (if (= 0 x)
        #f
        (even? (- x 1)))))
```

```
> (even? 2)
#t
>
```

Figure 11: Recursive modules in windows

user interface. As an example, Figure 11 presents the user interface corresponding to the *even* and *odd* modules. Note that the syntax is further simplified and the **with** statement is no longer necessary.

Chapter 4

The IMP System

Chapter 3 discusses semantic foundations for interactive modular programming. This chapter describes the IMP programming system—a practical system designed for interactive modular programming in Scheme [58, 35, 20]. IMP is based on Scheme and the λ_{imp} language.

This chapter addresses many practical issues of supporting interactive modular programming in Scheme. These issues include efficient implementation of variable lookup, separate compilation, project compilation (or delivered-mode code generation), using macros in the context of modules, and programming environment support. The experiences reported in this chapter are applicable to other languages, especially other Lisp dialects, as well.

This chapter is organized as follows. Section 1 gives an overview of the Scheme programming language. Section 2 points out drawbacks of Scheme's modular and interactive programming facilities. Section 3 proposes practical requirements for interactive modular programming in Scheme. Sections 4 and 5 present the design and implementation of the IMP system respectively. Section 6 illustrates problems and presents solutions of incorporating a macro system for IMP.

4.1 An Overview of Scheme

Scheme, a dialect of Lisp, was designed in 1975 at MIT by Guy Lewis Steel Jr. and Gerald Jay Sussman [72]. Scheme is a small language that has a simple syntax and an exceptionally clear semantics [58, 35].

4.1.1 Syntax

Figure 12 presents the syntax of a Scheme subset that is sufficient for our purpose. The notation ... denotes zero or more occurrences of the preceding form.

Scheme supports many data types. A few examples of these data types include booleans, numbers, characters, symbols, lists, vectors, and procedures. Members of any of these data types are referred to as objects.

The purpose of the **quote** expression is to allow lists or symbols to be treated as constants. A list that is not quoted is interpreted as an application and a symbol that is not quoted is interpreted as a variable reference or keyword. The quote character can be used to abbreviate a quoted expression. For example, '(x y) is equivalent to (**quote** (x y)).

Identifiers other than those used as keywords are treated as variable references. Variable references must be bound in the top-level environment or an enclosing **lambda** expression.

A **lambda** expression evaluates to a procedure. Free variables in the procedure body are found in the definition-time environment of the procedure. Bound variables (formal parameters) of the procedure are bound to the arguments (actual parameters) of the procedure when invoked. The syntax for invoking a procedure is wrapping the procedure and its arguments by a pair of parentheses.

An **if** expression has the obvious meaning. A **set!** expression changes the value stored in the location referred to by a variable.

$$\begin{array}{rcll} c & \in & \mathit{Con} & \text{constants} \\ i & \in & \mathit{Ide} & \text{identifiers} \\ e & \in & \mathit{Exp} & \text{expressions} \end{array}$$

$$\begin{array}{rcl} e & ::= & c \\ & | & (\textbf{quote}\ e) \\ & | & i \\ & | & (\textbf{lambda}\ (i\ \ldots)\ e\ e\ \ldots) \\ & | & (e\ e\ \ldots) \\ & | & (\textbf{if}\ e\ e\ e) \\ & | & (\textbf{set!}\ i\ e) \\ & | & (\textbf{let}\ ([i\ e]\ \ldots)\ e\ e\ \ldots) \\ & | & (\textbf{letrec}\ ([i\ e]\ \ldots)\ e\ e\ \ldots) \\ & | & (\textbf{define}\ i\ e) \end{array}$$

Figure 12: Syntax of a Scheme subset

A **let** expression establishes local variable bindings; a **letrec** expression establishes mutually recursive variable bindings. A **define** definition occurring at top level establishes a top-level variable binding.

4.1.2 Semantic Properties

Scheme features lexical scoping and dynamic typing and is applicative order and properly tail recursive. Scheme differs from most other programming languages in that it supports first-class procedures and first-class continuations.

Lexical scoping is a technique for controlling the visibility of variables in a program. A scope is a region of program text determinable by syntactic constructs defined in a programming language. Scopes may be properly nested or completely separated. The primary advantage of lexical scoping is that the visible regions of variable bindings are determined solely by the static structure of the program text and not by the dynamic behavior of a computation. The primary scoping construct of Scheme is the **lambda** expression.

Applicative order and normal order are terms used for describing whether arguments to procedures are evaluated before or after their invocation. Applicative order evaluates the arguments to a procedure before invoking the procedure. Normal order evaluates the arguments to a procedure when they are used by the procedure. Programs that do not terminate in applicative order evaluation can terminate in normal order evaluation. However, applicative order is easier to implement efficiently than normal order and is more appropriate for languages with side effects.

Types help in classifying objects and organizing operations on objects. Statically typed languages determine the types of objects in a program at compile time. Dynamically typed languages, however, defer type checking until run time. The advantages of statically typed languages are that they can be implemented more efficiently and can allow type errors to be discovered before running the program. However, dynamically typed languages are more flexible and more expressive than statically typed languages.

Scheme implementation are required to be *properly tail recursive* [24]. Proper tail recursion allows looping constructs of a language to be implemented with tail-recursive procedures. This feature simplifies Scheme since looping constructs need not be predefined.

Scheme supports both functional and imperative programming. Imperative programming relies on assignments to shared locations to communicate among different parts of a program. Functional programming, however, does not allow assignments; functions communicate only by passing values explicitly. Many researchers have argued that functional programs are more concise and higher level [34]. However, in many situations, assignments are more intuitive, and forbidding their use can significantly complicate solutions of many problems.

One of the most important features of Scheme is that it treats procedures as first-class objects. An object is first-class if it can be passed to and returned from procedures and stored in data structures. A first-class procedure closes the lexical bindings of its free variables when it is created. The values of the free variables may be changed and retained across different invocations of a first-class procedure. This property allows the use of first-class procedures to represent objects with local state that interface with the rest of a program through message passing. First class procedures also support a programming technique known as *currying* [65]. Currying allows a function to be applied in separate steps with each step remembering the results of previous steps and returning a new first-class procedure with the remembered information. Figure 13 gives an example of using the currying technique. First-class procedures also support delayed evaluation, allow the implementation of data structures that compute values on demand, and permit writing programs in the

```
(define add
  (lambda (x)
    (lambda (y)
      (+ x y))))

(define add3 (add 3))
(define add5 (add 5))

(add3 3) ⟹ 6
(add5 3) ⟹ 8
```

Figure 13: Currying

continuation passing style [24, 65].

Scheme also supports first-class continuations. In most programming languages, continuations are merely abstract concepts that capture the notion of "the rest of the computation." At an arbitrary point of the life time of a computation, a continuation can be associated with the point to represent the rest of the computation after the point. In Scheme, continuations can be obtained explicitly, stored in data structures, and later invoked to return to the computation. First-class continuations are particularly useful for implementing coroutines, processes, exception handling mechanisms, and nonblind backtracking [32, 22, 31, 30].

The richness of Scheme is supported only by a small number of core syntactic forms. However, Scheme is not confined by the core forms: additional syntactic forms may be defined using syntactic definitions. Syntactic definitions allow programmers to define new syntactic forms in terms of existing syntactic forms or procedures. For example, **let** may be defined with **lambda** and application:

```
(let ([x v] ...) e e...)
⟹ ((lambda (x ...) e e ...) v ...)
```

Syntactic definitions are also convenient in simplifying otherwise more complicated syntactic patterns.

Most Scheme implementations support interactive programming. Interactive programming reduces program development time by reducing the usual compile-link-execute step, at least conceptually, to a single evaluate step. Interactive programming makes it easier for programmers to experiment with different approaches toward solving a problem and to try out new ideas.

Overall, Scheme is a well-designed programming language that has a small but powerful set of abstraction mechanisms. Together with the simplicity of its syntax and provisions for syntactic extensions, these make Scheme an excellent language for programmers to create and experiment with new computational models or programming paradigms.

4.2 Weaknesses of Scheme

Scheme is both simple and flexible. However, this section illustrates two weaknesses of Scheme. One is Scheme's lack of support for modular programming, and the other is a consistency problem associated with interactive program development in Scheme.

4.2.1 Modular Programming

Lexical scoping is the only name-space management mechanism supported by Scheme. The advantages of lexical scoping are [28, 77]:

1. The programmer can determine the binding of a variable reference by looking at the static program text.

2. The compiler can determine variables' run-time locations at compile time.

```
(define init #f)
(define push #f)
(define pop #f)

(let ([stack '()])
  (set! init
    (lambda ()
      (set! stack '())))
  (set! push
    (lambda (v)
      (set! stack (cons v stack))))
  (set! pop
    (lambda ()
      (let ([v (car stack)])
        (set! stack (cdr stack))
        v))))
```

Figure 14: The stack module

However, lexical scoping also has a number of limitations [82, 28]. One of them is that the lexical structure of a program is forced to be tree like, and an other one is that direct communication among variables in different branches of a program tree is impossible.

Nevertheless, the combination of lexical scoping, assignment statements, and first-class procedures allow Scheme to support restricted name space control mechanisms among program modules. Figure 14 presents an example stack module which exports *init*, *push*, and *pop* procedures while keeping the internal representation of the stack private. However, this programming style has at least three problems:

1. Exported variables are visible everywhere in a program.

2. Variables exported by different modules may result in name conflicts. For example, a queue module may also export the *init* procedure.

3. Individual bindings in a module cannot be modified without reloading or recompiling the entire module. This is inconvenient for interactive programming.

The IMP system presented later in this chapter solves these problems.

4.2.2 Interactive Programming

Scheme is an expression-oriented language that favors experimental and interactive program development. Most Scheme systems realize interactive programming through a program known as the read-eval-print loop. A read-eval-print loop reads one expression, evaluates it, and prints its value, then loops back for more. Changes to variables and objects caused by evaluation of one expression are seen by subsequent expressions. The primary problem with the read-eval-print loop model is that the meaning of a program may be sensitive to the history of an interactive programming session because the programmer may enter commands that cause side effects, resulting in inconsistency between the static program text and the state of the Scheme system. As a result, the programmer cannot completely rely on the static program text to understand a program's behavior [59].

The example in Figure 15 illustrates this problem: The call (*fun* 3 4) right after the definition of *fun* returns (7 . 7). However, after redefining *op*, (*fun* 3 4) returns (−1 . 7) rather than (−1 . −1) which may not be the intention of the programmer, since the programmer may also want to change the value of *op1* as well. The problem is that a procedure can refer to variables whose values may be changed at conceptually different stages during interactive program development and that the effects of these changes can be overlooked by the programmer.

Three approaches may be used to solve this problem. The first one is to reevaluate all the definitions including those that do not rely on the altered bindings. However, this is essentially the traditional batch-oriented programming style. The second one is

```
> (define op +)
op
> (define fun
    (let ([op1 op])
      (lambda (x y)
        (cons (op x y) (op1 x y)))))
fun
> (fun 3 4)
(7 . 7)
> (define op −)
op
> (fun 3 4)
(−1 . 7)
```

Figure 15: The problem of the read-eval-print loop

to require programmers to keep track of the dependencies among top-level definitions and to redefine selected ones manually. However, this approach is unreliable and adds extra burden on the programmer. A third one is to have the system keep track of the dependencies among top-level definitions. After a top-level definition is redefined by the user, the system automatically redefines those affected definitions only. Unfortunately, an appropriate solution to this problem seems to be very difficult since the value of a variable can depend on an arbitrary computation. The example definition defined in Figure 16 illustrates why: If the invocation of the procedure *proc* returns false then *fun* would be sensitive to future modification of *op*, otherwise additional modification to *op* would be used in the procedure returned from the **lambda** expression automatically. Since *proc* is any arbitrary computation (possibly including side effects), it is undecidable whether modifying *op* requires *fun* to be redefined or not. Even assuming the worst case and requiring *fun* to be redefined whenever *op* is redefined, undoing the effect of executing *proc* and preparing the state to reevaluate *fun* could involve a significant amount of overhead. This overhead may

```
(define fun
  (if (proc)
      (lambda (x y) (op x y)
      op)))
```

Figure 16: Should *fun* be reevaluated?

even be greater than the cost of reloading the entire program again. Worse yet, the cost of keeping track of the dependencies and selectively undoing or redefining variable definitions and the cost of reloading the entire program may be extremely difficult to compare. It is therefore unclear which definitions should be reevaluated after an interactive modification.

The module system presented in this section is designed to allow the state of an interactive programming session to be reinitialized in a simple but practical manner (see Sections 4.4.1 and 4.5.4).

4.3 Requirements

Chapter 3 discusses basic requirements for the λ_{imp} language and presents its semantics. This section illustrates practical requirements for the IMP system designed for Scheme.

Interactive programming in Scheme typically involves the following repetitive steps: entering or modifying the source code of a program with an editor, saving the source code to a file, loading the file into the Scheme system, and testing the code using Scheme's read-eval-print loop. Some editors and window managers also allow an expression to be sent to the Scheme system directly from an edit window, thus avoiding the steps of saving and loading files.

One of the requirements of the IMP system is to support flexible and organized program development. A multi-window user interface that associates modules with files and edit windows is useful in organizing a user's activities. This user interface should allow a user to develop and test one or more modules at a time by directly modifying modules displayed in different edit windows. The user interface should also fix the consistency problem associated with traditional implementations of the read-eval-print loop by providing a command that reinitializes the state of a programming session as if the entire program were freshly loaded.

In addition to flexibility, efficiency is also an important consideration. The semantics of the λ_{imp} language employs late binding to determine values of free identifiers dynamically at run time. The IMP system, however, provides more efficient mechanisms to support interactive modular programming.

Most program development projects are initially unstable and require flexible development tools. However, these projects may still need to use modules that are fully developed and have fixed interfaces. A "developed" module should be permitted to compile separately and should run more efficiently. "Developing" modules and developed modules should be allowed to coexist in a project, although developed modules should not be allowed to import from developing modules. Once an entire project is complete, every module in the project would be developed and the entire project can be compiled to more efficient code with the overhead associated with the developing modules eliminated. The IMP system should therefore provide tools that allow a programming project to smoothly move from the developing stage toward the developed stage. During the developing stage, flexibility and discipline are important. At the developed stage, efficiency should become the highest priority.

Much of the power of Scheme stems from macros. The IMP system designed for Scheme should allow macros to be used flexibly and reliably with modules. Unfortunately, programs that use macros need to be macro-expanded before being evaluated. This "early" property makes flexible use of macros in the IMP system difficult and would require modules that use a redefined macro to be reexpanded and reevaluated. However, macros are indispensable for Scheme users and allowing macros to be used reliably with modules is essential for any module system designed for Scheme.

To summarize, an IMP system designed for Scheme should satisfy the following requirements:

1. The IMP system should provide an interactive multi-window programming interface that supports flexible and organized program development. This user-interface should allow its internal state to be synchronized upon the user's request.

2. The IMP system should support separate compilation for developed modules, and project compilation for developed projects. Developed modules and developing modules should be allowed to coexist in a project.

3. The IMP system should allow macros to be used reliably with modules.

4.4 Design

This section presents an integrated approach towards the design of IMP's module system and its programming environment. The module system defines the syntax and semantics of primitive operations for interactive modular programming. The programming environment takes care of translating user interactions into primitive operations of the module system. In order to simplify the presentation, additional language facilities that allow existing macro systems to be incorporated reliably into IMP are presented later in a separate section.

4.4.1 The Module System

id, ex-id, local-id, m-all,			
m-sel, module, variable	$\in$	*Ide*	Scheme identifiers
expression	$\in$	*Exp*	Scheme expressions
lambda-expression	$\in$	*LExp*	Scheme lambda expressions
s	$\in$	*Stmt*	statements

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

Figure 17: Syntax of the module system

The syntax of the module system is specified in Figure 17. The notation $\{x \mid y\}$ indicates the appearance of either x or y but not both. The $^+$ indicates one or more occurrences of the preceding form. The syntactic categories *Ide* and *Exp* are as defined in the Revised^4 Report [58]. The *LExp* are Scheme lambda expressions. This module system extends λ_{imp} by allowing the **import** statement to specify how name conflicts should be resolved among imported variables. As an example, the following expression:

$$(\textbf{import}\ \textit{main}\ (\textit{stack}\ (\textit{queue}\ (\textit{q-init}\ \textit{init})\ \textit{enq}\ \textit{deq})))$$

imports all the public bindings of the module *stack* and only the *init*, *enq*, and *deq* of the module *queue* to the module *main* with *init* renamed as *q-init*.

The semantic domains of the module system are given in Figure 18. The difference between these semantic domains and the semantic domains of the λ_{imp} language is the addition of store locations. The domain *MEnv* associates module names with modules.

L	locations
$MEnv = Ide \rightarrow Module$	module environments
$Module = PriEnv \times PubEnv \times Imports$	modules
$PriEnv = Ide \rightarrow (L \mid undefined)$	private environments
$PubEnv = Ide \rightarrow (L \mid undefined)$	public environments

Figure 18: Semantic domains

Although the module system for IMP extends the module system of the λ_{imp} language in a few different ways, the variable lookup mechanisms of the two module systems are essentially the same. A detailed description of the module system is given in Appendix B.

In order to satisfy the efficiency requirement stated in the previous section and to allow a simple implementation of the refresh-binding command, we employ the following restrictions on IMP's module system:

1. The expression part of a public definition must be a **lambda** expression.
2. Exported and imported variables cannot be assigned (although they may be redefined interactively at top level during program development).
3. Only **import**, **public**, and **private** statements can be loaded from files.
4. The load order of **public** and **private** variables in a module is unspecified.
5. The **import** module statement should be the first statement appearing in a file.

At first glance, it appears that the first two restrictions, which fix the value of exported variables, is too restrictive. However, there is no loss of functionality. Suppose we wish to export the value of the variable x and to allow x to be assignable by other modules. We simply export a reference procedure, *e.g.*, (**lambda** () x) and an assignment procedure, *e.g.*, (**lambda** (v) (**set!** x v)). Suppose we wish to export the value (**let** ([*local-x val-x*]) (**lambda** (*arg*) ...)) of a variable *y*. We can make a private definition (**private** *some-module my-y* (**let** ([*local-x val-x*]) (**lambda** (*arg*) ...)) and a public definition (**public** *some-module y* (**lambda** (*arg*) (*my-y arg*))) which exports *my-y* indirectly.

The primary benefit of employing the first two restrictions is that they encourage 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 that 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.

The implementation of the refresh-binding command is also simplified due to these restrictions. The first restriction does not allow any public definition to close over potentially assignable private state. Together with the second restriction, public definitions need not be reevaluated at all. Expressions appearing in a file usually perform some kind of initialization. This initialization should also be performed during refresh binding. However, maintaining the evaluation order of these expressions is a complicated task. The third and fourth restrictions force the user to put the initialization routines in public or private procedures which can be reevaluated by the

Project management:

