MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1998

Special Note – November 13, 1998

A Dynamic Scoping Extension to Scheme

In Problem Set 8, an implementation for integrating dynamic scoping together with lexical scoping is considered. The purpose of this note is to further discuss one possible set of syntactic and semantic extensions to add dynamic scoping to Scheme.

First, we will consider the "simple" cases, and then we examine some of the trickier issues for how we might want dynamic scoping to work.

We extend the syntax of the language to include a new construct. The special form `(dynamic <var> <val>)` creates a new dynamic variable `var` in the current frame. This both creates a binding for the variable (to the value `val`), and declares that `var` is accessible via dynamic scoping to procedures *called within* the current frame (and accessible to procedures called through any chain of procedure calls initiated within that frame). An important point is that the variable is not *automatically* accessible in those called procedures; rather, if one of those procedures wishes to gain access back up to the dynamic variable, this desire must be explicitly indicated as well. Within such a called procedure, the special form `(dynamic <var>)` indicates that within the current frame the variable `<var>` should be looked up in the enclosing dynamic environment rather than through the enclosing lexical environment.

The simplest example is shown below. In this case `test1` (a procedure defined in the global environment) creates a dynamic variable `*z*`, and `test2` (also defined in the global environment) is able to access that variable through the dynamic mechanism:

```
(define (test1)
  (dynamic *z* 100)
  (test2))

(define (test2)
  (dynamic *z*)
  *z*)

(test1) ==> 100
```

In the next example, we see that the dynamic mechanism can bridge across more than one procedure call to gain access to the dynamic variable.

```
(define (test1)
  (dynamic *z* 100)
  (test2))
```

```
(define (test2)
  (test3))

(define (test3)
  (test4))

(define (test4)
  (dynamic *z*)
  *z*)

(test1) ==> 100
```

The basic dynamic scoping mechanism above is relatively straightforward and easy to understand. The more difficult situations are when the lexical and dynamic scoping mechanisms overlap or conflict. Then we need to be very careful to make clear the semantics or the behavior we desire.

Consider first the case where we have a long chain of calls, as in the `test1`, `test2`, `test3`, `test4` example above. Somewhere in the middle of this chain of calls (e.g. in `test3`), the writer of this procedure may (inadvertantly or intentionally) also define a lexical variable with the same name as our dynamic variable:

```
(define (test3)
  (define *z* 42)
  (display "The answer is " *z*)
  (test4))

(test1) ==> 100
```

Inside `test3` we did *not* declare `*z*` to be dynamic, so it is reasonable to wish that this procedure act in a well-defined "isolated" fashion with respect to its internally scoped `*z*`, so that the presence of this local variable does not change the behavior of the dynamic variable relationship between `test1` and `test4`. Another way to say this is that a dynamic variable declaration (e.g. `(dynamic *z*)` does *not* affect bindings pervasively: inner bindings of a variable shadow a `dynamic` declaration, and the `dynamic` variable must be explicitly redeclared to be dynamic if within the shadowed scope.

A reasonable semantics is thus to consider that the creation of the orginal dynamic variable `*z*` is in some sense an "export" of the variable to only those dynamic frames that explictly "import" the dynamic variable through a `(dynamic *z*)` expression at the start of that frame. Thus, we might want our implementation of dynamic scoping to trace back through the chain of dynamic frames but only "find" a variable that matches the name in some frame if it has been explicitly declared to be an exported dynamic variable.

To reduce the risk of confusion between such lexical and dynamic variables, it is a common convention to put asterisks at the start and end of dynamic variable names.

The semantics described above have an interesting effect. Consider

```
(define (proc1)
  (dynamic *x* 10)
  (proc2))

(define (proc2)
  (dynamic *x*)
  (let ((a 1))
    *x*))

(proc1) ==>
;; Error: *x* is an unbound dynamic variable
```

The `let` binding creates a new local frame in which `*x*` has not been explicitly redeclared to gain dynamic access. To correctly gain access to the dynamic variable within the `let`, one should do:

```
(define (proc1)
  (dynamic *x* 10)
  (proc2))

(define (proc2)
  (let ((a 1))
    (dynamic *x*)
    *x*))

(proc1) ==> 10
```

The next case we consider is somewhat ambiguous. Consider the following:

```
(define (outer)
  (dynamic *b* 10)
  (let ((a 1))
    (+ a *b*)))

(outer)
==> 11
```

The key question is: should an exported dynamic variables ALSO be available lexically? This is a matter for debate. On the one hand, one might argue that dynamic variables should only be accessible through an explicit *import*. On the other hand, one might want *exported* (newly created) dynamic variables to be accessible via both lexical scoping and dynamic scoping. Both of these arguments have merit, so we arbitrarily pick how we want our system to work: we enable dynamic definitions to also be lexically scoped. Thus in the example above `(outer)` returns 11, because `*b*` is still visible in the inner lexical scope.

