Intermezzo: BSL

Thus far we have approached teaching BSL as if it were a natural language. Like toddlers, we first taught you the vocabulary of the language with an intuitive understanding of its meaning. Next we exposed you to some basic rules of how to compose and not to compose sentences. Truly effective communication, however, requires some formal study of a language's vocabulary, grammar, and meaning.

A programming language is in many ways like a natural language. It has a vocabulary and a grammar. The vocabulary is the collection of those words from which we compose sentences in our language. A sentence in a programming language is an expression or a function definition; the language's grammar dictates how to form complete sentences from words. Programmers use the word *syntax* to refer to the basic vocabularies and grammars of programming languages.

Not all grammatical sentences are meaningful—neither in English nor in a programming language. For example, the English sentence "the cat is round" is a meaningful sentence, but "the brick is a car" makes no sense even though it is completely grammatical. To determine whether a sentence is meaningful, we must know the *meaning* of a language; programmers call this *semantics*.

> The intermezzos of this book introduce ideas from programming languages that programmers must eventually understand and appreciate but are secondary to program design.

In this intermezzo, we discuss the meaning of BSL programs as if it were an extension of the familiar laws of arithmetic and algebra. After all, computation starts with this form of simple mathematics, and we should understand the connection between this mathematics and computing. The first three sections present the vocabulary, grammar, and meaning of a good portion of BSL. Based on this new understanding of BSL, the fourth resumes our discussion of errors. The remaining three sections revisit `and` and `or` expressions, constant definitions, and structure types.

BSL: Vocabulary

Figure 28 introduces and defines BSL's basic vocabulary. It consists of names that may or may not have meaning according to BSL and of constants, also known as values and pieces of data.

An *`name`* or a *`variable`* is a sequence of characters not including a space or one of the following: `" , ' ` ( ) [ ] { } | ; #`:

- A *primitive* is a name to which BSL assigns meaning, for example, `+` or `sqrt`.
- A *variable* is a name without preassigned meaning.

A *value*, also a called a piece of data, is one of:

- A *number* is one of: `1`, `-1`, `3/5`, `1.22`, `1.22`, `0+1i`, and so on. The syntax for BSL numbers is complicated because it accommodates a range of formats: positive and negative numbers, fractions and decimal numbers, exact and inexact numbers, real and complex numbers, numbers in bases other than `10`, and more. Understanding the precise notation for numbers requires a thorough understanding of grammars and parsing, which is out of scope for this intermezzo.
- A *boolean* is one of: `#t` or `#f`.
- A *string* is one of: `""`, `"a"`, `"doll"`, `"he says \"hello world\" ok"`, and so on. In general, it is a sequence of characters enclosed by a pair of `"`. For example, are all strings.
- An *image* is a png, jpg, tiff, and various other formats. We intentionally omit a precise definition.

We use `v`, `v-1`, `v-2` and the like when we wish to say "any possible value."

Figure 28: BSL core vocabulary

Each of the explanations defines a set via a suggestive iteration of its elements. Although it is possible to specify these collections in their entirety and in a formal manner, we consider this superfluous here and trust in your intuition. Keep in mind that each of these sets may come with some extra elements.

BSL: Grammar

Figure 29 shows a large part of the BSL grammar, which—in comparison to other languages—is extremely simple. As to BSL's expressive power, don't let the looks deceive you. The first action item is to discuss how to read such grammars. Each line with a `=` introduces a *syntactic category*; the best way to pronounce `=` as "is one of" and `|` as "or." Wherever you see three dots, read as many repetitions of what precedes the dots as you wish. This means, for example, that a *program* is either nothing or a single occurrence of *def-or-expr* or a sequence of two of them:

> Reading a grammar aloud makes it sound like a data definition. One could indeed use grammars to write down many of our data definitions.

```
def-or-expr
def-or-expr
```

or three, four, five, or however many. Since this example is not particularly illuminating, let us look at the second syntactic category. It says that *definition* is either

```
(define (variable variable) expr)
```

because "as many as you wish" includes zero, or

```
(define (variable variable variable) expr)
```

which is one repetition, or

```
(define (variable variable variable variable) expr)
```

which uses two.

```
    program = def-or-expr ...

def-or-expr = definition
            | expr

 definition = (define (variable variable variable ...) expr)

       expr = variable
            | value
            | (primitive expr expr ...)
            | (variable expr expr ...)
            | (cond [expr expr] ... [expr expr])
            | (cond [expr expr] ... [else expr])
```