```
(load-project name)
(save-project)
(save-project-as name)
(compile-project)
```

Module management:

```
(set-current-module module)
(load-module module string)
(change-order module after)
(compile-modules module)
(compile-one-module module)
(remove-module module)
```

Binding management:

```
(delete-binding module variable)
(refresh-binding)
```

Figure 19: Commands of the IMP system

refresh-binding command.

The fifth restriction prepares the rest of a file with an appropriate evaluation context.

4.4.2 The Programming Environment

The design of the programming environment concentrates on commands for managing projects, modules, and bindings and on ways that the user can interact between the read-eval-print loop and modules displayed in edit windows and stored in files.

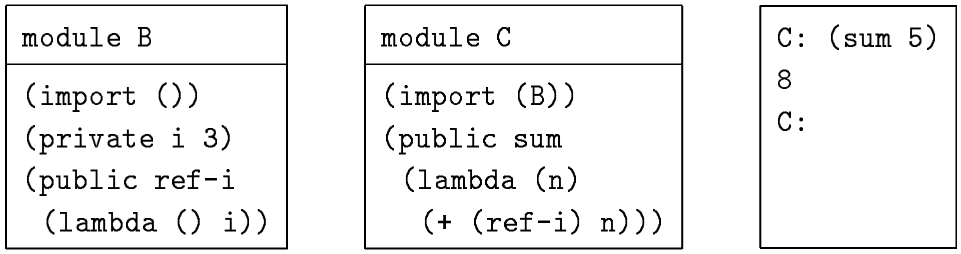
Figure 19 presents a list of commands provided by the IMP system for managing projects, modules, and bindings. An IMP project is composed of zero or more modules. A project may be saved and subsequently loaded. A saved project records the load order of modules and the directory from which to load the modules. The save-project command saves the current project. The save-project-as command saves the current project with a new name. A developed project may be compiled to generate a delivered application.

The set-current-module command causes expressions subsequently entered in the read-eval-print loop to be evaluated in the "current" module. This command allows a single read-eval-print loop to be shared among different modules and saves the programmer from entering the module name required by the statement syntax of the module system. An existing module may be included in the current project by using the load-module command. A new module is added to the current project when it is first set to be the current module. Since Scheme is an imperative language, the system must keep track of the load order of modules in order to implement the refresh-binding command properly. The load order of modules is kept in the *load order list.* By default, new modules are appended at the end of the load order list. However, the change-order command allows the user to change the load order of modules. Existing modules may be located in a specific directory by specifying an optional *path* (a string) to the load-module command. If the path is not specified, the module is loaded from the current directory. Modules may be compiled using the compile-modules command which recursively compiles a module and all of its imported modules, or the compile-one-module command which compiles a single module. Existing modules may be removed using the remove-module command.

Any public or private binding in a module may be removed using the delete-binding command. Invoking the refresh-binding command reinitializes the values of private variables to the values when the variables were last defined. Public bindings need not be reinitialized, since they do not contain local state and cannot be changed by side effects. The order in which private bindings are reinitialized within a module is not specified. The order in which the modules of a project are initialized is the same as the load order of modules.

In addition to commands for managing projects, modules, and bindings, the programming environment also provides a multi-window user interface that allows the user to develop programs in a flexible but organized manner. The best way to describe the user interface is with a few examples.

The following shows a simplified user interface for the IMP system with two modules and a read-eval-print loop:



```
module B
(import ())
(private i 3)
(public ref-i
  (lambda () i))
```

```
module C
(import (B))
(public sum
  (lambda (n)
    (+ (ref-i) n)))
```

```
C: (sum 5)
8
C:
```

The user can create edit windows interactively. After loading the two modules displayed in the left and the center edit windows, the user can test the modules by entering `(sum 5)` in the read-eval-print loop. Since `(sum 5)` is evaluated in module `C` which imports *ref-i* from module B, the system returns 8 as the result. The user interface implicitly supplies the current module name to the read-eval-print loop. The prompt of the read-eval-print loop, which is the name of a module followed by the colon character, indicates the current module.

The contents of an edit window can be modified and can be saved in a file. The user can also send module statements from edit windows to the read-eval-print loop directly without first changing the current module. For example, the user may want to use a locally defined `ref-i` rather than the `ref-i` imported from `B`. In this case, the user can enter the **private** statement that binds `ref-i` in module `C` and send the statement to the the read-eval-print loop directly:

```
module B
(import ())
(private i 3)
(public ref-i
 (lambda () i))
```

```
module C
(import (B))
(public sum
 (lambda (n) (+ (ref-i) n))
(private ref-i (lambda () 1))
```

```
B: evaluating ref-i of module C ...
ref-i
C: (sum 5)
6
C:
```

Note that the current module is changed automatically after sending the statement. Evaluating `(sum 5)` now returns 6. New modules can also be added interactively. For example, the user may prefer the following module structure:

```
module A
(import ())
(private i 2)
(public ref-i
 (lambda () i))
```

```
module B
(import ())
(private i 3)
(public ref-i
 (lambda () i))
```

```
module C
(import (A B))
(public sum
 (lambda (n)
  (+ (ref-i) n)))
```

```
C: (sum 5)
7
C:
```

Since the *ref-i* exported from module A have precedence over the *ref-i* exported from module B, evaluating `(sum 5)` in the context of module `C` returns 7.

An alternative user interface would employ separate read-eval-print windows for each module. This would eliminate the need for the set-current-module command. However, separate read-eval-print windows would occupy additional space on the display screen. Since, in most cases, the user need not explicitly use the set-current-module command, we adopt the design of having a single read-eval-print window with multiple evaluation contexts.

4.5 Implementation

This section describes the implementation of IMP's module system and its user interface. The implementation of the module system requires techniques for compiling developing modules, developed modules, and developed projects. The implementation of the `refresh-binding` command and the user interface are also described.

4.5.1 Compiling Developing Modules

The semantics of our module system requires free-variable bindings 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. A binding in the public or private environment of a module is allocated when a public or private definition of the module is evaluated. A binding in the free-variable environment of a module is allocated 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. An run-time error is signaled if a free variable is used during evaluation but is not bound

```
Module m1:
   (import (m2))

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

   (public fun
      (lambda () act))
Module m2:
   (import (m1))

   (private b 2)

   (public act
      (lambda ()
         (fun)
         (set-a! 3)))
```

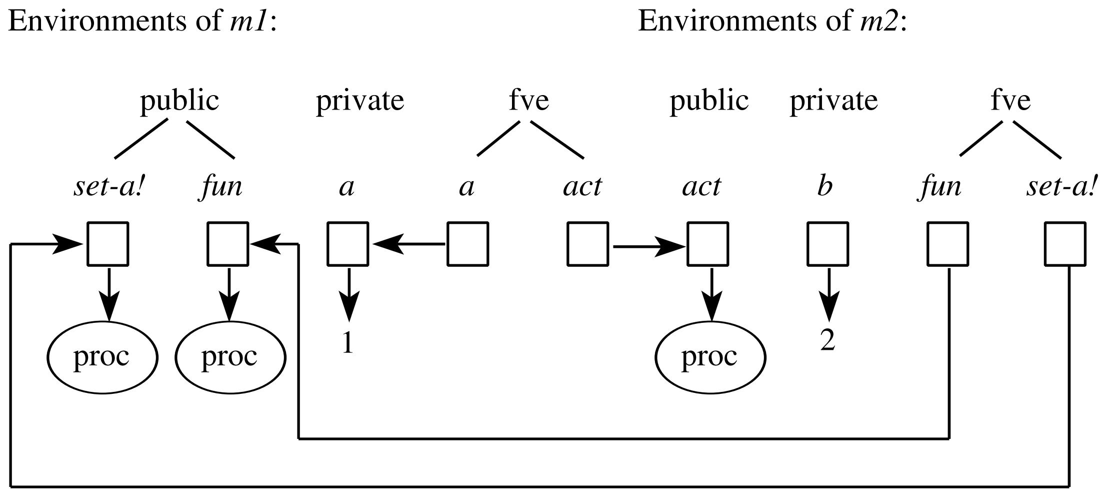


Figure 20: Module linkage

(indirectly) to a value. Figure 20 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 *act* from another module *m3*, the entry *act* in *m1*'s FVE must be changed to point to the *act* 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 20, if the public procedure *fun* is removed from module *m1*, the binding *act* in *m1*'s FVE should also be removed. The system uses reference counts to determine whether free variables should be removed.

4.5.2 Compiling Developed Modules

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 recursive modules, the system must allow loading of a developed module even though the developed module imports some items that

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

(do-binds 'm1 '((pribind a) (pubind fun) (pubind set-a!)))

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

Figure 21: Separate compilation

may not be available at load time.

3. The compiled code should run without the performance penalty associated with developing modules.

The first requirement can be satisfied by having the developed module establish the environments for its public and private variables. The second requirement requires a "delayed linking" mechanism for bindings that are not available at load time. The third requirement can be satisfied by accessing imported procedures in a developed module without using indirection. Furthermore, once a developed module has been compiled, information about its implementation can be exposed for use in optimizing code compiled for other developed modules.

Figure 21 shows the code generated for the module *m1* in Figure 20. Procedures *do-import*, *do-binds*, *do-public*, and *do-private* are provided by the IMP system[1]. The procedure *do-import* checks whether imported modules are also developed. The procedure *do-binds* initializes the private and public environments for the module. The procedures *do-public* and *do-private* put the value of a variable definition into the public or the private environment.

The most interesting part of Figure 21 is the **letrec** expression, which is used to establish bindings for free variables in *m1*. The procedure *get-box* returns a box object (an indirection cell) [21, 43] associated with an entry of the private environment. The box is used to reference or to assign free variables that are bound to locally defined private variables. It seems that allocating boxes for private variables is unnecessary, since they are not visible outside the **letrec** expression. However, the **refresh-binding** command needs to access these private bindings in order to reinitialize them. Allocating boxes for assignable private variables is therefore required to allow both the IMP system and the module to access the private variables. For private variables that are not assignable, boxes are not required.

Any free variable that refers to a locally defined public variable or an imported public variable is bound to an **if** expression that returns the public variable's value if it is available at load time or a procedure which delays the reference of the free variable until the procedure is invoked. The **set!** expression within this procedure changes the delay procedure to an imported procedure when it is first invoked. This *link-by-need* mechanism allows recursive modules to be loaded into the system and provides a more relaxed model for specifying the load order of modules in a project. The procedure *value-getable?* checks whether the value of a variable is available. The procedure *get-value* returns the value of a variable.

[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 unique symbols to avoid being captured by user-defined variables.

Except for the cost associated with the link-by-need mechanism and the use of boxes to access assignable private variables, the code generated for a developed module is as efficient as its corresponding Scheme program. Since many Scheme implementations introduce boxes for assignable private variables, the typical overhead is just the cost associated with the link-by-need mechanism.

4.5.3 Project Compilation

The goal of project compilation is to eliminate all of the 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.

For some applications, this project compilation mechanism is not completely satisfactory. A good example is the IMP system itself. The IMP system takes a user program as input, translates it according to the semantics of the module system, then evaluates the transformed expression. During the translation process, the IMP system generates code that looks like (if (value-getable? 'm 'a) ...), with a subexpression of the translation program coded as `(if (value-getable? ',module-name ',id-name) ...) where value-getable? is quoted as datum[2]. However, value-getable? is the name of a procedure defined in the module link-manager. The variable value-getable would therefore be renamed to link-manager-value-getable? after project compilation. Since

[2]The quasiquote (`) and unquote (,) expressions are a convenient way to specify which part of a subexpression should be quoted or evaluated. `(a ,b) is equivalent to (list 'a b).

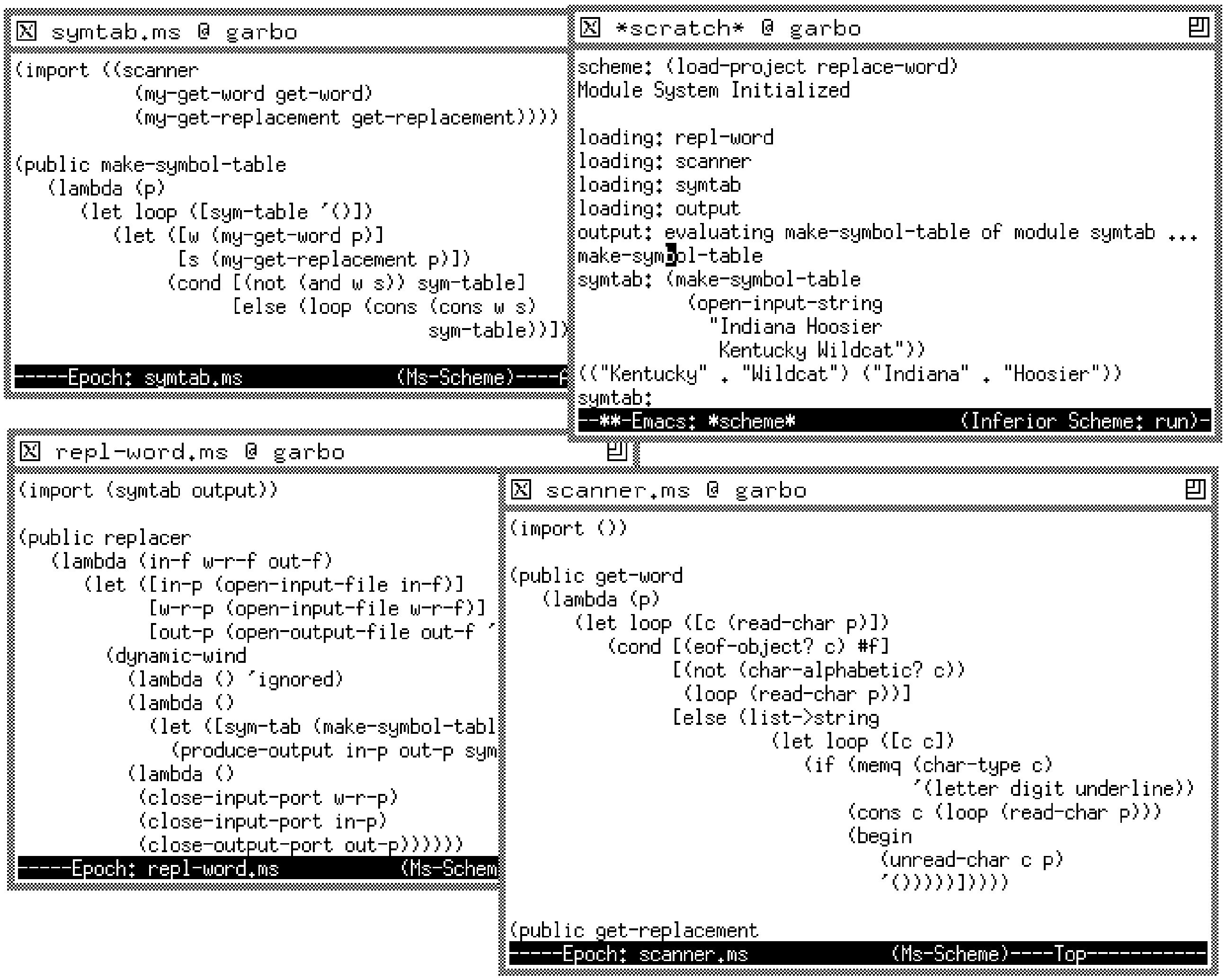
the compiled program uses quoted symbols that depend on variables defined by the program itself, the generated code no longer works. We have not yet found a satisfactory solution to this problem. For some applications, the programmer may be required to manually fix the generated code.

4.5.4 User Interface

A prototype IMP user interface has been constructed by interfacing the Epoch editor (a multi-window variant of the GNU Emacs editor) and the IMP system [38]. The IMP system runs as an inferior process within Emacs and implements commands for manipulating modules, files, and projects. New statements can be added to modules displayed in Emacs windows and can be sent and evaluated in IMP's read-eval-print-loop directly. The IMP system also implements the refresh-binding command. Figure 22 presents IMP's user interface.

The implementation of the refresh-binding command is straightforward. Since the module system does not specify the evaluation order of statements in a module and does not allow assignments to public variables, the refresh-binding command simply restores the most recently defined values for every private definition of every module in the project.

Two kinds of private definitions are possible: the first one binds an identifier to the value of an immediate **lambda** expression, and the second one binds an identifier to the result of evaluating an expression that may refer to or assign private variables at load time. To restore the value of the former, an additional box is used that contains a pointer to the procedure, so that it can be retrieved in case the variable is assigned to other values. To restore a value of the latter, a preprocessed version of the program text is saved, so that it can be reevaluated in an order consistent with the load order of modules in the project.



symtab.ms @ garbo

