Ch. 7 Special Forms

Recall that every element of the non-empty list is evaluated before any function is applied when using the default evaluation rule (DER) for function calls. The one exception to the DER is for language keywords called special forms. These keywords are part of the syntax of the Racket language, and they are called “special” because they appear to be used like a function call. However, parts of a non-empty list starting with a special form keyword may not be evaluated at all. In fact, each special form has its own rule for evaluation and each special form evaluation is different. A special form is a non-empty list that begins with a left parenthesis followed immediately by a character sequence that is a special form.

The define special form

This very important special form is used to put a new symbol/value pair into the Global Environment (GE). The syntax of the expression \( \text{define C1 C2} \) is a non-empty list in which define \( \rightarrow \) the keyword define, C1 \( \rightarrow \) n where n can be any valid Racket symbol, and C2 \( \rightarrow \) d is any valid Racket datum. The only part of \( \text{define C1 C2} \) that is evaluated is C2.

For example, for the non-empty list \( \text{define s e} \) would initiate the following steps:

1. Insert a new entry \( s / \text{void} \) into the GE.
2. Translate \( e \rightarrow E \).
3. Evaluate the datum denoted by \( E \) using DER to get \( E' \).
4. Insert \( E' \) into the position occupied by void in step 1. to yield \( s / E \).

The define special form has no output value. It is used only to insert symbols and their associated values into the GE, a \textit{very} important side effect.

Example 1: \( \text{define z (+ 24 3)} \) evaluates as follows:

1. \( z \rightarrow \) the symbol z
2. \textit{void} is associated with the name \( z \) in the GE
3. the non-empty list \( (+ 24 3) \) is evaluated by the DER \( \Rightarrow \) the number 27 \( \rightarrow 27 \)
4. result is the value 27 being associated with the symbol \( z \) in the GE

We can create a symbol for any type of Racket data type using define.
The **quote** special form

The **quote** special form is used to shield its argument from evaluation. It has no side effects. The shorthand notation for the **quote** special form is the apostrophe (’), also known to computer geeks as “single quote”. The **quote** special form is useful when treating symbols or non-empty lists as pieces of data, rather than using them as names of variables or function calls.

---

Ch. 9 **Defining Functions Using the lambda special form**

The **lambda** special form is introduced in chapter 9. Like any special form in Racket, the lambda special form is a list whose first element is a keyword symbol—in this case, the symbol **lambda**. The second element in a lambda special form is a list used to specify the names of the input parameter(s) for the function being defined. The rest of the elements in the lambda special form constitute the body of the function being defined. The lambda special form specifies everything about a function except its name.

A **lambda** expression has the following syntax:

```
(lambda (C_1 C_2 ... C_n) B)
```

where:

- each $C_i$ is a character sequence denoting some Racket symbol, $s_i$;
- the symbols, $s_1, s_2, \ldots, s_n$, are distinct (i.e., there are no duplicates); and
- $B$ is a character sequence denoting a Racket datum, D, of any kind.

Thus, $C_1 C_2 \ldots C_n$ specify $n$ distinct input parameters for the **lambda** expression, and $B$ specifies the body of the **lambda** expression.

Example 2: The mathematical expression $f(x) = x^2$ specifies

- a parameter name, $x$,
- a body, $x^2$, and
- a name, $f$, for the function being defined.

The **lambda** expression equivalent to the mathematical expression shown in Ex. 2 denotes a list whose first element happens to be the **lambda** symbol. The evaluation of a **lambda** special form always results in a function. If you type `(lambda (x) (* x x))` in the IW and press enter, it will evaluate to a nameless function shown as `#\langle procedure\rangle`. An expression like `(lambda (x) (* x x))` is called a “nameless lambda”.

A **lambda** expression invariably denotes a list—called a **lambda special form**—and the evaluation of that list invariably results in a Racket function. The semantics of the
lambda expression also includes a description of the behavior of that function should it ever be applied to any input(s). Assuming that

- each $C_i$ denotes a Racket symbol, $s_i$;
- the symbols, $s_1$, $s_2$, ..., $s_n$, are distinct; and
- $B$ denotes some Racket datum $D$,

then a lambda expression of the form $(\lambda (C_1 C_2 \ldots C_n) B)$ denotes a non-empty Racket list whose elements are as follows:

- the lambda symbol;
- a parameter list containing $n$ distinct symbols, $s_1$, $s_2$, ..., $s_n$; and
- the Racket datum, $D$.