Figure 29: BSL core grammar

The final point about grammars concerns the three "words" that come in a distinct font: `define`, `cond`, and `else`. According to the definition of BSL vocabulary, these three words are names. What the vocabulary definition does not tell us is that these names have a pre-defined meaning. In BSL, these words serve as markers that differentiate some compound sentences from others, and in acknowledgment of their role, such words are called *keywords*.

Now we are ready to state the purpose of a grammar. The grammar of a programming language dictates how to form sentences from the vocabulary of the grammar. Some sentences are just elements of vocabulary. For example, according to figure 29 42 is a sentence of BSL:

- The first syntactic category says that a program is a *def-or-expr*. Expressions may refer to the definitions.
- The second definition tells us that a *def-or-expr* is either a *definition*

In DrRacket, a program really consists of two distinct parts: the definitions area and the expressions in the interactions area.

or an *expr*.

- The last definition lists all ways of forming an *expr*, and the second one is *value*.

Since we know from figure 28 that `42` is a value, we have a complete confirmation.

The interesting parts of the grammar show how to form compound sentences, which are sentences built from other sentences. For example, the *definition* part tells us that a function definition is formed by using "(", followed by the keyword `define`, followed by another "(", followed by a sequence of at least two variables, followed by ")", followed by an *expr*, and closed by a right parenthesis ")" that matches the very first one. You can see from this example how the leading keyword `define` distinguishes definitions from expressions.

Expressions, called *expr*, come in six flavors: variables, constants, primitive applications, (function) applications, and two varieties of `cond`itionals. While the first two are atomic sentences, the last four are compound sentences. Note how the keyword `cond` distinguishes conditional expressions from primitive and function applications.

Here are three examples of expressions: `"all"`, `x`, and `(f x)`. The first one belongs to the class of strings and is therefore an expression. The second is a variable, and every variable is an expression. The third is a function application, because `f` and `x` are variables.

In contrast, the following parenthesized sentences are not expressions: `(f define)`, `(cond x)`, and `((f 2) 10)`. The first one partially matches the shape of a function application but it uses `define` as if it were a variable. The second one fails to be a correct `cond` expression because it contains a variable as the second item and not a pair of expressions surrounded by parentheses. The last one is neither a conditional nor an application because the first part is an expression.

Finally, you may notice that the grammar does not mention white space: blank spaces, tabs, and newlines. BSL is a permissive language. As long as there is some white space between the elements of any sequence in a program, DrRacket can understand your BSL programs. Good programmers, however, may not like what you write. These programmers use white space to make their programs easily readable. Most importantly, they adopt a style that favors human readers over the software applications that process programs (such as DrRacket). They pick up this style from careful reading code examples in books, paying attention to how it is formatted.

> Keep in mind that two kinds of readers study your BSL programs: people and DrRacket.

Exercise 106. Explain why the following sentences are syntactically legal expressions

```
x
(= y z)
(= (= y z) 0)
```

▮

Exercise 107. Explain why the following sentences are syntactically illegal

```
(3 + 4)
number?(l)
(x)
```

▮

Exercise 108. Explain why the following sentences are syntactically legal definitions

```
(define (f x) x)
(define (f x) y)
(define (f x y) 3)
```

▮

Exercise 109. Explain why the following sentences are syntactically illegal

```
(define (f 'x) x)
(define (f x y z) (x))
(define (f) 10)
```

▮

Exercise 110. Pick the legal from the legal from illegal sentences in the following list:

1. `(x)`

2. `(+ 1 (not x))`

3. `(+ 1 2 3)`

Explain why the sentences are legal or illegal. Determine whether they belong the category *expr* or *definition*. ▮

Note on Grammatical Terminology: The components of compound sentences have names. We have introduced some of these names on an informal basis. Figure 30 provides a summary of the conventions.

```
; function application:
(function argument ... argument)

; function definition:
(define (function-name parameter ... parameter)
  function-body)

; conditional expression:
(cond
  cond-clause
  ...
```

```
  cond-clause)

; cond clause
[condition answer]
```

Figure 30: Syntactic naming conventions

In addition to the terminology of figure 30, we say *function header* for second component of a definition. Accordingly, the expression component is called *function body*. People who consider programming languages as a form of mathematics use *left-hand side* for a header and *right-hand side* for the body.