```
(import ((scanner
           (my-get-word get-word)
           (my-get-replacement get-replacement))))

(public make-symbol-table
   (lambda (p)
      (let loop ([sym-table '()])
         (let ([w (my-get-word p)]
               [s (my-get-replacement p)])
           (cond [(not (and w s)) sym-table]
                 [else (loop (cons (cons w s)
                                   sym-table))])
-----Epoch: symtab.ms            (Ms-Scheme)----F
```

scratch @ garbo

```
scheme: (load-project replace-word)
Module System Initialized

loading: repl-word
loading: scanner
loading: symtab
loading: output
output: evaluating make-symbol-table of module symtab ...
make-symbol-table
symtab: (make-symbol-table
          (open-input-string
            "Indiana Hoosier
             Kentucky Wildcat"))
(("Kentucky" . "Wildcat") ("Indiana" . "Hoosier"))
symtab:
--**-Emacs: *scheme*             (Inferior Scheme: run)-
```

repl-word.ms @ garbo

```
(import (symtab output))

(public replacer
   (lambda (in-f w-r-f out-f)
      (let ([in-p (open-input-file in-f)]
            [w-r-p (open-input-file w-r-f)]
            [out-p (open-output-file out-f '
        (dynamic-wind
          (lambda () 'ignored)
          (lambda ()
            (let ([sym-tab (make-symbol-tabl
              (produce-output in-p out-p sym
          (lambda ()
           (close-input-port w-r-p)
           (close-input-port in-p)
           (close-output-port out-p))))))
-----Epoch: repl-word.ms          (Ms-Schem
```

scanner.ms @ garbo

```
(import ())

(public get-word
   (lambda (p)
      (let loop ([c (read-char p)])
         (cond [(eof-object? c) #f]
               [(not (char-alphabetic? c))
                (loop (read-char p))]
               [else (list->string
                        (let loop ([c c])
                           (if (memq (char-type c)
                                     '(letter digit underline))
                               (cons c (loop (read-char p)))
                               (begin
                                  (unread-char c p)
                                  '())))))])))

(public get-replacement
-----Epoch: scanner.ms            (Ms-Scheme)----Top-----------
```

Figure 22: User interface

4.6 Supporting Macros

Macros are a convenient way to extend the syntax of programming languages. Unlike most Scheme systems which employ a global environment to associate macro names with their definitions, this section presents an approach that incorporates a hygienic macro system in IMP that allows macros to be associated with modules and exported to other modules. This section also discusses problems of using macros with modules and presents some additional language facilities that allow existing macro systems to be incorporated into IMP.

4.6.1 Background

The primary function of a macro system is to allow user-defined macros. Defining a macro causes the macro name and an expander function to be installed in a macro environment. The macro environment associates macro names with expanders. A program known as the macro preprocessor is responsible for expanding the source code of a program using expanders stored in the macro environment. The macro preprocessor scans the input program, and upon encountering an expression that contains a macro call, the expander associated with the macro call is used to expand the expression. This process continues until all the macro calls are expanded. As an example, Figure 23 presents a program that defines the **or** macro, uses the **or** macro in an expression, and shows the result of expanding the expression. The expander for **or** expects a list of three elements as input. The first element of the list should be the name of the **or** macro. The expander uses the remaining two elements of the list to produce an output expression that defines the meaning of the macro call.

This traditional macro mechanism seems to be simple and flexible. However, it suffers from two kinds of variable capturing problems which prevents it from being used reliably [42, 4, 10]:

Defining the **or** macro:

```
(define-syntax or
  (lambda (e)
    `(let ([temp ,(cadr e)])
       (if temp temp ,(caddr e)))))
```

Using the **or** macro:

```
(let ([a #t]
      [b #f])
  (or a b))
```

Result of expanding the above expression:

```
(let ([a #t]
      [b #f])
  (let ([temp a])
    (if temp temp b)))
```

Figure 23: The **or** macro

Using the **or** macro:

```
(let ([temp #t])
  (or #f temp))
```

Result of expanding the above expression:

```
(let ([temp #t])
  (let ([temp #f])
    (if temp temp temp)))
```

Figure 24: The captured binding problem

1. A macro definition may introduce bindings which can capture references of variables in the client program.

2. A macro definition may contain variable references which can be captured by lexical bindings of client program that uses the macro.

The program presented in Figure 24 illustrates an example of the first kind of capturing problems. The expanded code is incorrect since the **or** macro introduces the binding *temp* which captures the reference of *temp* in the code that uses the **or** macro.

In addition to using quoted expressions to specify the output of an expander, pattern language can also be used to allow most macros to be specified conveniently. Figure 25 presents an example which uses a pattern language to define the *push* macro. This example also illustrates the second kind of capturing problems: The expanded code is again incorrect since the meaning of the *cons* introduced by the *push* macro should not depend on the binding of *cons* where *push* is used.

Recently, various techniques have been developed that solve these problems. Kohlbecker, *et al.*, invented *hygienic macro expansion*, which uses "time-stamps" to distinguish between macro-generated identifiers and program identifiers [42]. However,

Defining the **push** macro:

```
(define-syntax push
  (syntax-rules ()
    ((push v x) (set! x (cons v x)))))
```

Using the **push** macro:

```
(let ([stack '()]
      [cons 'a])
  (push 'b stack))
```

Result of expanding the above expression:

```
(let ([stack '()]
      [cons 'a])
  (set! stack (cons 'b stack)))
```

Figure 25: The captured reference problem

Kohlbecker's algorithm runs in quadratic time. Bawden and Rees developed *syntactic closures*, which uses syntactic environments to keep track of the syntactic roles of identifiers [4]. Together with Hanson's "alias" facility, syntactic closures can be used to support high level macros [27]. Unifying and extending ideas from hygienic macro expansion and syntactic closures, Clinger and Rees developed an algorithm that runs in linear time and allows the use of a pattern-based language for writing macros [10]. Hieb and Dybvig have developed a comprehensive macro system that runs with constant overhead, enforces the hygiene condition with a controlled variable capturing mechanism, and maintains referential transparency for all local macros. [33].

Using macros in the context of modules introduces a similar kind of capturing problem: A macro defined in a module may introduce variable references or macro calls that can be captured by bindings or macros defined in modules that use the macro. Figure 26 gives an example program illustrating this problem.

Module *m* defines the **syn** macro and the **plus** macro. Module *n* imports these two macros using the (**import-syntax** (*m*)) statement. Since the **syn** macro defined in module *m* refers to the identifiers **plus** and *minus* and both module *m* and module *n* define their own **plus** macros and their own *minus* procedures, a naive implementation of a macro system may incorrectly use the use-time environments of a macro call to lookup variables introduced by expanding the macro call.

To solve this problem, the macro system must know during macro expansion which module to use to lookup the expander to expand a macro call. When a macro introduces a free reference to a variable, the macro system must also generate information to determine which module to use to determine the binding of the reference. The correct expansion of (**syn** 1 2) in Figure 26 uses the (**with-var** *m minus*) expression. This expression instructs the module system to locate the binding of *minus* using variables defined in or imported by the module *m*. Note that *minus* is a private variable. However it is indirectly exported by a macro. Indirectly exported private variables may be referenced only and cannot be assigned, since the system does not provide a syntactic form (say **with-var!**) to allow a module to change the value of an indirectly imported variable. The reason behind this restriction is to free the programmer and the compiler from looking at other modules to determine whether a given variable is assigned.

Care must be taken when modifying an existing hygienic macro system to support modules. The modification must include module names as part of the context information associated with identifiers in the source code or generated during macro expansion. This added information can be used to look up expanders in the definition-time module. They can also be used to generate **with-var** expressions to link the program.

```
Module m:
  (import ())

  (define-syntax syn
    (syntax-rules ()
      ((syn x y)
       (begin (plus x y) (minus x y)))))

  (define-syntax plus
    (syntax-rules ()
      ((plus x y) (+ x y))))

  (private minus
    (lambda (x y)
      (- x y)))
Module n:
  (import ())
  (import-syntax (m))

  (define-syntax plus
    (syntax-rules ()
      ((plus x y) '(+ x y))))

  (private minus
    (lambda (x y)
      (- (+ x y))))
Correct expansion of (syn 1 2) in n:
  (begin (+ 1 2) ((with-var m minus) 1 2))
Incorrect expansion of (syn 1 2) in n:
  (begin '(+ 1 2) (minus 1 2))
```

Figure 26: Modules and capturing

$$\begin{array}{rcll}
\textit{id, ex-id, local-id, m-all,} & & & \\
\textit{m-sel, module, variable} & \in & \textit{Ide} & \text{Scheme identifiers} \\
\textit{expression, expander} & \in & \textit{Exp} & \text{expressions} \\
\textit{lambda-expression} & \in & \textit{LExp} & \text{Scheme lambda expressions} \\
s & \in & \textit{Stmt} & \text{statements}
\end{array}$$

$$\begin{array}{rcl}
s & ::= & (\textbf{import}\ \textit{module Imports}) \\
 & \mid & (\textbf{import-syntax}\ \textit{module Imports}) \\
 & \mid & (\textbf{define-syntax}\ \textit{module id expander}) \\
 & \mid & (\textbf{public}\ \textit{module variable lambda-expression}) \\
 & \mid & (\textbf{private}\ \textit{module variable expression}) \\
 & \mid & (\textbf{begin}\ s\ s \ldots) \\
 & \mid & (\textbf{with}\ \textit{module expression}) \\
\textit{Imports} & ::= & (\{\textit{m-all} \mid \textit{Select}\} \ldots) \\
\textit{Select} & ::= & (\textit{m-sel}\ \{\textit{id} \mid (\textit{local-id ex-id})\}^{+}) \\
\textit{expression} & ::= & (\textbf{with-var}\ \textit{module variable}) \\
 & \mid & \langle \text{other Scheme expressions} \rangle
\end{array}$$

Figure 27: IMP with macros

4.6.2 Design and Implementation

This section presents an example of incorporating Hieb and Dybvig's hygienic macro system in IMP [33]. The syntax of IMP extended with support for macros is presented in Figure 27.

The syntactic structure of **import** and **import-syntax** are the same. A module can import all the macros from imported modules or selectively import specific macros and rename them to resolve name conflicts. The macros defined in the "scheme" module are automatically imported by every module. Unlike imported variables, imported macros are not sensitive to interactive modifications. The user must load previously expanded and evaluated expressions again after a macro is modified in order to maintain the consistency of the entire program.

The **define-syntax** command defines a macro in a module by associating the name of the macro with its expander in the module's macro environment. Since macros are expanded before being evaluated the expander should not be allowed to use existing program variables. The expander is therefore evaluated in the default "scheme" module. Macros defined with **define-syntax** can be exported. A module does not indirectly export imported macros.

The **begin** syntactic form allows macro writers to group top-level statements together. However, the user should not write order-dependent code in the **begin** syntactic form.

IMP's macro system uses environments of the definition-time module rather than the use-time module to resolve the meaning of macro calls or references of variables introduced by expanding imported macros. As a result, expanding an expression in a module may encounter *foreign macro calls*, and a transformed expression may contain *foreign references.* The expander associated with a foreign macro call can be found in the module of the macro. Foreign references can be translated to **with-var** expressions. Foreign references can refer to other modules' private or public variables. However, side effects to foreign variables are impossible (the system does not provide a syntactic form to support this mechanism).

Figure 28 presents an example program using macros. The macro **mac** is defined in the module *m* and imported by the module *n*. It indirectly calls the **syn** macro which is not imported. The **syn** macro further introduces the private variable *op*. Both **syn** and **mac** introduced by expanding the macro **mac** in module *n* are resolved in the module *m* rather than *n*. This example also shows the use of **begin** to write a macro which defines a private variable and a public procedure that reference the private variable.

Implementing foreign references with the **with-var** expressions is straightforward.

