CS2135: Let, Letrec, and Scoping

Kathi Fisler, WPI

February 14, 2003

1 Let and Letrec : An Introduction to Scope

The monitor example introduced you to an important languages concept that we touched on very briefly a couple of weeks ago when we introduced **lambda**: scoping. When you introduce a new identifier (parameter or variable) in a program, scoping determines where that identifier is visible. Alternatively phrased, it determines which expressions can "see" (or refer to the value of) that identifier.

Whenever you learn a new language construct that can introduce identifiers, you should ask what scope those identifiers have (in other words, where are they visible). So far, we know four constructs that introduce new identifiers: **define**, **lambda**, **let** and **letrec**. Let's precisely define the scoping rules for each of these:

- When you define a function (**lambda**), its parameters are visible in the body of the function, but not outside the function.
- When you introduce a variable through **let**, it is visible in the body of the let, but not in the expressions associated with other identifiers in the same **let**. So, for example, if you wrote

  ```
  (let ([x 4]
        [y (+ x 1)])
    (* x y))
  ```

  you'd get an undefined identifier error on *x* in (+ *x* 1) because *x* and *y* are only visible in (∗ *x y*).
- When you introduce a variable through **letrec**, it is visible in the body of the **letrec** AND in all of the expressions associated with identifiers. This visibility supports defining recursive functions (among other things), which you cannot do with **let**. So, for example

  ```
  (letrec ([x 4]
           [y (+ x 1)])
    (* x y))
  ```

  returns 20 (without generating a syntax error). The following example also runs without error

  ```
  (letrec ([g (lambda (y) (f y))]
           [f (lambda (x) (+ x 3))])
    (g 2))
  ```

  and returns 5. Note that in **letrec**, *g* can refer to *f* even though *f* appears to be defined after *g*.
- When you introduce a variable through **define**, as in (**define** *talk1* (*make-talk* . . .)), the variable is visible everywhere in the program (i.e., it's a global variable).

What About Naming Conflicts?

Scoping tells you where a name is visible, it also tells you how to resolve naming conflicts between identifiers. Consider the following code:

```
(define (num-func anum)
  (let ([anum 3])
```

```
    (+ anum 5)))

(num-func 1)
```

What does this code return? We should get either 6 or 8, depending upon which value of *anum* gets used in (+ *anum* 5). Scheme (indeed, all modern languages) uses a policy called *static scoping*: use the nearest enclosing definition in the text of the program. In this case, the *anum* from **let** is closer to the (+ *anum* 5) than the parameter *anum*, so Scheme uses the **let** definition (the value 3).

In general, we define nearest by working backwards through the nested expressions until we find a definition of the needed identifier. For example, the following code

```
(define (num-func2 anum)
  (* (let ([anum 3]) (+ anum 1))
     (let ([anum 4]) (− anum 2))
     anum))

(num-func2 5)
```

returns 40. Why? This expression is equivalent to

```
(define (num-func2 anum)
  (* 4
     2
     anum))

(num-func2 5)
```

In each **let**, the value of *anum* is only used to compute the expression in the body of the **let**; the rest of the expression can't see those local definitions of *anum*. For the third argument to the multiplication, the nearest enclosing definition of *anum* comes from the parameter. The crucial term here is **enclosing**: the **let** statements don't contain the use of *anum* in the multiplication within their parentheses. Staying within parentheses, the parameter is the nearest *anum* to that third argument.