On occasion, you also hear or read the term *actual arguments* for the arguments in a function application. People tend to use this terminology when they think of the value of the arguments, which they also called their meaning.

BSL: Meaning

When DrRacket evaluates an expression, it uses nothing but the laws of arithmetic and algebra to convert an expression into a value—see figure 28. In ordinary mathematics courses, values are numbers. As you know from the figure, the values of BSL also include Booleans, String, and Images. Note how the collection of values is thus just a subset of the collection of expressions.

The rules of evaluation come in two categories. First, we need an infinite number of rules like those of arithmetic to evaluate applications of primitives:

```
(+ 1 1) == 2
(- 2 1) == 1
...
```

But BSL arithmetic is more general than just number crunching. It also includes rules for dealing with Boolean values, strings, and so on:

```
(not true) == false
(string=? "a" "a") == true
...
```

We use == to say that two expressions are equal according to BSL.

Like in algebra, you can always replace equals with equals in expressions:

```
(boolean? (= (string-length (string-append "hello" "world")) (+ 1 3)))
==
(boolean? (= (string-length (string-append "hello" "world")) 4))
==
(boolean? (= (string-length "helloworld") 4))
==
(boolean? (= 10 4))
```

```
==
(boolean? false)
==
true
```

Second, we need a rule from algebra to understand the application of a functions to arguments. Suppose the program contains the definition

```
(define (f x-1 ... x-n)
  f-body)
```

Then an application of a function is governed by the law:

```
(f v-1 ... v-n) == f-body
; with all occurrences of x-1... x-n
; replaced with v-1 ... v-n, respectively
```

Due to the history of languages such as BSL, we refer to this rule as the *beta* or *beta-value* rule.

This rule is as general as possible, so it is best to look at a concrete example. Say the definition is

```
(define (poly x y)
  (+ (expt 2 x) y))
```

Then the first step in an evaluation of `(poly 3 5)` uses beta:

```
(poly 3 5) == (+ (expt 2 3) 5) ... == (+ 8 5) == 13
```

For the rest, DrRacket uses plain arithmetic.

In addition to beta, we need a couple of rules that help us determine the value of `cond` expressions. These rules are algebraic rules but are not a part of the standard algebra curriculum. When the first condition is `false`, then the first `cond`-line disappears:

```
(cond                                  ==  (cond
  [false ...]                              ; the first line disappeared
  [condition-1 answer-1]                   [condition-1 answer-1]
  ...)                                     ...)
```

This rule has the name $\texttt{cond}_{false}$. When the first condition is `true`, DrRacket picks the right-hand side of this `cond` clause:

```
(cond                                  ==  answer
  [true answer]
  [condition-1 answer-1]
  ...)
```

We call this rule $\texttt{cond}_{true}$. It also applies when the condition is `else`.

Consider the following evaluation:

```
(cond
  [(zero? 3) 1]
  [(= 3 3) (+ 1 1)]
  [else 3])
== ; by plain arithmetic and “equals for equals”
(cond
  [false 1]
  [(= 3 3) (+ 1 1)]
  [else 3])
== ; by rule cond_false
(cond
  [(= 3 3) (+ 1 1)]
  [else 3])
== ; by plain arithmetic and “equals for equals”
(cond
  [true (+ 1 1)]
  [else 3])
== ; by rule cond_true
(+ 1 1)
```

The calculation illustrates plain arithmetic, the replacement of equals by equals, and the two `cond` rules.