The last case highlights one other ambiguity. If one explictly imports a dynamic variable but no such variable exists in the dynamic environment, then the variable is unbound and an error ensues. This is true even if there is an available variable with the same name somewhere in the surrounding lexical frame. This is best seen with an example, which is also illustrated in Fig. 1:

```
(define (make-thing x)
  (dynamic *w* 10)
  (lambda (v)
    (dynamic *w*)
    *w*))

(define thing (make-thing 1))

(thing 1) ==>
;; Error: dynamic variable *w* is unbound
```

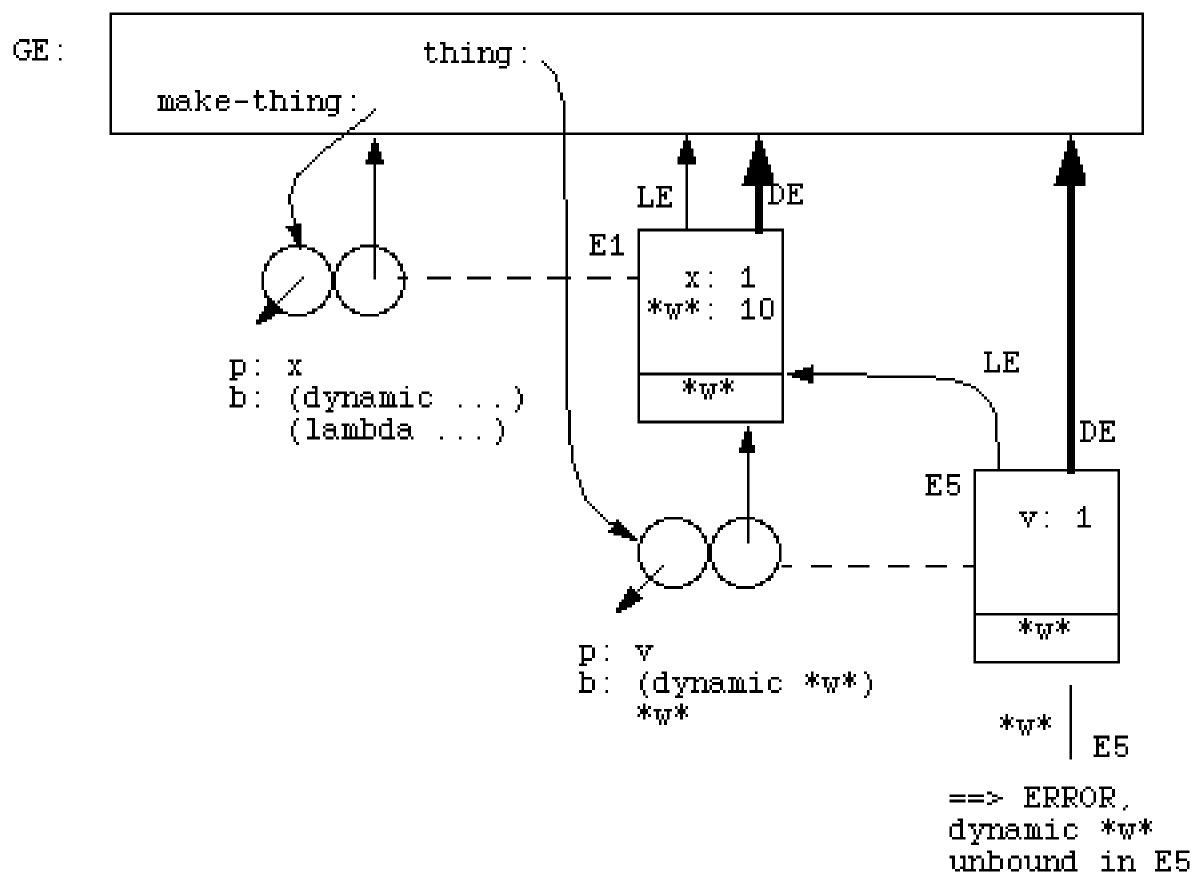

Figure 1: An extended environment diagram where both lexical and dynamic scoping coexist.

An "extended" environment diagram as shown in Fig. 1 can be used to illustrate this case. Here we show arrows from a frame both for the enclosing lexical environment (LE) and for the enclosing dynamic environment (DE). At the bottom of the frame, we list the variables that have been declared to be dynamic within that frame (by evaluation of either `(dynamic <var> <val>)` or `(dynamic <var>)`).

In this case, the frame `E5` resulting from the application of `thing` has the global environment as it's surrounding dynamic environment. When we look up the dynamic variable `*w*` in this frame and

it's surrounding dynamic environment, the variable's definition in `E1` is not visible and the variable is unbound in the dynamic environment.

If one actually wanted to get access to the `*w*` variable from within the `thing` lambda expression, one should just use lexically scoping:

```
(define (make-thing2 x)
  (dynamic *w* 10)
  (lambda (v)
    *w*))

(define thing2 (make-thing2 1))

(thing2 1) ==> 10
```

We have discussed one set of syntactic and semantic extensions to Scheme that implement dynamic scoping. As you can see, such extensions (especially those that deal so fundamentally with the language) need to be thought through very carefully. There is also room for debate about what the best semantics are – welcome to the world of language design!