```
Module m:
   (import ())

   (define-syntax mac
     (syntax-rules ()
       ((mac x y) (syn x y))))

   (define-syntax syn
     (syntax-rules ()
       ((syn x y) (op x y))))

   (define-syntax parameter
     (syntax-rules ()
       ((parameter x ref v)
        (begin (private x v)
               (public ref
                 (lambda () x))))))

   (private op +)
   (parameter a ref-a '())

Module n:
   (import ())
   (import-syntax ((m mac)))

   (private op −)
   (private op1 *)
   (private x (lambda () (mac 1 2)))
```

Figure 28: Macros in IMP

They are linked to the free-variable environments of the module that should be used to resolve the foreign references. The foreign reference of *op* introduced by calling the **mac** macro in the module n should therefore be compiled as if it were a free variable of the module m. Supporting separate compilation and project compilation with foreign references are also straightforward.

Chapter 5

Adding The Object System

Object-oriented programming offers a natural way to model the variable nature of many real world applications [68]. The most important feature of an object oriented programming system is that it allows programs to be written in a way that is flexible and extensible. As a result, object-oriented programs are easier to reuse, maintain, and extend. The functionalities of object-oriented programming has prompted many language designers to extend existing languages to support object-oriented programming.

This chapter presents the IMOOP programming system. In addition to supporting interactive modular programming, IMOOP also supports object-oriented programming. The primary benefit of supporting interactive, modular, and object-oriented programming in a programming system is that the programmer can use any or all of these programming styles where appropriate.

IMP provides a flexible module system that supports exporting variables, renaming imported variables, and defining recursive modules. These features allow the design of an object system that offers many advantages over traditional Lisp-based object systems. These advantages include a flexible and efficient slot-access mechanism with strong encapsulation, a simple approach for constructing generic functions, a flexible mechanism for handling conflicts resulting from multiple inheritance, and tight integration of modular programming and object-oriented programming.

This chapter is organized as follows. Section 1 gives background information about object-oriented programming and outlines major features of the IMOOP programming system. Section 2 presents the design of the IMOOP programming system, and Section 3 describes its implementation. The term *object* used in this chapter refers to objects of some classes rather than Scheme objects.

5.1 Background and Overview

Of fundamental importance to object-oriented programming is the concept of inheritance. Inheritance allows existing code to be reused, extended, or specialized for different situations. Most earlier object-oriented languages support single inheritance which restricts a class from inheriting more than one direct super class. More advanced object-oriented languages support multiple inheritance, which allows a class to inherit several direct super classes. Multiple inheritance provides more opportunities for code reuse but it is more difficult to implement efficiently [7, 45].

The two most common approaches to supporting inheritance in object-oriented languages are the class/instance approach and the delegation/prototype approach [48, 67, 68]. The class/instance approach employs class declarations to specify the behavior of instances. A class may inherit from its superclasses. It may also add new *instance variables* or *methods* to extend or specialize the inherited behaviors. The delegation/prototype approach, however, does not use explicit classes to describe inheritance relationships. If an object cannot perform a given operation, it delegates the operation to a "parent" object.

The benefits of the delegation/prototype approach are that it is simple, since it does not need to support class declarations, and more flexible, since an object does not belong to a class and can easily modify its inheritance structure. However, the cost of this simplicity and flexibility is a loss of structure. CLOS, Flavors, CommonLoops, CommonObjects, and Oaklisp are examples of Lisp-based object-oriented languages supporting the class/instance approach [39, 53, 5, 63, 47]. Adams and Rees's proposal and Drescher's proposal for object-oriented programming in Scheme adopt the delegation/prototype approach [2, 18].

Another important feature of object-oriented languages is dynamic lookup of object operations. Syntactically, dynamic lookup of operations of an object may be expressed by passing a message to an object or by passing an object to a generic function. The message-passing protocol perceives an object as a closure with local state that responds to messages. The generic function protocol perceives an object as a block of storage containing a type tag and slot values. Most Lisp-based languages adopt the generic function protocol for two reasons: (1) it conforms to the Lisp function-call syntax, and (2) Lisp objects are also tagged with type information.

The IMOOP programming system presented in this chapter supports multiple inheritance using the class/instance approach, represents objects as blocks of storage, and uses generic functions to support dynamic lookup of object operations. The primary difference between IMOOP and other Lisp-based object-oriented programming systems with similar features is that IMOOP's object system is embedded within a module system. The module system provides utilities that allow a module to define private bindings, export public bindings, and rename imported bindings to resolve name conflicts, and it supports mutual recursive module definitions. These utilities allow a flexible and efficient slot access mechanism without compromising encapsulation, simplify the treatment of generic functions, support flexible handling of conflicts resulting from multiple inheritance, and permit tight integration of modular programming and object-oriented programming.

Encapsulation refers to the degree of independence among different classes [64].

Most Lisp-based object-oriented programming systems provide weak encapsulation by allowing unrestricted access to an object's state. As a result, the object's internal representation cannot be safely modified without affecting other classes. IMOOP's object system restricts access to slots defined in a class to procedures directly associated with the class, and IMOOP's module system provides mechanisms to export these procedures. These two features allow indirect access to an instance's slots from classes other than the instance's class without compromising encapsulation.

Generic functions are defined in terms of methods with the same names across different classes. Most Lisp-based object-oriented systems disallow direct use of methods, since a method cannot in general be referenced unambiguously. These systems provide a default mechanism for combining methods to produce generic functions or allow the user to specify different ways of combining methods. In IMOOP, the renaming capability of the module system allows programmers to use existing methods to define new methods, which eliminates the need to specify method combinations and thus simplifies the treatment of generic functions. The renaming capability also allows programmers to resolve conflicts resulting from multiple inheritance and permits direct invocation of a specific method without going through the generic procedure.

Object-oriented programming and modular programming use different approaches to provide services to client programs. Object-oriented programming offers objects and inheritance while modular programming offers exported interfaces. Some applications are most suitable for object-oriented programming and some other applications are most suitable for modular programming. Allowing a module to be defined as a class allows the programmer to use either programming style where appropriate.

5.2 Design

Conceptually, the design of IMOOP is composed of two interrelated components. One component concerns class definition, instance creation, and operations on instances. The other component concerns mechanisms used for supporting generic functions. In the next subsection, we present the syntactic forms and primitive procedures used for manipulating classes and objects. In Section 5.2.2 and Section 5.2.3, we present two approaches to supporting generic functions.

5.2.1 Classes and Instances

In IMOOP, the basic program building blocks are modules. A module may be defined as a class. A class definition specifies classes inherited by the class and instance variables defined by the class. Instance variables locally defined in a class and inherited from super classes specify the number of slots in an instance of the class. Throughout this chapter, the term *slot* is used to refer to a storage location in an instance, and the term *instance variable* is used to refer to an identifier appearing in a class definition for naming slots in instances.

IMOOP supports a flexible slot-access mechanism with strong encapsulation and a flexible approach for handling conflicts among instance variables resulting from multiple inheritance. Figure 29 presents syntactic forms and primitive procedures defined by IMOOP. The **class** and **make-instance** syntactic forms support class definitions and instance creation. Other syntactic forms and the **type-of** primitive procedures provide operations for accessing and determining the types of instances. The remainder of this section illustrates these facilities using examples that demonstrate IMOOP's flexibility and strong support for encapsulation. The **method** syntactic form is discussed in Section 5.2.3.

The **class** statement defines a class associated with a module. The following class

Syntax:

$$\begin{array}{rcll}
\textit{id, ex-id, local-id, m-all,} & & & \\
\textit{m-sel, module, variable,} & & & \\
\textit{class, instance-variable} & \in & \mathit{Ide} & \text{Scheme identifiers} \\
\textit{expression} & \in & \mathit{Exp} & \text{expressions} \\
\textit{lambda-expression} & \in & \mathit{LExp} & \text{Scheme lambda expressions} \\
s & \in & \mathit{Stmt} & \text{statements}
\end{array}$$

$$\begin{array}{rcl}
s & ::= & (\textbf{import}\ \textit{module Imports}) \\
 & \mid & (\textbf{public}\ \textit{module variable lambda-expression}) \\
 & \mid & (\textbf{private}\ \textit{module variable expression}) \\
 & \mid & (\textbf{class}\ \textit{module Inherits Instance-Var} \ldots) \\
 & \mid & (\textbf{method}\ \textit{module variable module lambda-expression}) \\
 & \mid & (\textbf{begin}\ s\ s \ldots) \\
 & \mid & (\textbf{with}\ \textit{module expression}) \\
\textit{Imports} & ::= & (\{\textit{m-all} \mid \textit{Select}\} \ldots) \\
\textit{Select} & ::= & (\textit{m-sel}\ \{\textit{id} \mid (\textit{local-id ex-id})\}^{+}) \\
\textit{Inherits} & ::= & (\textit{class} \ldots (\textit{instance-variable}\ (\textit{class class class} \ldots)) \ldots) \\
\textit{Instance-Vars} & ::= & \{\textit{instance-variable} \mid (\textit{instance-variable expression})\} \\
\textit{expression} & ::= & (\textbf{with-var}\ \textit{module variable}) \\
 & \mid & (\textbf{make-instance}\ \textit{class}\ (\textit{instance-variable expression}) \ldots) \\
 & \mid & (\textbf{slot-ref}\ \textit{instance instance-variable}) \\
 & \mid & (\textbf{slot-set!}\ \textit{instance instance-variable expression}) \\
 & \mid & (\textbf{type?}\ \textit{instance class}) \\
 & \mid & \langle \text{other Scheme expressions} \rangle
\end{array}$$

Primitives:

$(\textbf{type-of}\ \textit{instance})$

Figure 29: Syntax of IMOOP

declaration occurring in the *account* module declares an *account* class[1]:

(**class** () *owner* (*balance* 0))

The account class inherits no other classes and defines two instance variables. The *owner* instance variable does not have a default initial value, whereas the default initial value for the *balance* instance variable is 0. IMOOP's classes are not first-class objects. The reason behind this restriction is that classes are always associated with modules and modules are not first class. Allowing first-class anonymous modules and classes would completely destroy the static flavor of the system.

Instances of a class are created with **make-instance**. The expression:

(**make-instance** *account* (*owner* "John Doe") (*balance* 100))

creates an *account* instance with its owner slot initialized to "John Doe" and its balance slot initialized to 100, in place of the default initial value. Note that both **class** and **make-instance** are syntactic forms. The class name and slot names in the forms are not evaluated.

A class may inherit other classes and add instance variables of its own. The following declaration in the *joint-account* module defines the *joint-account* class:

(**class** (*account*) *second-owner*)

An instance of *joint-account* contains three slots: *owner*, *balance*, and *second-owner*.

Slots may be accessed with **slot-ref** and modified with **slot-set!**. As in Smalltalk, Oaklisp, and CommonObjects, which restrict the access of instance variables of a class to methods of the class, a procedure defined in a class can refer only to instance variables defined by that class. For example, the *get-owner* public procedure in Figure 30 is legal only if it occurs in the *account* class.

[1]Again, the module name *account* is omitted in the code. We assume that it can be obtained from the window name or the file name of the *account* module.

```
(public get-owner
  (lambda (obj)
    (if (type? obj account)
        (slot-ref obj owner)
        (error 'get-owner
          "Object ~s is not a sub-type of account" obj))))
```

Figure 30: A slot access procedure

The restriction of the use of **slot-ref** and **slot-set!** supports efficient slot access since slot access information can be kept locally in a class and may be sharable by its subclasses (see Section 5.3.1). This restriction also permits strong encapsulation since a class can freely change the internal representation of its instances without affecting client code so long as the changes maintain the external interface of the class [64]. However, slot reference and modification can still be provided via exported procedures. The ability to export/import procedures increases the flexibility of slot access without compromising encapsulation.

Unlike Oaklisp and CommonObjects, which treat instance variables as implicit lexical variables, IMOOP requires explicit use of **slot-ref** and **slot-set!** to access and assign slots. The disadvantage of this approach is the apparent syntactic overhead; however, the benefit is that procedures or methods can accept several objects as arguments. An example procedure presented in Figure 31, which computes the distance between two points, illustrating such a situation.

The **type?** syntactic form asks whether the type of an object is a sub-type of certain class. The inheritance relation also defines the sub-type relation. If class B inherits class A, then an instance of B is a sub-type of B, A, and any of A's super classes. The **type-of** primitive returns the type of an object.

IMOOP supports multiple inheritance by allowing a class to inherit from more

```
(public distance
  (lambda (p1 p2)
    (let ([x1 (slot-ref p1 x)]
          [y1 (slot-ref p1 y)]
          [x2 (slot-ref p2 x)]
          [y2 (slot-ref p2 y)])
      ...)))
```

Figure 31: Accessing slots in two objects

than one parent class. One of the problems resulting from multiple inheritance is that different superclasses may define slots with the same name. Different approaches have been adopted by object-oriented languages to solve this problem. CLOS and Flavors always combine slots with the same name into a single slot. In addition to combining them, CommonLoops can also signal an error for overlapped slot names. CommonObjects does not combine slots. It allows them to be treated separately by using them as implicit lexical variables in methods of the defining class. These approaches are not completely satisfactory, since slots with the same name may mean the same thing sometimes and may mean different things at other times.

Eiffel uses renaming to resolve conflicts among inherited features [52]. The utilities of our module system provide a similar capability. IMOOP treats inherited slots with the same name as separate slots or allows the slots to be combined to form a single slot.

Suppose two classes have been defined for an application. The *teacher* class defines two slots: the *name* slot and the *hours* slot. The *student* class also defines two slots with names identical to the slot names of the *teacher* class. Also suppose that an unexpected change in the specification has required the addition of the *teaching-assistant* class. The *name* slots should be unified while the *hours* slots should be kept separate. The following declaration for the *teaching-assistant* class specifies that the *name* slots of the two classes are combined and their other slots separated:

(**class** (*teacher student* (*name* (*teacher student*))))

Although multiple inheritance in cases like this can be difficult to handle without modifying existing client code, the ability to rename imported procedures allows the public procedures in the *teacher* or the *student* class to be reused to define new procedures.

If a class is inherited from more than one path, the class is treated as if it were inherited only once.

The remainder of this section presents two different ways to support generic functions. First, generic functions may be created manually using recursive modules and the import/export/renaming capabilities of the module system. Second, the **method** declaration form may be used to define generic functions automatically.

5.2.2 User-defined Generic Functions

A generic function is just like an ordinary function except that it is defined in terms of definitions distributed across different classes. Upon receiving its arguments, a generic function decides which one of its definitions to apply dynamically depending on the type of its first argument[2].

IMOOP's module system supports recursive modules and allows renaming of imported identifiers. These two features together with the ability to define classes, create instances, access instances, and determine the types of instances are sufficient to support user-defined generic functions.

Figure 32 presents an example that supports dynamic lookup of object operations with user-defined generic functions. The generic function *distance* is defined in both

[2]Some other Lisp-based object systems also support generic functions that dispatch on multiple arguments. However, individual definitions in these generic functions cannot be associated with a class.

The *point* class:

```
(import ((manpoint (man-distance distance))))
(class () (x 0) (y 0))

(public get-x
  (lambda (obj)
    (if (type? obj point)
        (slot-ref obj x)
        (error 'get-x "Object ~s is not a point" obj))))

(public get-y ... ) ; similar to get-x

(public distance
  (lambda (obj)
    (if (type? obj point)
        (let ([x (slot-ref obj x)]
              [y (slot-ref obj y)])
          (sqrt (+ (* x x) (* y y))))
        (error 'distance "Object ~s is not a point" obj))))

(public closer
  (lambda (obj p)
    (< (man-distance obj) (man-distance p))))
```

The *manpoint* class:

```
(import ((point (point-distance distance) get-x get-y)))
(class (point))

(public distance
  (lambda (obj)
    (if (type? obj manpoint)
        (+ (get-x obj) (get-y obj))
        (point-distance obj))))
```

Figure 32: User-defined generic functions

```
(let ([p1 (make-instance point (x 3) (y 4))]
      [p2 (make-instance manpoint (x 1) (y 5))])
  (closer p1 p2))
```

Figure 33: Calling a generic function

the *point* class and the *manpoint* class using two public procedures [37]. The *distance* procedure defined in the *point* class is exported to the *manpoint* class using the name *point-distance*, and the *distance* procedure defined in the *manpoint* class is exported to the *point* class using the name *man-distance*. The *distance* procedure of the *manpoint* class performs its own distance calculation if its argument is a *manpoint*. Otherwise, it calls the *point-distance* procedure. The two *distance* procedures form a *generic dispatch chain*. It is possible to construct a generic dispatch chain of arbitrary depth. The definition at the end of a generic dispatch chain should always produce an error upon receiving an invalid argument. The definition at the beginning of a generic dispatch chain can access the rest of the methods by climbing up the generic dispatch chain. The main idea behind user-defined generic functions is to use the bottom definition of a generic dispatch chain whenever the complete definition of a generic function is desirable. For example, the procedure *closer* uses the complete definition of the *distance* generic function on its arguments. The expression in Figure 33, which computes which of *p1* or *p2* is closer to the origin, returns `#t`. Note that the ability to support recursive modules is necessary to allow generic functions to be used freely in any module.

One of the major benefits of object-oriented programming is the ability to add new classes and to reuse existing classes. The programming style suggested by the point example does not properly support this desired property. In the point example, the generic function *distance* may need to be modified when a new kind of point is added

The *point* class:

```
(import ((generic (g-distance distance))))
(class () (x 0) (y 0))
. . .
(public distance
  (lambda (obj)
    (let ([x (slot-ref obj x)]
          [y (slot-ref obj y)])
      (sqrt (+ (* x x) (* y y))))))

(public closer
  (lambda (obj p)
    (< (g-distance obj) (g-distance p))))
```

The *manpoint* class:

```
(import ((point get-x get-y)))
(class (point))

(public distance
  (lambda (obj)
    (+ (get-x obj) (get-y obj))))
```

The *generic* module:

```
(import ((point (point-dist distance)) (manpoint (man-dist distance))))