Exercise 111. Evaluate the following expressions step by step:

```
(+ (* (/ 12 8) 2/3)
   (- 20 (sqrt 4)))

(cond
  [(= 0 0) false]
  [(> 0 1) (string=? "a" "a")]
  [else (= (/  1 0) 9)])

(cond
  [(= 2 0) false]
  [(> 2 1) (string=? "a" "a")]
  [else (= (/  1 2) 9)])
```

Exercise 112. Suppose the program contains these definitions:

```
(define (f x y)
  (+ (* 3 x) (* y y)))
```

Show how DrRacket evaluates the following expressions, step by step:

```
(+ (f 1 2) (f 2 1))

(f 1 (* 2 3))

(f (f 1 (* 2 3)) 19)
```

BSL: Errors

When DrRacket discovers that some parenthesized phrase does not belong to BSL, it signals a *syntax error*. To determine whether a fully-parenthesized program is syntactically legal, DrRacket uses the grammar in figure 29 and reasons along the lines explained above. Not all syntactically legal programs have meaning, however.

When DrRacket evaluates a syntactically legal program and discovers that some operation is used on the wrong kind of value, it raises a *run-time error*. Consider the simple example of `(/ 1 0)`. It is a syntactically legal sentence, but you know from mathematics that 1/0 does not have a value. Since BSL's calculations must be consistent with mathematics, DrRacket signals an error:

```
> (/ 1 0)
/: division by zero
```

Naturally it also signals an error when an expression such as `(/ 1 0)` is nested deeply inside of an another expression:

```
> (+ (* 20 2) (/ 1 (- 10 10)))
/: division by zero
```

DrRacket's behavior translates into our calculations as follows. When we find an expression that is not a value and when the evaluation rules allow no further simplification, we say the computation is *stuck*. This notion of stuck corresponds to a run-time error. For an example, calculating with the expression from above leads to a stuck state:

```
(+ (* 20 2) (/ 1 (- 10 10)))
==
(+ (* 20 2) (/ 1 0))
==
(+ 40 (/ 1 0))
```

What this calculation also shows is that DrRacket eliminates the context of a stuck expression as it signals an error. In this concrete example, it eliminates the addition of `40` to the stuck expression `(/ 1 0)`.

Not all nested stuck expressions end up signaling errors. Suppose a program contains this definition:

```
(define (my-divide n)
```

```
  (cond
    [(= n 0) "inf"]
    [else (/ 1 n)]))
```

If you now apply `my-divide` to `0`, DrRacket calculates as follows:

```
(my-divide 0)
==
(cond
  [(= 0 0) "inf"]
  [else (/ 1 0)])
```

It would obviously be wrong to say that the function signals the division-by-zero error now, even though an evaluation of the shaded subexpression may suggest it. The reason is that `(= 0 0)` evaluates to `true` and therefore the second `cond` clause does not play any role:

```
(my-divide 0)
==
(cond
  [(= 0 0) "inf"]
  [else (/ 1 0)])
==
(cond
  [true "inf"]
  [else (/ 1 0)])
==
"inf"
```

Fortunately, our laws of evaluation take care of these situations automatically. We just need to keep in mind when the laws apply. For example, in

```
(+ (* 20 2) (/ 20 2))
```

the addition cannot take place before the multiplication or division. Similarly, the shaded division in

```
(cond
  [(= 0 0) 'inf]
  [else (/ 1 0)])
```

cannot be substituted for the complete `cond` expression until the corresponding line is the first condition in the `cond`.

As a rule of thumb, it is best to keep the following in mind:

> Simplify the outermost and left-most subexpression that is ready for evaluation.

While this guideline is a simplification, it always explains BSL's results.