We have seen a special form that puts symbol/value pairs in the GE. The define special form can be used along with the lambda special form to name the function so it can be used inside a program.

To name a function, we use the define and lambda special forms as follows:

\[(\text{define } \text{ sqr} \ (\lambda (x) (* x x)))\]

Recall that the define special form does not generate any output, so after the line above is executed, we have written the symbol sqr in the GE and the value $(\lambda (x) (* x x))$ is the value of $\text{sqr}$.

The evaluation of a lambda special form only creates the function; it does not apply it to any input(s). After the symbol sqr and its value, the lambda expression, are written in the GE, if you type the name sqr in the interactions window, you will see the name of the function along with the type it evaluates to: `#(procedure:sqr)`. Once you have given a name to a function, you can then call it like any of the built-in functions, as demonstrated below at the IW prompt:

\[
\begin{align*}
> & \ (\lambda (x) (* \ x \ x)) \ ; \ a \ nameless \ lambda \\
> & \ #<\text{procedure}> \\
> & \ (\text{define } \text{sqr} \ (\lambda (x) (* \ x \ x))) \ ; \ defines \ \text{sqr} \ function \\
> & \ \text{sqr} \\
> & \ #<\text{procedure:sqr}> \\
> & \ (\text{sqr} \ 5) \\
> & \ 25 \\
> & \ (\text{sqr} \ -9) \\
> & \ 81 \\
> & \ (\text{sqr} \ 4) \\
> & \ 16
\end{align*}
\]
> (sqr "hi")
*: contract violation
  expected: number?
given: "hi"
  argument position: 1st
  other arguments...:

Notice that the call to (sqr "hi") failed because the body of the sqr function attempts to apply the * function to a string. The functions we define are evaluated according to the DER for non-empty lists: the sqr symbol is evaluated by looking it up in the GE and, because it is a lambda special form, applying it to the desired input value. This function is called a “one-parameter” function because it only has one symbol in the parameter list—x.

If we could not name functions, we could still use them as nameless lambdas by including a nameless lambda expression inside a list that contains the lambda expression followed by input(s) to the function, as shown at the IW as shown below:

> ((lambda (x) (* x x)) 6) 
36
> ((lambda (x) (* x x)) 10) 
100

These expressions don’t violate the DER because the (lambda (x) ...) character sequence evaluates to a function. I hope you can see that programming could be very time consuming if we could use only unnamed lambda expressions!

When the lambda function is applied to some input values, as shown above for the value 6:

- A local environment is created that contains an entry for every symbol in the parameter list; in the case of the sqr function a single entry is added to the local environment in which the symbol x has the value 6.
- The expression (* x x), the body of the function, is evaluated with respect to the local environment, where it finds x evaluates to 6. The symbol * is evaluated with respect to the GE as the multiplication function if no symbol for * appears in the local environment.
- The value 36 is the output value that results from applying the sqr function to the value 6.

See Fig. 6.1 on page 28 and 9.1 on page 53 of our textbook for a picture of the application of a function to a value.
Notice that expressions in the body of a function can refer to data that are stored in one of two places:

1. the environment within which the function was created—in this case, the Global Environment; or
2. the local environment that contains entries associated with the input parameters.

The local environment is searched first, then the GE.

There can be zero-, one-, two-,...(a finite number of) parameters for each function. The definition of a zero-parameter function would look something like this:

\[
\text{(define whoKnows () "who knows?")}
\]

Note that the body of this function is not a list...it is the string “who knows?”. This is OK since the body of a function just needs to be a character sequence that evaluates to a valid Racket data type.

It is more common that the parameter list of a function is not empty, and that the body of a function is a non-empty list containing references to the parameter names that evaluates to some valid Racket data type. The body can also be composed of a number of expressions that are evaluated from top to bottom. However, only one value can be returned by a function, so it is wise to use only side-effect producing functions for all but the last expression in the function body.

Next Topic Testing Functions using the check-expect special form.