(public distance
  (let ([recent-kind #f]
        [recent-method #f])
    (lambda (obj)
      (if (eq? (type-of obj) recent-kind)
          (recent-method obj)
          (case (type-of obj)
            [point (set! recent-kind 'point)
                   (set! recent-method point-dist)
                   (point-dist obj)]
            [manpoint (set! recent-kind 'manpoint)
                      (set! recent-method man-dist)
                      (man-dist obj)]
            [else (error 'distance "Wrong type of argument ~s" obj)])))))
```

Figure 34: The generic module

to the program. This problem might be fixed by adopting a programming convention that uses an interface module, say *generic*, to import and export the definitions of all the generic functions. With this convention, a new kind of point can be added to the class structure without modifying existing classes. Only the *generic* module must be modified. Figure 34 presents the point example using the *generic* module. For expository purposes, it also shows the use of an explicit cache in the generic function *distance* to improve the efficiency of dynamic lookup.

5.2.3 Automatically Generated Generic Functions

Using the *generic* module to construct generic functions may be acceptable for programs with a small number of generic functions. It may also be useful for customizing the generic dispatch process for improved performance. However, for applications with a large number of generic functions this approach is unacceptable. So we now extend the object system to allow the construction of generic functions automatically.

The **method** statement is used to define a method for a generic function. A method is also a public procedure. Defining a method the first time automatically creates a public generic function in a module specified by the user. The user may use several modules to store generic functions with different purposes. The definition of a generic function may be extended by defining additional methods. Generic functions as well as methods may be imported and renamed as with any other ordinary public procedures. Unlike CLOS and Flavors which use class precedence lists and method combination types to specify the order of method invocation and the roles of *before*, *after*, *primary*, or *around* methods, the renaming facility gives user the ability to compose a new method using existing methods, procedures, or generic functions. This approach is substantially simpler than CLOS and Flavor's approach. IMOOP's methods cannot be created at run time. The reason behind this is that the module system requires methods, which are also public procedures, to be treated in a more static manner to provide greater efficiency and to preserve the flavor of lexical scoping.

Figure 35 presents an example of a generic function defined in terms of method declarations. The generic function *deposit*, which can be imported from *bank-generic*, is jointly defined by two methods. One of them is defined in the *regular-account* class, and the other one is defined in the *now-account* class. The *deposit* method of the *regular-account* class is exported to the *now-account* module which renames it to *regular-deposit*. The *deposit* generic function is exported to the *bank* module without renaming. The *bank-generic* module is automatically created by the system. Unlike ordinary modules, the *bank-generic* module is used for exporting generic functions only. The free variable *regular-deposit* in the *now-account* module refers to the *deposit* method defined in *regular-account*, and the free variable *deposit* in the *bank* module denotes the *deposit* generic function exported from the *bank-generic* module. This example also shows how modular programming can be integrated with object-oriented programming, since *bank* is a module, not a class.

A method defined in a superclass is automatically inherited by its subclasses. Just as object-oriented languages supporting multiple inheritance must resolve conflicts among inherited slots, they must also resolve conflicts among inherited operations. Several approaches may be used to resolve conflicts of inherited operations. Eiffel and Trellis/Owl [61] require the programmer to explicitly resolve the conflict by renaming or redefining the operation in the child class. CLOS, Flavors, and CommonLoops interpret the inheritance graph as a linear chain which can be used with method combinations to resolve conflicts. POOL separates subtyping from inheritance to allow flexible treatment of multiple inheritance for strongly typed languages [3]. A detailed discussion about resolving conflicts of multiple inheritance can be found in [64]. We adopt a weaker version of the Eiffel and Trellis/Owl's approach. A warning message is given if a class inherits from more than one direct superclass that define

The regular-account class:

```
(import ())
(class () owner balance)

(method deposit bank-generic
  (lambda (obj amount)
    (if (> amount 0)
        (slot-set! obj balance
          (+ (slot-ref obj balance) amount))
        (error 'deposit "Cannot deposit negative amount"))))
...
```

The now-account class:

```
(import ((regular-account (regular-deposit deposit))))
(class (account))

(method deposit bank-generic
  (lambda (obj amount)
    (if (>= amount 100)
        (regular-deposit obj amount)
        (error 'deposit "Cannot deposit less than 100 in now account"))))
...
```

The bank module:

```
(import (bank-generic))

(public new-account
  (lambda (kind name amount)
    (case kind
      [regular
        (let ([new-obj (make-instance regular-account (owner name))])
          (deposit new-obj amount)
          new-obj)]
      [now
        (let ([new-obj (make-instance now-account (owner name))])
          (deposit new-obj amount)
          new-obj)]
      [else (error 'new-account "Wrong kind of account ~s" kind)])))
```

Figure 35: Generic functions defined via method declarations

methods with the same name. This warning message indicates which method is inherited by the class. However, the user may redefine the method in the child class to resolve the conflict.

To summarize, the semantics of method inheritance is described informally as follows:

- The *applicable methods* of a class include methods defined in the class and the applicable methods of its direct super classes. Applicable methods of a class define operations on instances of the class.

- If a class redefines an applicable method inherited from one of its direct super classes, the locally defined method shadows the inherited one and is treated as an applicable method of the class.

- A warning message is given if a class inherits applicable methods with the same name defined in different super classes. The warning message indicates which method is inherited by the class. The user can redefine the method to resolve the conflict.

5.3 Implementation

Efficient implementation of object-oriented programming systems is difficult and often involves trade-offs between time and space. Many fast implementation techniques introduce intolerable space overhead, and many space-efficient approaches are often too slow to be acceptable. A good implementation of an object-oriented programming system not only allows fast determination of run-time bindings or addresses but also allows compact storage of information used for the determination process.

IMOOP's object system extends the implementation of its module system with support for instances and generic functions. In this section, we present the implementation of slot access, automatically generated generic functions, separate compilation, and project compilation.

5.3.1 Slot Access

The efficiency of slot access within instances is critical to the performance of object-oriented languages. For those object-oriented languages that support only single inheritance, the mechanism of accessing slots can be implemented efficiently by embedding constant slot offsets in the code. However, for object-oriented languages supporting multiple inheritance, efficient implementation of slot access within an instance is difficult. Depending on the types of instances, the slot offsets of the same instance variable for instances of different classes may be different.

A simple technique for accessing slots within an instance is to store the offsets of all the instance variable of all the classes of a program in a mapping table. The offset of a slot of an instance can then be found by using the enumerated name of the instance variable and the enumerated type of the instance as keys to find the offset in the mapping table. This technique is at least twice as slow as the speed of accessing an ordinary record field. Furthermore, the space overhead of this technique is the total number of instance variables times the total number of classes in a program. This is clearly unacceptable. More sophisticated techniques use hash tables together with caches to reduce the space overhead.

Many researchers have invented various approaches to alleviate the problem. By relying on IMOOP's strong support of encapsulation, we have developed a fast but space efficient technique to access slots. The technique can be used for other object oriented languages with similar features.

Previous Work

Dixon *et al.* suggested an approach which uses a two dimensional table to store slot offsets [17]. Rather than assigning a unique index to each slot name, they use an algorithm that allows different slot names to share the same index. The indices are chosen so that no two slots accessible by the same object have the same index. Since slot indices are not unique, the size of the table is reduced. Pugh and Weddell developed a two-directional record (instance) layout method that allows a fixed offset to be assigned to each field (slot) at compile time with low space overhead [55]. However, both of the two methods are global optimization techniques and cannot be used for interactive program development.

Connor *et al.* described an object addressing mechanism for statically-typed languages with multiple inheritance [11]. Unlike traditional object addressing mechanisms which associate address maps to objects, their approach associates mutable address maps that are associated with variable locations where objects may be assigned. However, their method can be applied only to statically-typed languages.

Borning and Ingalls presented a mechanism of accessing instance state in an implementation of Smalltalk-80 supporting multiple inheritance [7]. The mechanism compiles multiple copies for methods of classes that are involved in multiple inheritance with different offsets.

Krogdahl describes a slot-access implementation technique for a restricted form of multiple inheritance that does not allow a class to be inherited more than once by a common subclass [45]. His technique computes offsets between "reference points" of slots of different classes and uses the offsets to compute slot addresses for instances of different classes.

Most other object-oriented systems use a combination of table look up and indirections to determine the addresses of slots. Flavors stores slot offsets in mapping tables that are associated with each generic functions [53]. Rose gave a detailed analysis about the trade-offs on execution speed, program size, and system flexibility using assembly code with different table lookup strategies using static or relocated tables, large or small table sizes, and larger or small table offsets [60].

Implementation

An instance is implemented as a tagged vector of slots. The tag determines the type of the instance, *i.e.*, the encoded class index (see below). The implementation of instance access uses offsets stored in mapping tables. The implementation takes advantage of the restriction that procedures defined in a class can access only slots owned by that class and of the fact that some classes in a mapping table may share the same mapping information to substantially decrease the size of the mapping table.

Every instance variable of a class and every class in a program are encoded with integer indices. These indices are assigned at load time and are used to locate slot offsets in a mapping table. For each call to **slot-ref** or **slot-set!** within a class definition, the compiler implicitly supplies the mapping table of the class as a hidden argument. The slot index along with an instance's class index can then be used to locate the offset in the mapping table. The class index for an instance is also the type tag for the instance. In general, the class index of an instance will not be the same as the index of the class that contains the **slot-ref** or **slot-set!** form, since the instance may be a member of a subclass of this class. If a program has a fixed class structure, and all the subclasses of a class in the program have the same mapping information, then the offsets for accessing slots in the class can be compiled directly in the code. The implementation of project compilation uses this information to produce more efficient code.

Figure 36 defines four classes with mapping tables and instance structures shown for each class. Since a **slot-ref** or **slot-set!** occurring in class C_0 can access only

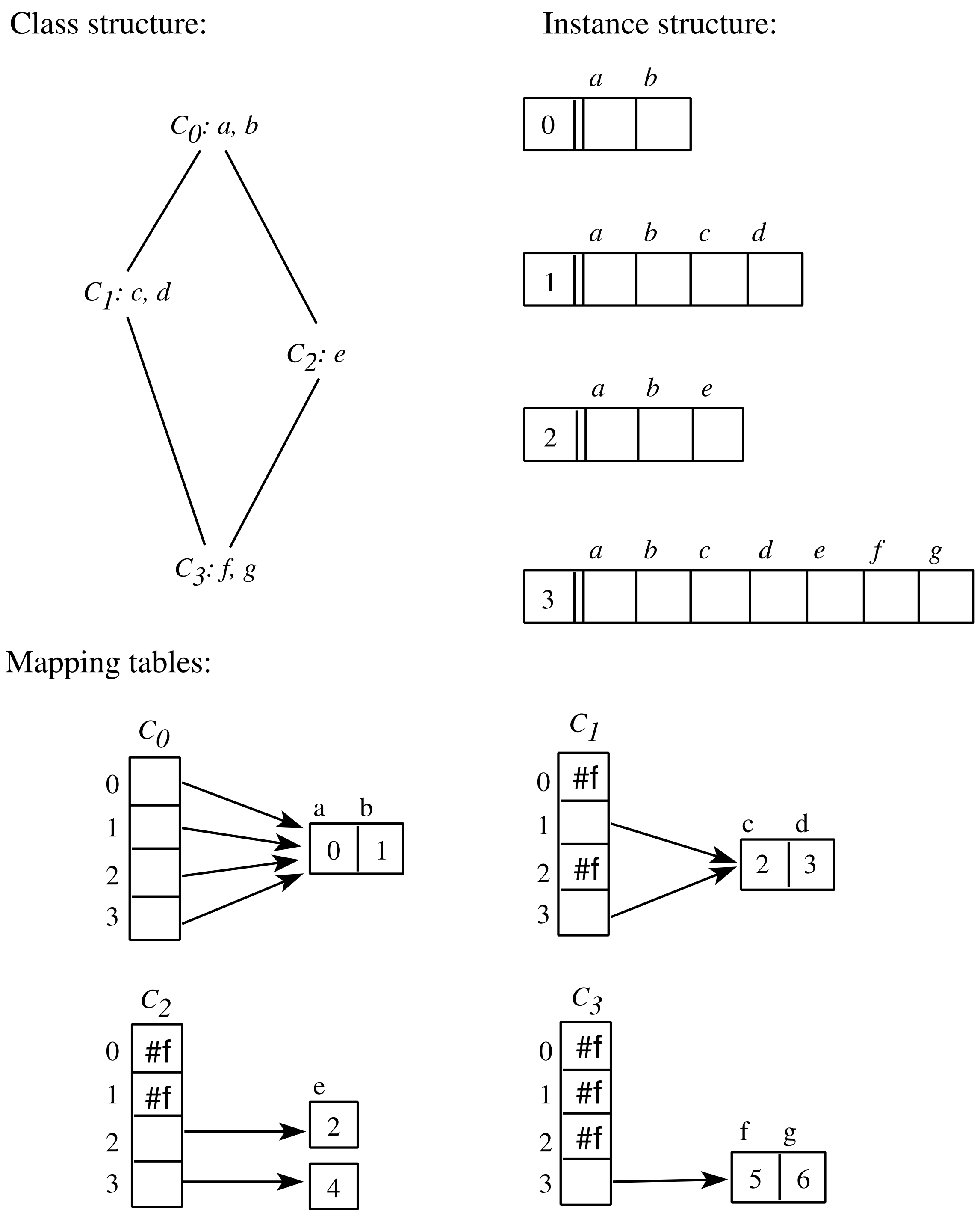


Figure 36: Mapping tables

slot a or slot b of instances of C_0 or instances of its sub-classes, the mapping table of C_0 contains only offsets for a and b for C_0 and each of C_0's subclasses. Also, since instances of C_0 and instances of any subclass of C_0 all place a and b at the same offsets, the mapping information for these slots need not be duplicated.

Class C_2 defines only one slot, e. However, class C_3 inherits both C_1 and C_2. Two different offsets for slot e, one for instances of C_2 and the other for instances of C_3, must be stored. If the class structure is fixed, then the offsets for accessing slots in class C_0, C_1, and C_3 can be compiled in the code.

The entries marked with `#f` in the tables are not wasted. They can be used to implement **type?**, which checks whether an instance is of some type. For example, C_0 and C_2 are not subtypes of C_1, thus, the corresponding entries in C_1's mapping table are marked with `#f`.

Classes may be redefined. Redefining a class automatically updates data structures of the class and affected classes. However, instances created before the redefinition are not modified. The refresh-binding command may have to be used to maintain the consistency of the system. The system also allows terminal classes to be removed.

5.3.2 Generic Functions

Generic functions and message passing are two different approaches to capture the notion of dynamic lookup of object operations. For pure object-oriented languages such as Smalltalk, efficient implementation of dynamic lookup is critical. In IMOOP, the programmer has the flexibility of invoking a method directly without going through the dynamic lookup process. This section presents a simple approach to implementing generic functions.

Previous Work

The most common approach to implementing dynamic lookup is using method caches [44]. Method caches store commonly used methods in a program. Existing implementations of object-oriented programming systems differ on the data structures used in implementing method caches and the places where method caches are used.

A global hash table that stores commonly used methods may be shared by all the generic functions in a program. Hash tables may also be associated with individual classes or individual generic functions [53, 41]. More advanced implementations uses special techniques to control the size and density of these hash tables [41]. In [60], Rose gives a detailed analysis about implementing dynamic lookup on stock hardware using assembly code.

Hash tables associated with individual generic functions may be referred to as *callee caches*. Caches may also be associated with each individual call point of a generic function. This technique may be referred to as *in-line caching* or *caller caches* [16, 5]. In-line caching of method addresses relies on the observation that the locality of type usages in a program is usually high [16].

Some Lisp-based object systems allow generic functions to dispatch on multiple arguments. A common technique for implementing generic functions in these systems is to use different dispatch mechanisms to implement generic functions with a single method, with multiple methods, or dispatch on multiple arguments [5, 41, 19].

Implementation

The implementation of generic functions is based on the observation that any generic function is either *chained* or *unchained*:

1. A chained generic function has all of its methods defined in classes that form a *class chain*.

2. An unchained generic function has its methods defined in classes that cannot form a class chain.

A class chain may contain a single class or two or more classes. A group of two or more classes forms a class chain if their exist a permutation C_i, C_{i+1}, ..., such that, for every i, C_i is a super class of C_{i+1}. Figure 37 presents an example of a chained generic function and Figure 38 presents an example of an unchained generic function.

The classes *c1*, *c3*, and *c4* form a class chain since *c1* is a super class of *c3* and *c3* is a super class of *c4*. The generic function m is thus a chained generic function. The classes *b1*, *b3*, and *b4* do not form a class chain since *b1* and *b4* do not have an inheritance relation. The generic function n is therefore unchained.

The implementation of chained generic functions is based on a technique similar to the binary search. As an example, Figure 39 shows the implementation of the generic function m presented in Figure 37. The variables *c1-m*, *c3-m*, and *c4-m* bind to boxes that contain pointers that point to methods m defined in *c1*, *c3*, and *c4* respectively. The variables *c1-map*, *c3-map*, and *c4-map* bind to the mapping-tables for *c1*, *c3*, and *c4* respectively. The values of these variables are computed by a generic function generator using information about individual methods of the generic function. The procedure *sub-class?* checks whether the class of an object is a sub-class of a class. A class is a sub-class of itself and its super classes.

For a generic function with three methods, this technique takes two comparisons before method dispatch. The advantage of this technique is that the average comparison required for method dispatch is the logarithmic in the actual number of methods that define the generic function. Referring to Figure 37, even though the generic function m is available in five different classes, the property of binary search allows the correct method to be dispatched for objects of each of the five different classes.

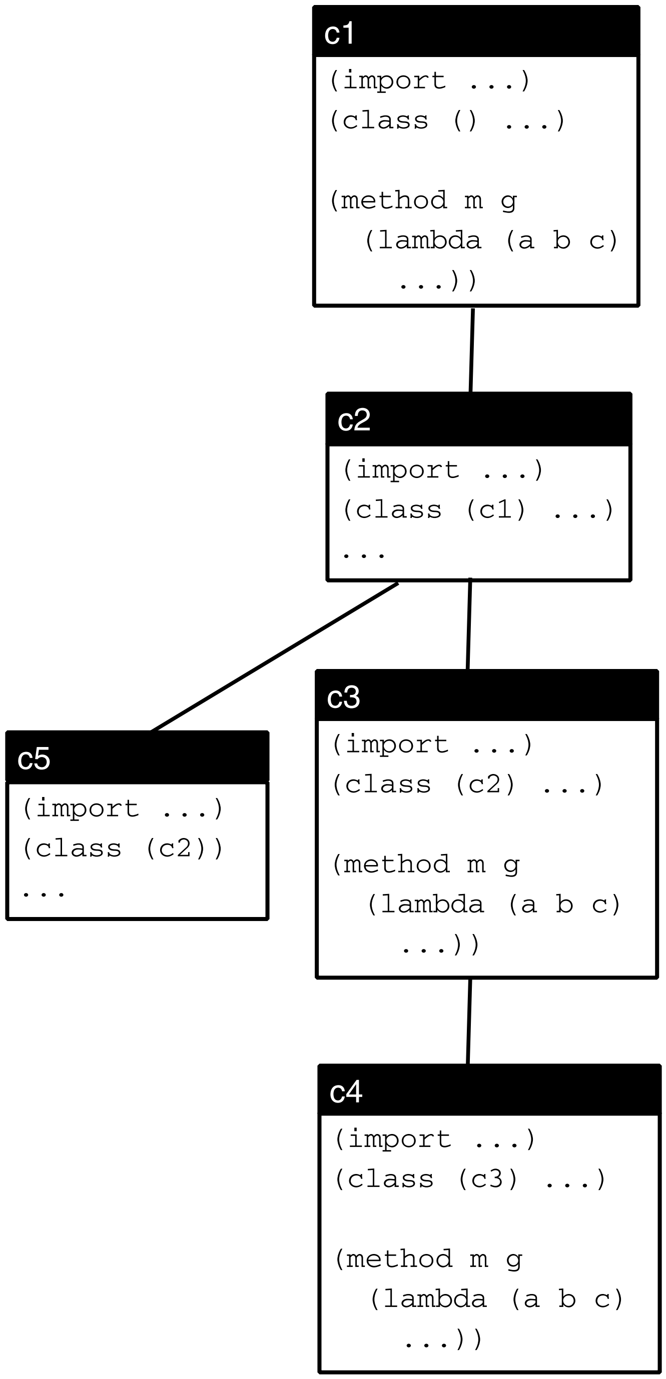


Figure 37: Chained generic function

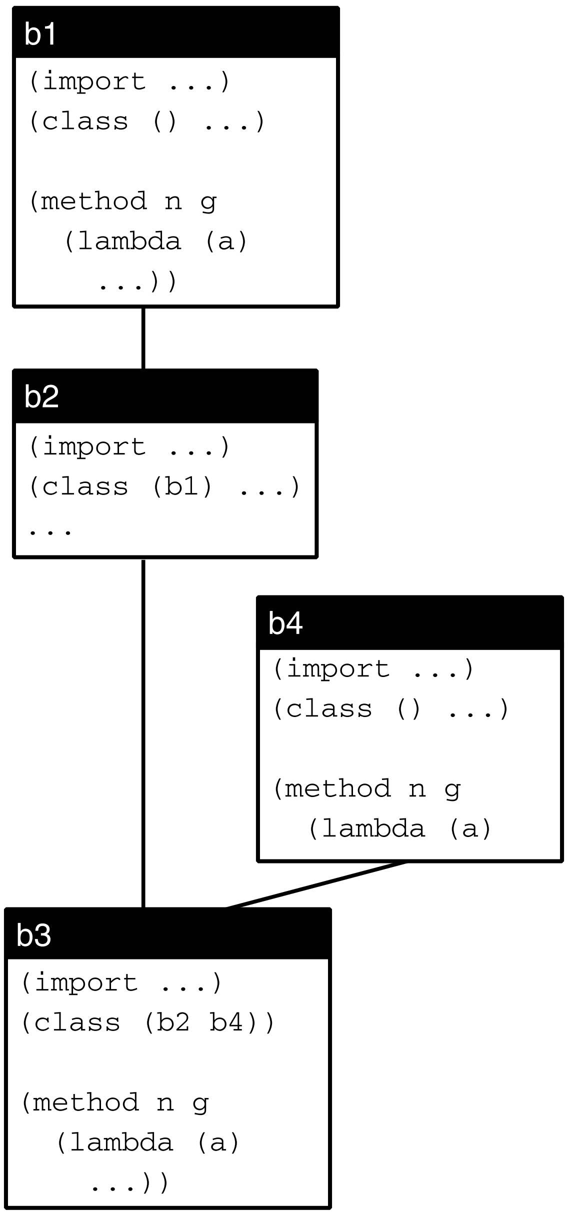


Figure 38: Unchained generic function