In some cases, programmers also want to define functions that raise errors. Recall the checked version of `area-of-disk` from Input Errors:

```
(define (checked-area-of-disk v)
  (cond
    [(number? v) (area-of-disk v)]
    [else (error "number expected")]))
```

Now imagine applying `checked-area-of-disk` to a string:

```
(- (checked-area-of-disk "a")
   (checked-area-of-disk 10))
==
(- (cond
     [(number? "a") (area-of-disk "a")]
     [else (error "number expected")])
   (checked-area-of-disk 10))
==
(- (cond
     [false (area-of-disk "a")]
     [else (error "number expected")])
    (checked-area-of-disk 10))
==
(- (error "number expected")
   (checked-area-of-disk 10))
```

At this point you might try to evaluate the second expression, but even if you do find out that its result is roughly `314`, your calculation must eventually deal with the `error` expression, which is just like a stuck expression. In short, the calculation ends in

```
==
(error "number expected")
```

Boolean Expressions

Our current definition of the Beginning Student BSL language omits two forms of expressions: `and` and `or` expressions. Adding them provides a case study of how to study new language constructs. We must first understand their syntax and then their semantics.

Here is the revised grammar of expressions:

```
expr = ...
     | (and expr expr)
     | (or expr expr)
```

The grammar says that `and` and `or` are keywords, each followed by two expressions. At first glance, the two expressions look like function applications.

To understand why `and` and `or` are not BSL-defined functions, we must look at their pragmatics first. Suppose we need to formulate a condition that determines whether `(/ 1 n)` is `r`:

```
(define (check n r)
  (and (not (= n 0)) (= (/ 1 n) r)))
```

We formulate the condition as an `and` combination of two Boolean expressions because we don't wish to divide by `0` accidentally. Now let us apply `check` to `0` and `1/5`:

```
(check 0 1/5)
==
(and (not (= 0 0)) (= (/ 1 0) 1/5))
```

If `and` were an ordinary operation, we would have to evaluate both subexpressions, and doing so would trigger an error. Instead `and` is not a primitive operation, and its evaluation throws away the second expression when the first one is `false`. In short an `and` expression short-circuits evaluation.

Now it would be easy to formulate evaluation rules for `and` and `or`. Another way to explain their meaning is to show how to translate them into equivalent expressions:

```
(and exp-1 exp-2)
```

is short for

```
(cond
  [exp-1 (if exp-2 true false)]
  [else false])
```

and

```
(or exp-1 exp-2)
```

is short for

```
(cond
  [exp-1 true]
  [else (if exp-2 true false)])
```

So if you are ever in doubt about how to evaluate an `and` or `or` expressions, use the above equivalences to calculate. But we trust that you understand these operations intuitively, and that is almost always enough.

Note You may be wondering why the above abbreviations use

```
(if exp-2 true false)
```

on the right-hand side instead of just

```
exp-2
```

And indeed, if *exp-2* evaluates to `true` or `false`, writing *exp-2* suffices and is shorter than the `if` expression. An expression in BSL may evaluate to all kinds of values, however. If it does evaluate to something other than a Boolean, `(if exp-2 true`

`false)` would signal an error while *exp-2* would not. Since `(and` *exp-1 exp-2*`)` also signals an error when *exp-2* evaluates to something other than a Boolean, we use the above right-hand side and not the inaccurate short-cut.

Exercise 113. The use of `if` may have surprised you in another way because this intermezzo does not mention this form elsewhere. In short, the intermezzo appears to explain `and` with a form that has no explanation either. At this point, we are relying on your intuitive understanding of `if` as a short-hand for `cond`. Write down a rule that shows how to reformulate

```
(if exp-test exp-then exp-else)
```

as a `cond` expression. ∎

Constant Definitions

Programs consist not only of function definitions but also constant definitions, but these weren't included in our first grammar. So here is an extended grammar that includes constant definitions:

```
definition = ...
           | (define name expr)
```

The shape of a constant definition is similar to that of a function definition. It starts with a "(", followed by the keyword `define`, followed by a variable, followed by an expression, and closed by a right parenthesis ")" that matches the very first one. While the keyword `define` distinguishes constant definitions from expressions, it does not differentiate from function definitions. For that, a human reader must look at the second component of the definition.

Next we must understand what a constant definition means. For a constant definition with a literal constant on the right hand side, such as

> As it turns out DrRacket has another way of dealing with function definitions; see Nameless Functions.

```
(define RADIUS 5)
```

it is easy to see that the variable is just a short hand for the value. That is, wherever DrRacket encounters `RADIUS` during an evaluation, it replaces it with `5`.

