The Scheme of Things:

Portability

William Clinger
Semantic Microsystems, Inc.

A reader has asked for suggestions on how to write Common Lisp programs so that they can in the future be translated into Scheme without too much difficulty. Someone else wants to write programs in Scheme that might eventually be translated into Common Lisp. These are interesting questions, but a related question is more immediate: How do you write programs in Scheme that can easily be translated to run in other implementations of Scheme?

In practice, three characteristics of Scheme seem to encourage non-portable programs:

1. Scheme is a very small language. Implementations often add features to compensate for real or imagined deficiencies.

2. The de facto Scheme standard effectively defines both "full" Scheme and an "essential" subset. Most implementations fall somewhere in between.

3. The Scheme standard deliberately leaves many things unspecified.

Programmers can overcome problems of the first kind simply by learning the difference between implementation-dependent extensions and the standard language. There is nothing particularly wrong with using implementation-dependent extensions in portable code, provided the extensions can be duplicated easily in other implementations. For example, use of the SCOOPS object system causes few portability problems since its source code is available and has been ported to most Scheme systems already. Use of engines or multi-tasking, on the other hand, may cause serious portability problems because they cannot be implemented in standard Scheme.

The differences between full and essential Scheme cause few problems because most implementations are reasonably close to full Scheme. The few problems that arise are obvious and can usually be solved by adding missing procedures or altering a program's syntax to avoid the missing feature.

The most vexing portability problems have to do with deliberate under-specification in the definition of Scheme. For example, Scheme does not specify the order of evaluation of expressions in a procedure call, so a program that inadvertently depends on right-to-left evaluation order may break mysteriously when ported to an implementation that evaluates from left to right. This particular bug does not crop up very often, perhaps because most implementors are compiler writers who would like to exploit the flexible evaluation order to obtain more efficient code. If they're not exploiting it now, they'd like to in the future. Hence their manuals emphasize that the order of evaluation cannot be relied upon, making Scheme programmers aware of it.

Whenever the language definition leaves some behavior unspecified, it creates an opportunity for implementations to invent a new feature by specifying that behavior. For example, the value returned by an assignment is unspecified in Scheme. In MIT Scheme, however, an assignment always returns the old value of the variable being assigned. In many other implementations, an assignment always returns the new value. Some programmers view these implementation-dependent behaviors as useful features and rely on them, thereby writing non-portable code. Implementations can help programmers to write portable code by returning a useless value such

as `#!unspecified` as the result of an assignment.

Implementations that try to prevent programmers from depending on unspecified behavior are said to be *strict*. Implementations that permit or even encourage programmers to rely on unspecified behavior are said to be *lenient*. It is usually easiest to port code from a strict implementation and to a lenient implementation.

PC Scheme, for example, is one of the most lenient implementations of Scheme. One reason is that Texas Instruments, as a vendor of Common Lisp, is trying to make it easy for Common Lisp programs to be translated into Scheme. PC Scheme therefore guarantees Common Lisp-like behavior in many places where the Scheme behavior is unspecified. Thus the value returned by an assignment in PC Scheme is the new value. As another example, PC Scheme extends the domains of `car` and `cdr` to include the empty list, taking advantage of the fact that Scheme does not require errors like `(car '())` to be signalled. It is perfectly reasonable for programmers to take advantage of these implementation-dependent behaviors when translating Common Lisp code to run in PC Scheme, but programmers get into trouble when they begin to rely on them in new Scheme code that should be portable.

Other implementations have less reason to be concerned about compatibility with other languages and may choose to be strict in order to help their users write more portable code. Thus some of Scheme's under-specification is good because it gives implementations freedom to fulfill their users' specific needs.

On the other hand, some of the things that Scheme leaves unspecified can be excused—if at all—only by appeal to history. For example, the de facto Scheme standard contains no English statement to the effect that the fundamental data types are disjoint. The formal semantics implies disjointness, but several parts of the English text contradict it. For example, the section on characters says:

> There is no requirement that the data type of characters be disjoint from other data types; implementations are encouraged to have a separate character data type, but may choose to represent characters as integers, strings, or some other type.

The presence of such explicit statements suggests that disjointness is to be assumed in the absence of such a statement, but as a matter of fact the authors of the *Revised[3] Report on the Algorithmic Language Scheme* actually debated an explicit requirement for disjointness and decided against it. The section on equivalence predicates fails to help either, since it gives conditions under which the predicates must return true but fails to give many conditions under which they must return false. As a result, it appears that a Scheme system in which `(eq? '(a) 34)` is true could satisfy the English text of the Scheme report. This is ridiculous.

Fortunately, most implementors are reasonable people. All implementations of Scheme supply disjoint types except for loopholes explicitly sanctioned by the report. Even so, these loopholes have caused more portability problems than any other misfeatures of Scheme, including the absence of a standard macro facility. Why then do the loopholes remain? They were intended as "grandfather" clauses that give implementations more time to convert to disjoint types.

In the long run, the fact that most implementations of Scheme continue to represent both `#f` and the empty list by the same object will create portability problems. The Scheme report makes clear that `#f` and the empty list ought to be two distinct objects, and the only suggestion that they might be the same is a parenthetical remark in the description of the `null?` procedure:

> (In implementations in which the empty list is the same as `#f`, `null?` will return `#t` if *obj* is `#f`.)

Because so many implementations take advantage of this obscure loophole, it is all too easy for programmers to lapse into thinking that `#f` is the same as the empty list. That is not so. Code that assumes that `#f` and the empty list are interchangeable is not portable.

For example, the value returned by `assoc` is either a pair or `#f`; `assoc` never returns the empty list. Thus it is correct to write

```
(define (lookup x)
  (let ((entry (assoc x *table*)))
    (if entry
        (cdr entry)
        #f)))
```

but incorrect to write

```
(define (lookup x)
  (let ((entry (assoc x *table*)))
    (if (null? entry)
        #f
        (cdr entry))))
```

because `(null? entry)` will never be true except in an implementation that represents `#f` as the empty list.

Although the empty list currently counts as false in boolean contexts, it is very poor style for a program to rely upon this. That the empty list counts as false in Scheme is nothing more than a concession to implementations that confuse `#f` with the empty list. As these implementations become obsolete, it will become possible to change the language so that the empty list counts as true. Some implementations have already made this change internally, and are using a compatibility package to make the empty list appear false in user code.

Returning to the question of Common Lisp: Not only does Common Lisp fail to distinguish `#f` from the empty list, it identifies both with the symbol `nil`. Furthermore Common Lisp has no separate notation analogous to Scheme's `#f` for the false boolean value. When writing Common Lisp code that may eventually be translated into Scheme, it is very important to follow the stylistic guidelines laid down by *Common Lisp: the Language*:

> ...as a matter of style, this manual uses the notation `()` when it is desirable to emphasize the use of an empty list, and uses the notation `nil` when it is desirable to emphasize the use of the Boolean "false." The notation `'nil` (note the explicit quotation mark) is used to emphasize the use of a symbol...

Followed conscientiously, these guidelines set up a one-to-one textual correspondence between the three uses of Common Lisp's single object `nil` and Scheme's three distinct objects:

Common Lisp	Scheme
`()`	`'()`
`nil`	`#f`
`'nil`	`'nil`

The textual translation is made even easier if the empty list is written as `'()` in Common Lisp instead of `()`, since the latter often indicates syntax (such as an argument list) instead of a true object. Unfortunately these stylistic guidelines do not help when translating data from Common Lisp to Scheme or other languages.