```
(let ([c1-m ...]
      [c3-m ...]
      [c4-m ...]
      [c1-map ...]
      [c3-map ...]
      [c4-map ...])
  (lambda (a1 a2 a3)
    (let ([my-class (get-type a1)])
      (if (sub-class? my-class c3-map)
          (if (sub-class? my-class c4-map)
              ((unbox c4-m) a1 a2 a3)
              ((unbox c3-m) a1 a2 a3))
          (if (sub-class? my-class c1-map)
              ((unbox c1-m) a1 a2 a3)
              (error))))))
```

Figure 39: Implementation of chained generic functions

For a generic function with seven methods, three comparisons are need before method dispatch. Since the cost of *sub-class?* is small (a memory reference to the mapping table; see Section 5.3.1) and most caching techniques aim at one to three comparisons before a cache hit, adding caches seems unnecessary for the majority of chained generic functions.

The implementation of unchained generic functions and chained generic functions with a large number (say more than 7) of methods uses a two dimensional table keyed by the name of classes and the name of those generic functions. This approach is based on the assumption that the number of generic functions belonging to this category would be small for most applications. The system can afford the memory overhead associated with the table if the assumption is true in an application.

When a new method is added to a chained generic function, the system checks whether the chain is maintained or if the number of methods in the generic function exceeds a certain limit. The system then updates the generic function with a new generic function depending on the number of methods in the generic function and whether it is chained.

5.3.3 Compiling Developed Classes

In IMOOP, free variables in a developed class or module may refer to three kinds of variables: locally defined private, public, or method variables, imported public or method variables, and imported generic function variables. Unlike developed modules in the IMP system which does not allow import of developing modules and modification of imported variables, IMOOP allows developed modules to import generic modules even though they may change their representation of generic functions. This additional feature requires a different separate compilation strategy for developed classes or modules.

Figure 41 presents the code generated for the *point* class displayed in Figure 40. Since the representation of the generic function *distance* can change after loading the *manpoint* class (which also defines the *distance* method), a box is needed to allow the point class to observe the changes. The procedure *do-class* creates data structures for the new class and returns an integer code assigned to the class. The procedure *get-mapping-table* returns the mapping table of a class. This mapping table is used in the **slot-ref** and **slot-set!** procedures. Note that the call to **slot-ref** in the *distance* method uses the integer codes 0 for x and 1 for y.

5.3.4 Project Compilation

The implementation technique for project compilation is based on the technique used by the IMP system with additional support for class and method declarations.

The IMOOP system provides Scheme code that allows a developed IMOOP project

```
(import ((generic (g-distance distance))))
(class () (x 0) (y 0))

(public get-x ...)

(public get-y ...)

(method distance generic
  (lambda (obj)
    (let ([x (slot-ref obj x)]
          [y (slot-ref obj y)])
      (sqrt (+ (* x x) (* y y))))))

(method closer generic
  (lambda (obj p)
    (< (g-distance obj) (g-distance p))))
```

Figure 40: The *point* class

to create and use class-related data structures. Generic functions are automatically generated. The name of a generic function contains a prefix representing the name of the module that it is exported from. To allow methods to be used as ordinary procedures, a method definition, say *a*, defined in a class, say *c*, is translated to the following Scheme code:

```
(define c:a (lambda ...))
```

If *c:a* is a method of a chained generic function, it is called directly in the generic function. If *c:a* is a method of an unchained generic function or a generic function with the number of its methods exceeding a certain limit, the following Scheme code is generated to install the method in the two dimensional table that is used to dispatch the method.

```
(install-method 'c 'a c:a))
```

```
(do-import 'point '((generic (g-distance distance))))

(do-binds 'point '((method closer) (method distance)
                   (pubind get-y) (pubind get-x)))

(letrec ([+ ...] ...
         [g-distance
          (if (box-getable? 'generic 'distance)
              (get-box 'generic 'distance)
              (box
                (lambda args
                  (if (box-getable? 'generic 'distance)
                      (begin (set! g-distance (get-box 'generic 'distance))
                             (apply (unbox g-distance) args))
                      (error '()
                        "Variable ~s imported from ~s to ~s is not bound"
                        'g-distance 'generic 'point))))))]
         [class-index #f]
         [mt #f])
   (set! class-index (do-class 'point '() `(x ,0) `(y ,0)))
   (set! mt (get-mapping-table class-index))
   (let ([slot-ref
          (lambda (obj slot-code)
            (vector-ref obj
              (vector-ref (vector-ref mt (class-code obj)) slot-code)))]
         [slot-set!
          (lambda (obj slot-code val)
            (vector-set! obj
              (vector-ref (vector-ref mt (class-code obj)) slot-code)
              val))])
     (do-method 'point 'closer 'generic
        (lambda (obj p)
          (< ((unbox g-distance) obj) ((unbox g-distance) p))))
     (do-method 'point 'distance 'generic
        (lambda (obj)
          ((lambda (x y) (sqrt (+ (* x x) (* y y))))
           (slot-ref obj 0) (slot-ref obj 1))))
     (do-public 'point 'get-y ... )
     (do-public 'point 'get-x ... ))))
```

Figure 41: Separate compilation for developed class

Since the definitions of generic functions are fixed in a developed project, the compiler is free to open code any generic function.

For certain classes, the slot-access information for instances of all of their subclasses are the same. The offsets needed for performing **slot-ref** or **slot-set!** in these classes can be compiled directly in the code.

Chapter 6

Conclusion

Most programming languages are designed to meet the needs of application domains or to adhere to certain paradigms such as functional programming or logic programming. With the notable exception of Smalltalk, most programming languages are designed without consideration of the program development environment. This dissertation presents an integrated design of a language and environment that strongly connects modules with files, edit windows, and read-eval-print loops to support interactive, modular, and object-oriented programming. This chapter summarizes major results and suggests future research directions.

6.1 Summary of Major Results

A programming system supporting interactive, modular, and object-oriented programming offers programmers a variety of choices during the design and development of computer programs. Interactive programming allows programmers to make changes easily and dynamically. This feature encourages programmers to explore different approaches toward solving a problem and to experiment with vaguely defined problems in order to better understand the problems.

Modular programming encourages programmers to decompose a program into modules. These modules communicate with one another via predefined interfaces. Module decomposition allows programmers to concentrate on implementing one module at a time. Modules not only increase a program's readability but also make it easier for several programmers to work jointly to solve a problem.

Object-oriented systems provide language facilities to support many real world applications directly. These facilities include classes, objects, and inheritance. Together, they support a programming style that encourages encapsulation, code sharing within a single program, and code reuse among different applications. Object-oriented programming has been proven to be a valuable programming style with many successful applications.

In previous chapters, we have used a series of semantic descriptions, designs, implementations, example programs, and window-based user interfaces to demonstrate that interactive, modular, and object-oriented programming can be integrated coherently in a single programming system. The resulting system allows programmers to use, in the same application, all the facilities for interactive, modular, and object-oriented programming.

The semantics of the λ-calculus always determines the values of free identifiers in a λ-expression using the lexical environment. The semantics of Scheme uses the lexical environment augmented with a top-level environment to determine the values of free identifiers. The λ_{imp} language uses the module environment to serve the purpose of an organized top-level environment and provides the semantic foundation for a multi-window user interface that supports interactive modular programming. The λ_{imp} language demonstrates that the effects of interactive updates can be propagated automatically using a late binding semantics together with a novel method for creating closures.

The design and implementation of the IMP system demonstrate that the concept of interactive modular programming can be supported in a practical and efficient manner. IMP uses double indirections and the import/export relations among modules to support late binding semantics efficiently. The notions of developing modules, developed modules, and developed projects also allow various optimization techniques to be applied to IMP projects to eliminate the remaining overhead. The provision of the **with-var** syntactic form allows hygienic macro systems to be added to IMP.

The design and implementation of the IMOOP system demonstrates that object-oriented programming can be merged with interactive modular programming. IMOOP extends IMP with support for objects, classes, inheritance, and generic functions. The import/export and renaming capabilities of the IMP system not only simplify the design of IMOOP but also allow IMOOP to offer many improvements over existing Lisp-based object systems. These improvements include a flexible and efficient slot-access mechanism with strong encapsulation, a simplified treatment of generic functions, a flexible mechanism for handling conflicts resulting from multiple inheritance, and a programming style that merges interactive, modular, and object-oriented programming.

6.2 Future Work

There are several advanced features supported by a few existing Lisp-based object-oriented programming systems that we have not yet addressed. These features include dynamically changing an instance's class, automatic reinitialization of instances, and the ability to customize the object system's behavior with metaclasses. One area of future work would be to investigate the design and implementation strategies for supporting these features and to study their impact on performance, separate compilation, project compilation, and the overall IMOOP architecture.

The best way to test the IMOOP system is to actually use it to develop large applications. A few good candidates for such applications would be graphical user interface construction tools, visualization tools, or artificial intelligence applications. Actually constructing these applications would provide important feedback on how to improve the functionality of the IMOOP system.

The ability to accumulate and reuse existing code is critical to further advances in software engineering technology [13]. IMOOP supports both modules and classes and allows them to be mixed in the same program. This feature encourages the construction of libraries of reusable modules and classes. The resulting libraries would be more versatile than libraries for languages that support only object-oriented programming or only modular programming.

Appendix A

An Interpreter for the λ_{imp} Language

This interpreter implements λ_{imp} with the following extensions:

1. It supports functions with zero or more arguments.
2. The **if** expression is added.
3. It supports some primitive functions and Scheme data types.
4. It has a simple read-eval-print loop.

Help procedures:

```
(define 1st car)
(define 2nd cadr)
(define 3rd caddr)
(define 4th cadddr)
```

```
(define constants
  `((+ ,(lambda (menv . args) (apply + args)))
    (- ,(lambda (menv . args) (apply - args)))
    (* ,(lambda (menv . args) (apply * args)))
    (/ ,(lambda (menv . args) (apply / args)))
    (sqrt ,(lambda (menv arg) (sqrt arg)))
    (= ,(lambda (menv . args) (apply = args)))
    (< ,(lambda (menv . args) (apply < args)))
    (car ,(lambda (menv arg) (car arg)))
    (cdr ,(lambda (menv arg) (cdr arg)))
    (cons ,(lambda (menv arg1 arg2) (cons arg1 arg2)))))

(define constant?
  (lambda (e)
    (if (or (number? e) (boolean? e))
        #t
        (assq e constants))))

(define constant-value
  (lambda (e)
    (if (or (number? e) (boolean? e))
        e
        (cadr (assq e constants)))))

(define init-menv '())
(define init-lenv '())

(define extend
  (lambda (ids vals lenv)
    (cond
      ((null? ids) lenv)
      (else (cons (cons (car ids) (car vals))
                  (extend (cdr ids) (cdr vals) lenv))))))
```