For a constant definition with a proper expression on the right-hand side, say,

```
(define DIAMETER (* 2 RADIUS))
```

we must immediately determine the value of the expression. This process make use whatever definitions precede this constant definition. Hence,

```
(define RADIUS 5)
```

```
(define DIAMETER (* 2 RADIUS))
```

is equivalent to

```
(define RADIUS 5)
(define DIAMETER 10)
```

This recipe even works when function definitions are involved:

```
(define RADIUS 10)

(define DIAMETER (* 2 RADIUS))

(define (area r) (* 3.14 (* r r)))

(define AREA-OF-RADIUS (area RADIUS))
```

As DrRacket steps through this sequence of definitions, it first determines that `RADIUS` stands for `10`, `DIAMETER` for `20`, and `area` is the name of a function. Finally, it evaluates `(area RADIUS)` to `314` and associates `AREA-OF-RADIUS` with that value.

Mixing constant and function definitions gives rise to a new kind of run-time error, too. Take a look at this program:

```
(define RADIUS 10)

(define DIAMETER (* 2 RADIUS))

(define AREA-OF-RADIUS (area RADIUS))

(define (area r) (* 3.14 (* r r)))
```

It is like the one above with the last two definitions swapped. For the first two definitions, evaluation proceeds as before. For the third one, however, evaluation goes wrong. The recipe calls for the evaluation of `(area RADIUS)`. While the definition of `RADIUS` precedes this expression, the definition of `area` has not yet been encountered. If you were to evaluate the program with DrRacket, you would therefore get an error, explaining that ``this function is not defined.'' So be carefully about using functions in constant definitions only when you know they are defined.

Exercise 114. Evaluate the following program:

```
(define PRICE 5)
(define SALES-TAX (* 0.08 PRICE))
(define TOTAL (+ PRICE SALES-TAX))
```

Does the evaluation of the following program signal an error?

```
(define COLD-F 32)
(define COLD-C (fahrenheit->celsius COLD-F))
(define (fahrenheit->celsius f)
```

```
  (* 5/9 (- f 32)))
```

How about the next one?

```
(define LEFT -100)
(define RIGHT 100)
(define (f x) (+ (* 5 (expt x 2)) 10))
(define f@LEFT (f LEFT))
(define f@RIGHT (f RIGHT))
```

Structure Type Definitions

As you can imagine, `define-struct` is by far the most complex BSL construct. We have therefore saved its explanation for last. Here is the addition to the grammar:

```
definition = ...
           | (define-struct name [name ...])
```

A structure type definition is a third form of definition. The keyword `define-struct` distinguishes this form of definition from function and constant definitions.

Here is a simple example:

```
(define-struct point [x y z])
```

Since `point`, `x`, `y`, and `z` are variables and the parentheses are placed according to the grammatical pattern, it is a proper definition of a structure type. In contrast, these two parenthesized sentences

```
(define-struct (point x y z))

(define-struct point x y z)
```

are illegal definitions, because `define-struct` is not followed by a single variable name and a sequence of variables in parentheses.

While the syntax of `define-struct` is straightforward, its meaning is difficult to spell out with evaluation rules. As mentioned several times, a `define-struct` definition defines several functions at once: a constructor, several selectors, and a predicate. Thus the evaluation of

```
(define-struct c [s-1 ... s-n])
```

introduces into the following functions into the program:

1. `make-c`: a *constructor*;

2. `c-s-1`... `c-s-n`: a series of *selectors*; and

3. `c?`: a *predicate*.

These functions have the same status as `+`, `-`, or `*`. Before we can understand the rules that govern these new functions, however, we must return to the definition of values. After all, one purpose of `define-struct` is to introduce a distinct class of values.

Simply put, the use of `define-struct` extends the universe of values. To start with, it now also contains structures, which compound several values into one. When a program contains a `define-struct` definition, its evaluation modifies the definition of values:

> A *value*, also a called a piece of data, is one of: a *number*, a *boolean*, a *string*, an *image*,
>
> - or a structure value:
>
>   ```
>   (make-c _value-1 ... _value-n)
>   ```
>
>   assuming a structure type definition for `c` exists and the constructor `make-c` is available.