```
(define lookup
  (lambda (id menv mid lenv)
    (let ([pair (assq id lenv)])
      (if pair
          (cdr pair)
          (let ([mid-mentry (assq mid menv)])
            (if mid-mentry
                (let* ([mentry (2nd mid-mentry)]
                       [pair (assq id (1st mentry))])
                  (if pair
                      (cdr pair)
                      (let ([pair (assq id (2nd mentry))])
                        (if pair
                            (cdr pair)
                            (let ([imports (3rd mentry)])
                              ((rec loop
                                 (lambda (imports)
                                   (if (null? imports)
                                       (error 'lookup
                                         "variable ~s not bound" id)
                                       (let ([mid-mentry
                                              (assq (car imports) menv)])
                                         (if mid-mentry
                                             (let
                                               ([pair (assq id
                                                            (2nd (2nd
                                                              mid-mentry)))])
                                               (if pair
                                                   (cdr pair)
                                                   (loop (cdr imports))))
                                             (loop (cdr imports)))))))
                               imports))))))
                (error 'lookup "module ~s undefinded" mid)))))))
```

```
(define imp
  (lambda ()
    (display "imp: ")
    (let ([first-result (S (expand (read)) init-menv)])
      ((rec loop
         (lambda (result)
           (let ([new-menv (1st result)]
                 [answer (2nd result)])
             (display new-menv) (newline)
             (display answer) (newline)
             (display "imp: ")
             (loop (S (expand (read)) new-menv))))) first-result))))

(define S
  (lambda (stmt menv)
    (cond ((pair? stmt)
           (case (car stmt)
             [(module) (S-module stmt menv)]
             [(private) (S-private stmt menv)]
             [(public) (S-public stmt menv)]
             [(with) (let ([mid (2nd stmt)]
                           [exp (3rd stmt)])
                       (list menv (E exp menv mid init-lenv)))]
             [else (error 'S "bad module command ~s" stmt)]))
          (else (error 'S "bad module command ~s" stmt)))))

(define S-module
  (lambda (stmt menv)
    (let* ([mid (2nd stmt)]
           [imports (cdddr stmt)]
           [mid-mentry (assq mid menv)])
      (if mid-mentry
          (let* ([mentry (2nd mid-mentry)]
                 [new-mentry (list (1st mentry) (2nd mentry) imports)]
                 [new-menv (cons (list mid new-mentry) (remq mid-mentry menv))])
            (list new-menv mid))
          (let* ([new-mentry (list '() '() imports)]
                 [new-menv (cons (list mid new-mentry) menv)])
            (list new-menv mid))))))
```

```
(define S-private
  (lambda (stmt menv)
    (let* ([mid (2nd stmt)]
           [id (3rd stmt)]
           [exp (4th stmt)]
           [mid-mentry (assq mid menv)])
      (if mid-mentry
          (let* ([mentry (2nd mid-mentry)]
                 [e-val (E exp menv mid init-lenv)]
                 [new-menv
                   (let ([new-pri (cons (cons id e-val) (1st mentry))])
                     (cons (list mid (list new-pri (2nd mentry) (3rd mentry)))
                           (remq mid-mentry menv)))])
            (list new-menv e-val))
          (let* ([e-val (E exp menv mid init-lenv)]
                 [new-pri (list (cons id e-val))]
                 [new-menv (cons (list mid (list new-pri '() '())) menv)])
            (list new-menv e-val))))))

(define S-public
  (lambda (stmt menv)
    (let* ([mid (2nd stmt)]
           [id (3rd stmt)]
           [exp (4th stmt)]
           [mid-mentry (assq mid menv)])
      (if mid-mentry
          (let* ([mentry (2nd mid-mentry)]
                 [e-val (E exp menv mid init-lenv)]
                 [new-menv
                   (let ([new-pub (cons (cons id e-val) (2nd mentry))])
                     (cons (list mid (list (1st mentry) new-pub (3rd mentry)))
                           (remq mid-mentry menv)))])
            (list new-menv e-val))
          (let* ([e-val (E exp menv mid init-lenv)]
                 [new-pub (list (cons id e-val))]
                 [new-menv (cons (list mid (list '() new-pub '())) menv)])
            (list new-menv e-val))))))
```

```
(define E
  (lambda (exp menv mid lenv)
    (cond
      ((constant? exp) (constant-value exp))
      ((symbol? exp) (lookup exp menv mid lenv))
      ((pair? exp)
       (case (1st exp)
         [(lambda) (let ([formals (2nd exp)]
                         [body (3rd exp)])
                     (lambda (menv1 . actuals)
                       (E body menv1 mid
                          (extend formals actuals lenv))))]
         [(if) (if (E (2nd exp) menv mid lenv)
                   (E (3rd exp) menv mid lenv)
                   (E (4th exp) menv mid lenv))]
         [else (apply (E (car exp) menv mid lenv)
                      menv
                      (map (lambda (x) (E x menv mid lenv))
                           (cdr exp)))]))
      (else (error 'E "bad expression ~s" exp)))))
```

Appendix B

The IMOOP System

IMOOP is a programming system designed for the Scheme programming language. IMOOP extends Scheme with support for interactive, modular, and object-oriented programming. The IMOOP system can be described by its language definition and its programming environment.

B.1 The Language

An IMOOP program consists of a sequence of statements. These statements can be used to define modules, classes, and their components. A module has three components: an environment for private variables, an environment for public variables, and an import specification. The private and public environments map variables to locations that stores the values of the variables. The import specification specifies which modules are imported by the module and which variables of the modules are imported with which local names in the module. These three components of a module together with Scheme's lexical environment are used to determine the locations of variables in a program. The import specification is interpreted dynamically when determining the location of a variable in a module.

A module may also be defined as a class. A class defines instance variables and methods and inherits instance variables and methods from other classes. IMOOP supports multiple inheritance. A class can inherit from more than one direct super class.

Instance variables (including inherited instance variables) of a class are used as "templates" for creating instances of the class. An instance is a tagged vector of slots. The tag is used to represent the class or the type of the instance. The slots can be accessed with special language facilities using the instance and the names of the instance variables.

Methods of a class (including inherited methods) provide operations for instances of the class. By convention, the first argument of a method should receive an instance object. Methods are public procedures. They can also be used to define generic functions. Generic functions are stored in public environments of user specified *generic* modules. Upon receiving its arguments, a generic function selects a method to apply to its arguments using the tag (class) of its first argument.

Classes can be redefined. Redefining a class automatically updates inherited instance variables and methods of the class and all of its subclasses. However, existing instances of these classes are not modified.

IMOOP also defines expression-level syntax and procedures. These syntax and procedures allows the creation, accessing, and type operations on instances.

We assume every Scheme identifier initially defines a module that contains an empty private environment, an empty public environment, an empty import specification that specifies no imported modules, and no class definition.

Section B.1.1 and B.1.2 are organized into entries. Each entry describes a language feature which is either a syntactic construct or a built-in procedure. The format of an entry follows the Revised^4 Report with the following modifications [58]:

- The header line of an entry starts with a **bold**-faced syntactic keyword or a procedure name.
- The rest of the header line is formatted in an *italic* type style.
- Meta names *module*, *class*, *variable*, *slot*, and *instance-variable* represent Scheme identifiers, *expression* indicates IMOOP expressions which also includes any Scheme expressions, *lambda-expression* represents a lambda expression, *statement* represents an IMOOP statement, and *instance* indicates an instance object.
- The notation $\{form1 \mid form2\}$ represents an occurrence of either *form1* or *form2* but not both. The $^+$ represents one or more occurrences of the preceding form.

B.1.1 Statements

(**import** *module Imports*) statement syntax

Syntax: The syntax of *Imports* is specified as follows:

$$\begin{aligned} \mathit{Imports} &::= (\{\mathit{m\text{-}all} \mid \mathit{Select}\} \ldots) \\ \mathit{Select} &::= (\mathit{m\text{-}sel}\ \{\mathit{id} \mid (\mathit{local\text{-}id}\ \mathit{ex\text{-}id})\}^{+}) \end{aligned}$$

where *m-all* and *m-sel* are module names and *local-id*, *ex-id*, and *id* are variable names. These names are Scheme identifiers.

Semantics: Imports becomes the import specification of the *module.* A module may explicitly import zero or more modules. All the public variables of *m-alls* and only selected public variables of *m-sels* are imported. The selected variables may be assigned local names that may be different from their original names. The *scheme* module is also imported implicitly. Public variables of the *scheme* module includes at least all the essential procedures defined in the Revised^4 Report [58]. *Imports* is interpreted dynamically using the public environments of *m-alls* and *m-sels* when determining locations of variables in the *module.* Name conflicts among variables in imported modules are resolved from left to right (see Section B.1.3).

Examples:

1. (**import** *m* ())
   Module *m* imports all the public variables in the *scheme* module.

2. (**import** *m* (*n* (*o* (x y) *z*)))
   Module *m* imports all the public variables of the module *n* and *scheme* and only the *y* and *z* of the module *o* with *y* renamed as *x*.

(**public** *module variable lambda-expression*) statement syntax

The public environment of the *module* is extended with an entry that binds the *variable* to a fresh location holding the result of evaluating the *lambda-expression* in the *module.* The semantics of evaluating an expression in a module is defined in Section B.1.3.

If the *variable* also has a binding in the *module*'s private environment, the binding is removed from the private environment.

It is an error to assign, *i.e.*, **set!**, public variables.

(**private** *module variable expression*) statement syntax

The private environment of the *module* is extended with an entry that binds the *variable* to a fresh location holding the result of evaluating the *expression* in the *module.*

If the *variable* also has a binding in the *module*'s public environment, the binding is removed from the public environment.

(**begin** *statement statement* ...) statement syntax

The *statements* are evaluated in an unspecified order.

(**with** *module expression*) statement syntax

Returns the result of evaluating *expression* in the *module.*

(**class** *module Inherits Instance-Var* ...) statement syntax

Syntax: The syntax of *Inherits* and *Instance-Var* are specified as follows:

$$\begin{aligned} \textit{Inherits} &::= (\textit{class} \ldots (\textit{instance-variable}\ (\textit{class class class} \ldots)) \ldots) \\ \textit{Instance-Var} &::= \{\textit{instance-variable} \mid (\textit{instance-variable expression})\} \end{aligned}$$

Semantics: The *module* is also defined as a class. The **class** statement defines instance variables and inherits *effective instance variables* and *applicable methods* from its direct super classes. The number of effective instance variables of a class determines the number of *slots* of an instance of the class. The applicable methods of a class define operations on instances of the class. Redefining existing classes is allowed.

Inherits explicitly specify *direct super classes* and *combined instance variables* of the *module.* The direct super classes must exist in the system already. A combined instance variable specifies that the instance variables defined in several different classes with the same names are combined into a single instance variable. The effect of combining instance variables is that a single slot is used for all the instances variables that are combined.

A class may define zero or more instance variables. An instance variable may be defined by a single identifier with an unspecified default initial value for its slots, or a two element list. The first element of the list must be an identifier. The second element of the list is an IMOOP expression. The value of the expression, evaluated at instance creation time, is the default initial value for slots named by the identifier. Only locally defined instance variables in a class can be used as arguments to **slot-ref** or **slot-set!** forms occurring in the class (see Section B.1.2).

Semantics of instance variable inheritance:

- The effective instance variables of a class include instance variables defined in the class and effective instance variables (after combining) inherited from its direct super classes.
- The default initial values of slots associated with combined instance variables are unspecified.
- The effective instance variables of direct super classes with the same names and originating from the same classes are automatically combined and inherited.

Semantics of method inheritance:

- The applicable methods of a class include methods defined in the class and applicable methods inherited from its direct super classes.
- If a class redefines an applicable method inherited from one of its direct super classes, the locally defined method shadows the inherited one and is treated as an applicable method of the class.
- A warning message is given if a class inherits applicable methods with the same name from different super classes. The warning message indicates which method is inherited by the class. The user can redefined the method in the class to resolve the conflict.

Examples:

1. (**class** *w* () *a*)
   Module *w* is defined as a class which inherits no other class and defines an instance variable *a*.
2. (**class** y () (*a* 0))
   Module *y* is defined as a class which inherits no other class and defines an instance variable *a*. The default initial value for slots associated with *a* is 0.

3. (**class** *z* (*w*) *b* *c*)
   Module *z* is defined as a class which inherits applicable methods and effective instance variables of *w* and defines instance variables *b* and *c*.

4. (**class** x (y *z* (*a* (y *w*))))
   Module *x* is defined as a class which inherits applicable methods and effective instance variables of *y* and *z*. Class *x* does not define instance variables. Both *y* and *w* defines an instance variable named *a*, these two instance variable is combined into a single instance variable which is inherited by *x*. The initial value for slots associated with *a* is unspecified. Note that *w* is a super class and not a direct super class of *x*.

(**method** *module variable g-module lambda-expression*) statement syntax

Module must be defined as a class. Evaluating a **method** statement has the effect of evaluating the statement as a **public** statement, plus the method is inherited by subclasses of *module* using the rules of method inheritance, plus the method is installed in a public generic function named by *variable* in *g-module*. Upon receiving arguments, this generic function invokes the method if the method is an applicable method of the class of its first argument. *G-module* is automatically created by the IMOOP system when it is first used in a **method** statement.

B.1.2 Expressions and Essential Procedures

IMOOP expressions include all the Scheme expressions plus the syntactic forms and procedures defined in this section.

(**with-var** *module variable*) expression syntax

Returns the value of evaluating the *variable* in the *module*. The *variable* and the *module* are treated as symbols.

(**make-instance** *class (instance-variable expression) ...*) expression syntax

Creates an instance of the *class*. The initial values of the slots associated with the *instance-variables* are the values of the *expressions*. The initial values of other slots are the default initial values of effective instance variables of the class or unspecified.

The *class* and the *instance-variables* are treated as symbols. The *class* must be the name of an existing class. An *instance-variable* must be the name of one of the effective instance variables of the *class*.

(**slot-ref** *instance instance-variable*) expression syntax

Returns the value of the *instance-variable* in the *instance*. *Instance-variable* is a symbol (it is not evaluated) and must be the name of an instance variable. The instance variable must be defined in the module where the **slot-ref** form appears.

(**slot-set!** *instance instance-variable expression*) expression syntax

Changes the value of the *instance-variable* in the *instance* to the value of the *expression*. *Instance-variable* is a symbol and must be the name of an instance variable. The instance variable must be defined in the module where the **slot-ref** form appears. The value returned by the **slot-set!** form is unspecified.

(**type?** *instance class*) expression syntax

Returns `#t` if the class of the *instance* is a sub-class of the *class*, otherwise returns `#f`. The *class* is a symbol which names a class.

(**type-of** *instance*) essential procedure

Returns the class of the *instance*.

B.1.3 Expression Semantics

The only difference between the semantics of IMOOP and Scheme is the way in which locations are associated with variables.

Expressions in a module's **public**, **private**, **method**, **class**, and **with** statements are the module's *top-level* expressions. Variables appearing in a top-level expression are either free or bound. A bound variable reference or assignment in a top-level expression refers to a fixed location of a binding established within the top-level expression. The value of a free variable reference in a top-level expression of a module is the value of the location of a private variable, a public variable, or an imported variable of the module. A free variable assignment in a top-level expression of a module assigns a value to the location of a private variable of the module.

A free variable access may not always use a fixed location. It uses the location to which the free variable is currently bound. During the evaluation of an IMOOP program, a free variable access in a top-level expression of a module uses the location of one of the module's private, public, method, or imported variables that has the same name as the free variable when the access is made. The imported variables of a module is determined dynamically for every free variable access using the import specification of the module and the public environments of the imported modules. Is is an error to reference or assign an unbound variable.

Name conflicts among private, public, method, or imported variables are resolved using the following precedence rules:

- Locally defined private, public, or method variables has precedence over imported variables.
- The precedence of imported variables exported from different modules depends on the position of these modules appearing in the module's import specification. If two or more modules exports a public variable with the same name, the public variable of the leftmost module in the import specification shadows the public variables of other imported modules.
- Variables exported from the *scheme* module have the lowest precedence.

B.2 The Programming Environment

The IMOOP programming environment provides tools to manage the development of IMOOP projects. These tools consists of a read-eval-print loop, its interface with the GNU Emacs editor, and commands for managing projects, modules, and bindings.

B.2.1 The Read-Eval-Print Loop

The read-eval-print loop allows the user to enter IMOOP statements and commands interactively. A statement entered in the read-eval-print loop does not contain the module name that indicates the module in which the statement should be evaluated. The statement is evaluated in the "current" module. The current module can be changed by the user using the `set-current-module` command. The prompt of the read-eval-print loop displays the name of the current module. Assuming that the name of the current module is *module*, text entered in the read-eval-print loop is transformed using the following rules before being evaluated ($\textit{statement}^{\Rightarrow}$ indicates the result of recursively applying the following rule to *statement*):

$$\begin{aligned}
\langle\text{IMOOP commands}\rangle &\Longrightarrow \langle\text{IMOOP commands}\rangle \\
(\textbf{import}\ \textit{Imports}) &\Longrightarrow (\textbf{import}\ \textit{module Imports}) \\
(\textbf{public}\ \textit{variable expression}) &\Longrightarrow (\textbf{public}\ \textit{module variable expression}) \\
(\textbf{private}\ \textit{variable expression}) &\Longrightarrow (\textbf{private}\ \textit{module variable expression}) \\
(\textbf{class}\ \textit{Inherits Instance-Vars}) &\Longrightarrow (\textbf{class}\ \textit{module Inherits Instance-Vars}) \\
(\textbf{method}\ \textit{variable expression}) &\Longrightarrow (\textbf{method}\ \textit{module variable expression}) \\
(\textbf{begin}\ \textit{statement}\ \ldots) &\Longrightarrow (\textbf{begin}\ \langle\textit{statement}^{\Rightarrow}\rangle\ \ldots) \\
\langle\text{expression}\rangle &\Longrightarrow (\textbf{with}\ \textit{module}\ \langle\text{expression}\rangle)
\end{aligned}$$

B.2.2 The GNU Emacs Interface

The GNU Emacs interface allows modules to be created, edited, and saved in files. It also allows statements in a module to be "sent" and evaluated in the read-eval-print loop directly. This interface can also be used with Epoch. Epoch is a version of GNU Emacs that supports true multi-window editing under the X window system.

Files associated with IMOOP modules must be named using their names extended with the *.ms* extension. Statements in a file need not specify the name of the module that the statement belong. The Emacs interface can obtain the module name from the file name and inform the read-eval-print loop to change its current module before sending the statement to evaluate. The programming environment adopts the convention that only statements excluding the **with** statement can appear in a file and can be sent to the read-eval-print loop.

B.2.3 IMOOP Commands

IMOOP provides commands for managing projects, modules, and bindings. These commands are recognized only by the read-eval-print loop and cannot be used in program statements.

Project Management

(load-project *name*)

Removes the current project and loads the project stored in the file *name.ps* from the current working directory. The name of a project file have the *.ps* extension and contains a list of *module descriptions* that specifies the load order of modules in the project. Each module description specifies the name of a module and the directory from which to load the module. An error is signaled if *name.ps* cannot be found in the current directory. A project named *no-name* with no modules is initially loaded into the system.

(save-project)

Saves the current project, say *project*, in the file *project.ps* in the current working directory.

(save-project-as *name*)

Saves the current project in *name.ps* in the current working directory.

(compile-project)

Compile the current project. The object code is put in *name.ss* in the current working directory. *Name* is the name of the current project.

Module Management

(set-current-module *module*)

Set the current module of the project to the *module*. When a module is set to be the current module for the first time, it is automatically added to the current project.

(load-module *module string*)

Module is set to be the current module and is loaded from *module.so*, if it exists; otherwise *module* is loaded from *module.ms*. The **import** and **class** statements should appear at the beginning of *module.ms* and are loaded first. Other statements are loaded in an unspecified order. The load order of modules is kept in the *load order list*. By default, the *module* is appended at the end of the load order list. When a module is set as the current module for the first time, the module is appended at the end of the load order list.

String is an optional argument. It specifies a directory path name. If *string* is provided, the module is loaded from the directory, otherwise the module is loaded from the current working directory. An error is signaled if the file is not found.

(change-order *module after*)

This command removes *module* from the load order list and reinserts it right after module *after* in the load order list. If *after* is not present, then the module is inserted at the end of the load order list. An error is signaled if *module* or *after* cannot be found in the load order list.

(remove-module *module*)

Removes the *module* from the current project. Module *scheme* becomes the current module. An error is signaled if the *module* is not in the current project or a class is defined in the module.

(remove-class *module*)

Removes the class definition from the *module*. The class must not be a super class of any other class.

(compile-one-module *module*)

Compiles *module* and stores the output in the file *module.so*. This file is put in the directory where *module* is loaded. An error is signaled if *module* is not in the current project. Components of compiled modules cannot be changed interactively in the read-eval-print loop. Every imported module of a compiled module must also be a compiled module.

(compile-modules *module*)

Recursively compiles the *module* and all the directly or indirectly imported modules of the *module*. An error is signaled if any one of the modules is not in the project.

Binding Management

(delete-binding *module variable*)

Deletes the binding of the *variable* from the *module*. An error is signaled if the *module* is not a developing module or if the *variable* is not a public, private, or method variable in the *module*.

(refresh-binding)

Reinitializes the values of private variables in every module to the values when the variables were last defined. The modules are initialized according to the order specified by the load order list.

Bibliography

[1] Abelson, H., Sussman, G. J., and Sussman, J. *Structure and Interpretation of Computer Programs.* MIT Press, 1984.

[2] Adams, N., and Rees, J. Object-oriented programming in Scheme. In *1988 ACM Conference on Lisp and Functional Programming* (1988), pp. 277–288.

[3] America, P. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of OOPSLA EOOOP '90, Object-Oriented Programming Systems, Languages, and Applications* (October 1990), pp. 161–168. printed as SIGPLAN Notices, 25(10).

[4] Bawden, A., and Rees, J. Syntactic closures. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming* (Salt Lake City, Utah., July 1988).

[5] Bobrow, D., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. CommonLoops: Merging Lisp and object-oriented programming. In *Proceedings of OOPSLA'86, Object-Oriented Programming Systems, Languages, and Applications* (November 1986), pp. 17–29. printed as SIGPLAN Notices, 21(11).

[6] Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. Common Lisp Object System Specification. *SIGPLAN NOTICES 23*, 9 (September 1988), 1–48.

[7] Borning, A. H., and Ingalls, D. H. H. Multiple inheritance in Smalltalk-80. In *Proceedings of 1982 AAAI National Conference on Artificial Intelligence* (1985), pp. 234–238.

[8] Brooks Jr, F. P. *The Mythical Man-Month.* Addison-Wesley, 1982.

[9] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. Modula-3 report. Tech. Rep. 31, DEC Systems Research Center, 1988.

[10] Clinger, W., and Rees, J. Macros that work. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages* (January 1991), pp. 155–162.

[11] Connor, R. C. H., Dearle, A., Morrison, R., and Brown, A. L. An object addressing mechanism for statically typed languages with multiple inheritance. In *Proceedings of OOPSLA '89, Object-Oriented Programming Systems, Languages, and Applications* (October 1989), pp. 279–286. printed as SIGPLAN Notices, 24(10).

[12] Cox, B. *Object-Oriented Programming, An Evolutionary Approach.* Addison Wesley, 1987.

[13] Cox, B. J. There is a silver bullet. *Byte* (October 1990), 209–218.

[14] Curtis, P., and Rauen, J. A module system for Scheme. In *Conference Record of the 1990 ACM Lisp and Functional Programming* (1990).

[15] Dahl, O., Myhrhaug, B., and Nygaard, K. *Simula67 Common Base Language*, second ed. Norwegian Computing Center, 1970.

[16] Deutsch, L. P., and Schiffman, A. M. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages* (1983), pp. 297–302.

[17] Dixon, R., Mckee, T., Vaughan, M., and Schweizer, P. A fast method dispatcher for compiled languages with multiple inheritance. In *Proceedings of OOPSLA '89, Object-Oriented Programming Systems, Languages, and Applications* (October 1989), pp. 211–214. printed as SIGPLAN Notices, 24(10).

[18] Drescher, G. L. Object Scheme: Object inheritance as fluid binding. Thinking Machines Corporation, 1990.

[19] Dussud, P. H. TICLOS: An implementation of CLOS for the explorer family. In *Proceedings of OOPSLA '89, Object-Oriented Programming Systems, Languages, and Applications* (October 1989), pp. 215–220. printed as SIGPLAN Notices, 24(10).

[20] DYBVIG, R. K. *The Scheme Programming Language.* Prentice-Hall, 1987.

[21] DYBVIG, R. K. *Three Implementation Models for Scheme.* PhD thesis, University of North Carolina, Chapel Hill, 1987.

[22] DYBVIG, R. K., AND HIEB, R. Engines from continuations. *Journal of Computer Languages 14*, 2 (1989), 109–123.

[23] FRIEDMAN, D. P., AND FELLEISEN, M. A closer look at export and import statements. *Computer Language 11*, 1 (1986), 29–37.

[24] FRIEDMAN, D. P., WAND, M., AND HAYNES, C. T. *Essentials of Programming Languages.* MIT Press and McGraw-Hill, 1991.

[25] GOLDBERG, A. *Smalltalk-80 The Interactive Programming Environment.* Addison-Wesley, 1983.

[26] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80 The Language and Its Implementation.* Addison-Wesley, 1983.

[27] HANSON, C. A syntactic closures mocro facility. unpublished manuscript.

[28] HANSON, D. R. Is block structure necessary? *Software Practice and Experience 11* (1981), 853–866.

[29] HARPER, R., MILNER, R., AND TOFTE, M. The definition of Standard ML. Tech. Rep. ECS-LFCS-89-81, Department of Computer Science, University of Edinburgh, 1989.

[30] HAYNES, C. T. Logic continuations. In *Proceedings of the Third International Conference on Logic Programming* (July 1986), Springer-Verlag, pp. 671–685.

[31] HAYNES, C. T., AND FRIEDMAN, D. P. Abstracting timed preemption with engines. *Journal of Computer Languages 12*, 2 (1987), 109–121.

[32] HAYNES, C. T., AND FRIEDMAN, D. P. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems 9*, 4 (Oct. 1987), 582–598.

[33] HIEB, R., AND DYBVIG, R. K. Syntactic abstraction in Scheme. unpublished manuscript.

[34] HUDAK, P. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys 21*, 3 (September 1989), 359–411.

[35] IEEE. *IEEE Standard for the Scheme Programming Language.* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991. IEEE Std 1178-1990.

[36] JAGANNATHAN, S. *A Programming Language Supporting First-Class Parallel Environments.* PhD thesis, Massachusetts Institute of Technology, January 1989.

[37] KAMIN, S. Inheritance in Smalltalk-80: a denatational definition. In *ACM Symposium on Principles of Programming Languages* (1988), pp. 80–87.

[38] KAPLAN, S., CARROLL, A. M., LOVE, C., AND LALIBERTE, D. M. *Epoch - GNU Emacs for the X Window System.* Department of Computer Science, University of Illinois at Urbana-Champaign, 1990.

[39] KEENE, S. E. *Object-Oriented Programming in Common Lisp.* Addison Wesley, 1989.

[40] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language.* Prentice Hall, Englewood, New Jersey, 1988. The second edition.

[41] KICZALES, G., AND RODRIGUEZ, L. Efficient method dispatch in PCL. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming* (1990), pp. 99–105.

[42] KOHLBECKER, E. E., FRIEDMAN, D. P., FELLEISEN, M., AND DUBA, B. Hygienic macro expansion. In *Proceedings of the 1986 ACM Symposium on LISP and Functional Programming* (Aug. 1986), pp. 151–161.

[43] KRANZ, D., KELSEY, R., REES, J., HUDAK, P., PHILBIN, J., AND ADAMS, N. Orbit: An optimizing compiler for Scheme. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction* (1986), 219–233. published as *SIGPLAN Notices* 21, 7 (July 1986).

[44] KRASNER, G., Ed. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, Reading, MA., 1983.

[45] KROGDAHL, S. Multiple inheritance in Simula-like languages. *BIT 25* (1985), 318–326.

[46] Lamping, J. O. A unified system of parameterization for programming languages. In *1988 ACM Conference On Lisp and Functional Programming* (July 1988), pp. 316–326.

[47] Lang, K. J., and Pearlmutter, B. A. Oaklisp: An object-oriented Scheme with first class types. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages, and Applications* (November 1986), pp. 30–37. printed as SIGPLAN Notices, 21(11).

[48] Lieberman, H. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages, and Applications* (November 1986), pp. 214–223. printed as SIGPLAN Notices, 21(11).

[49] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction mechanisms in CLU. *Communications of the ACM 20*, 8 (1977), 564–576.

[50] MacQueen, D. Modules for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (1984), pp. 198–207.

[51] Meyer, B. Eiffel: A language and environment for software engineering. *The Journal of Systems and Software* (1988).

[52] Meyer, B. Eiffel: The language. Tech. rep., Interactive Software Engineering Inc., 1989.

[53] Moon, D. A. Object-oriented programming with Flavors. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages, and Applications* (November 1986), pp. 1–8. printed as SIGPLAN Notices, 21(11).

[54] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* (December 1972), 1053–1058.

[55] Pugh, W., and Weddell, G. Two-directional record layout for multiple inheritance. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* (June 1990), pp. 85–91. printed as SIGPLAN Notices, 25(6).

[56] Queinnec, C., and Padget, J. A detailed summary of a deterministic model of modules and macros for Lisp. Tech. Rep. LIX/RR/90/01, Ecole Polytechnique,

Laboratoire d'Informatique, 91128 Palaiseau Cedex (France), July-December 1989.

[57] Reddy, U. Objects as closures: Abstract semantics of object-oriented languages. In *1988 ACM Conference on Lisp and Functional Programming* (July 1988), pp. 289–297.

[58] Rees, J., and Clinger, W. (Editors), Revised^4 report on the algorithmic language Scheme. *Lisp Pointers 4*, 3 (1991), 1–55.

[59] Rodríguez, R. G., Duba, B. F., and Felleisen, M. Can you trust your read-eval-print loop? unpublished manuscript.

[60] Rose, J. R. Fast dispatch mechanisms for stock hardware. In *Proceedings of OOPSLA '88, Object-Oriented Programming Systems, Languages, and Applications* (November 1988), pp. 27–35. printed as SIGPLAN Notices, 23(11).

[61] Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. An introduction to Trellis/Owl. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages, and Applications* (November 1986), pp. 9–16. printed as SIGPLAN Notices, 21(11).

[62] Schmidt, D. A. *Denotational Semantics.* Allyn and Bacon, Inc., Boston, MA, 1986.

[63] Snyder, A. CommonObjects: An overview. *SIGPLAN Notices 21*, 10 (October 1986), 19–28.

[64] Snyder, A. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages, and Applications* (November 1986), pp. 38–45. printed as SIGPLAN Notices, 21(11).

[65] Springer, G., and Friedman, D. P. *Scheme and the Art of Programming.* MIT Press and McGraw-Hill, 1989.

[66] Steele Jr, G. L. *Common Lisp.* Digital Press, 1990. Second Edition.

[67] Stein, L. A. Delegation is inheritance. In *Proceedings of OOPSLA '87, Object-Oriented Programming Systems, Languages, and Applications* (December 1987), pp. 138–146. printed as SIGPLAN Notices, 22(12).

[68] Stein, L. A., Liberman, H., and Ungar, D. A shared view of sharing: The Treaty of Orlando. In *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. H. Lochovsky, Eds. ACM Press, 1989, ch. 3, pp. 31–48.

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

[70] Stroustrup, B. *The C++ Programming Language.* Addison-Wesley, 1986.

[71] Stroustrup, B. An overview of C++. *SIGPLAN Notices 21*, 10 (October 1986), 7–18.

[72] Sussman, G. J., and Steel Jr, G. L. Scheme: an interpreter for extended lambda calculus. Tech. rep., Massachusetts Institute of Technology Artificial Intelligence Memo 349, 1975.

[73] Swinehart, D., Zellweger, P., Beach, R., and Hagmann, R. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems 8*, 4 (October 1986), 419–490.

[74] Teitelman, W. A tour through Cedar. *IEEE Software* (April 1984), 44–73.

[75] Teitelman, W., and Masinter, L. The Interlisp programming environment. *IEEE Computer 14*, 4 (1981), 25–34.

[76] Tennent, R. D. The denotational semantics of programming languages. *Communications of the ACM 19*, 8 (August 1976), 437–453.

[77] Tennent, R. D. Two examples of block structure. *Software Practice and Experience 12* (1982), 385–392.

[78] Ungar, D., and Smith, R. B. Self: The power of simpicity. In *Proceedings of OOPSLA '87, Object-Oriented Programming Systems, Languages, and Applications* (December 1987), pp. 227–242. printed as SIGPLAN Notices, 22(12).

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

[80] Walker, J. H., Moon, D. A., Weinreb, D. L., and McMahon, M. The Symbolics Genera programming environment. *IEEE Software 4*, 6 (November 1987), 36–45.

[81] Wirth, N. *Programming in Modula-2.* Springer Verlag, 1983.

[82] Wulf, W., and Shaw, M. Global variable considered harmful. *ACM SIGPLAN Notices* (Februry 1973), 28–34.