For example, the definition of `point` means the addition of values of this shape:

```
(make-point 1 2 -1)
(make-point "one" "hello" "world")
(make-point 1 "one" (make-point 1 2 -1))
...
```

Now we are in a position to understand the evaluation rules of the new functions. If `c-s-1` is applied to a `c` structure, it returns the first component of the value. Similarly, the second selector extracts the second component, the third selector the third component, and so on. The relationship between the new data constructor and the selectors is best characterized with *n* equations added to BSL's rules:

```
(c-s-1 (make-c V-1 ... V-n))     ==     V-1

(c-s-n (make-c V-1 ... V-n))     ==     V-n
```

For our running example, we get the specific equations

```
(point-x (make-point V U W))     ==     V

(point-y (make-point V U W))     ==     U

(point-z (make-point V U W))     ==     W
```

When DrRacket encounters `(point-y (make-point 3 4 5))` in a calculation, it equates it with `4` and `(point-x (make-point (make-point 1 2 3) 4 5))` to `(make-point 1 2 3)`.

The predicate `c?` can be applied to any value. It returns `true` if the value is of kind `c` and `false` otherwise. We can translate both parts into two equations:

```
(c? (make-c V-1 ... V-n))      ==      true

          (c? V)               ==      false
```

if `V` is a value not constructed with `make-c`. Again, the equations are best understood in terms of our example:

```
(point? (make-point U V W))    ==      true

         (point? X)            ==      false
```

if `X` is a value but not a `point` structure. Thus, `(point? (make-point 3 4 5))` is `true` and `(point? 3)` is `false`.

Exercise 115. Pick the legal from the legal from illegal sentences in the following list:

1. `(define-struct oops [])`

2. `(define-struct child [parents dob date])`

3. `(define-struct (child person) [dob date])`

Explain why the sentences are legal or illegal. ∎

Exercise 116. Identify which of the following expressions are *value*s:

1. `(make-point 1 2 3)`

2. `(make-point (make-point 1 2 3) 4 5)`

3. `(make-point (+ 1 2) 3 4)`

Explain why the expressions are values or not. ∎

Exercise 117. Suppose the program contains

```
(define-struct ball [x y speed-x speed-y])
```

Predict the results of evaluating the following expression:

1. `(number? (make-ball 1 2 3 4))`

2. `(ball-speed-y (make-ball (+ 1 2) (+ 3 3) 2 3))`

3. `(ball-y (make-ball (+ 1 2) (+ 3 3) 2 3))`

4. `(ball-x (make-posn 1 2))`

5. `(ball-speed-y 5)`

Verify your predictions with DrRacket. ∎

BSL: Tests

Figure 31 collects all the elements of BSL that this intermezzo explains plus the tests: `check-expect`, `check-within`, and `check-error`. The actual grammar of BSL is even a bit larger; see the Help Desk for details.

```
      program = def-or-expr ...

  def-or-expr = definition
              | expr
              | test-case

   definition = (define (name variable variable ...) expr)
              | (define name expr)
              | (define-struct name [name ...])

         expr = (name expr expr ...)
              | (cond [expr expr] ... [expr expr])
              | (cond [expr expr] ... [else expr])
              | (and expr expr expr ...)
              | (or expr expr expr ...)
              | name
              | number
              | string
              | image

    test-case = (check-expect expr expr)
              | (check-within expr expr expr)
              | (check-error expr)
```

Figure 31: Beginning Student Language, Full Grammar

The meaning of testing expressions is easy to explain. When you click the “Run” button DrRacket collects all testing expressions and moves them from the middle of the program to the end, retaining in the order in which they appear. Then it evaluates the definitions, expressions, and tests and allows you to explore additional expressions in the interactions area.

In principle, the evaluation of a test evaluates its pieces and then compares them with an equality predicate, `equal?` to be precise. Beyond that, tests communicate with DrRacket to collect some success and failure statistics and information on how to display test failures.