v.6.11.0.4

Intermezzo 1: Beginning Student Language

Fixed-Size Data deals with BSL as if it were a natural language. It introduces the "basic words" of the language, suggests how to compose "words" into "sentences," and appeals to your knowledge of algebra for an intuitive understanding of these "sentences." While this kind of introduction works to some extent, truly effective communication requires some formal study.

In many ways, the analogy of Fixed-Size Data is correct. A programming language does have a vocabulary and a grammar, though programmers use the word *syntax* for these elements. A sentence in BSL is an expression or a definition. The grammar of BSL dictates how to form these phrases. But 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*.

This intermezzo presents BSL as if it were an extension of the familiar language of arithmetic and algebra in middle school. 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 syntax and semantics of a good portion of BSL. Based on this new understanding of BSL, the fourth resumes our discussion of errors. The remaining sections expand this understanding to the complete language; the last one expands the tools for expressing tests.

> Programmers must eventually understand these principles of **computation**, but they are complementary to the principles of **design**.

BSL Vocabulary

Figure 39 introduces and defines BSL's basic vocabulary. It consists of literal constants, such as numbers or Boolean values; names that have meaning according to BSL, for example, `cond` or `+`; and names to which programs can give meaning via `define` or function parameters.

A *`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`* is one of:

- A *`number`* is one of: `1`, `-1`, `3/5`, `1.22`, `#i1.22`, `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: `#true` or `#false`.
- A *`string`* is one of: `""`, `"he says \"hello world\" to you"`, `"doll"`, and so on. In general, it is a sequence of characters enclosed by a pair of `"`.

- 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 39: BSL core vocabulary

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

BSL Grammar

Figure 40 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, though, is to discuss how to read such grammars. Each line with a = introduces a *syntactic category*; the best way to pronounce = is as "is one of" and | as "or." Wherever you see three dots, imagine 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-expr* or a sequence of two of them, or three, four, five, or however many. Since this example is not particularly illuminating, let's look at the second syntactic category. It says that *def* is either

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

```
(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-expr ...

def-expr = def
         | expr

     def = (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 40: 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 40 `42` is a sentence of BSL:

- The first syntactic category says that a program is a *def-expr*. Expressions may refer to the definitions.
- The second one tells us that a *def-expr* is either a *def* or an *expr*.
- The last definition lists all ways of forming an *expr*, and the second one is *value*.

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

Since we know from figure 39 that `42` is a value, we have confirmation.

The interesting parts of the grammar show how to form compound sentences, those built from other sentences. For example, the *def* 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. Note how the leading keyword `define` distinguishes definitions from expressions.

Expressions (*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. Like `define`, the keyword `cond` distinguishes conditional expressions from 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, these parenthesized sentences are not legal 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 carefully reading code examples in books, paying attention to how they are formatted.

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

Exercise 116. Take a look at the following sentences:

1. `x`
2. `(= y z)`
3. `(= (= y z) 0)`

Explain why they are syntactically legal expressions

Exercise 117. Consider the following sentences:

1. `(3 + 4)`

2. `number?`

3. `(x)`

Explain why they are syntactically illegal.

Exercise 118. Take a look at the following sentences:

1. `(define (f x) x)`

2. `(define (f x) y)`

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

Explain why they are syntactically legal definitions

Exercise 119. Consider the following sentences:

1. `(define (f "x") x)`

2. `(define (f x y z) (x))`

Explain why they are syntactically illegal.

Exercise 120. Discriminate the legal from the illegal sentences:

1. `(x)`

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

3. `(+ 1 2 3)`

Explain why the sentences are legal or illegal. Determine whether the legal ones belong to the category *expr* or *def*.

Note on Grammatical Terminology The components of compound sentences have names. We have introduced some of these names on an informal basis. Figure 41 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 41: Syntactic naming conventions

In addition to the terminology of figure 41, we say *function header* for the second component of a definition. Accordingly, the expression component is called a *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. **End**

BSL Meaning

When you hit the return key on your keyboard and ask DrRacket to evaluate an expression, it uses the laws of arithmetic and algebra to obtain a *value*. For the variant of BSL treated so far, figure 39 defines grammatically what a value is—the set of values is just a subset of all expressions. The set includes Booleans, Strings, and Images.

The rules of evaluation come in two categories. An infinite number of rules, like those of arithmetic, explain how to determine the value of an application of a primitive operation to values:

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

Remember == says that two expressions are equal according to the laws of computation in BSL. 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
...
```

And, like in algebra, you can always replace equals with equals; see figure 42 for a sample calculation.

```
(boolean? (= (string-length (string-append "h" "w"))
             (+ 1 3)))
==
(boolean? (= (string-length (string-append "h" "w")) 4))
==
(boolean? (= (string-length "hw") 4))
==
(boolean? (= 2 4))
==
(boolean? #false)
== #true
```

Figure 42: Replacing equals by equals

Second, we need a rule from algebra to understand the application of a function 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 formulated as generally as possible, so it is best to look at a concrete

See Computing with lambda for more on this rule.

example. Say the definition is

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

and DrRacket is given the expression `(poly 3 5)`. Then the first step in an evaluation of the expression uses the beta rule:

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

In addition to beta, we need rules that determine the value of `cond` expressions. These rules are algebraic even if they are not explicitly taught as part of the standard curriculum. When the first condition is `#false`, the first `cond`-line disappears, leaving the rest of the lines intact:

```
(cond                    == (cond
  [#false ...]                ; first line removed
  [condition2 answer2]        [condition2 answer2]
  ...)                        ...)
```

This rule has the name $\texttt{cond}_{false}$. Here is $\texttt{cond}_{true}$:

```
(cond                    == answer-1
  [#true answer-1]
  [condition2 answer2]
  ...)
```

The rule also applies when the first 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 the rules of plain arithmetic, the replacement of equals by equals, and both `cond` rules.

Exercise 121. Evaluate the following expressions step-by-step:

1.
```
(+ (* (/ 12 8) 2/3)
   (- 20 (sqrt 4)))
```
2.
```
(cond
  [(= 0 0) #false]
  [(> 0 1) (string=? "a" "a")]
  [else (= (/  1 0) 9)])
```
3.
```
(cond
  [(= 2 0) #false]
  [(> 2 1) (string=? "a" "a")]
  [else (= (/  1 2) 9)])
```

Use DrRacket's stepper to confirm your computations.

Exercise 122. Suppose the program contains these definitions:

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

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

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

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

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

Use DrRacket's stepper to confirm your computations.

Meaning and Computing

The stepper tool in DrRacket mimics a student in a pre-algebra course. Unlike you, the stepper is is extremely good at applying the laws of arithmetic and algebra as spelled out here, and it is also extremely fast.

> A scientist calls the stepper a **model** of DrRacket's evaluation mechanism. Refining Interpreters presents another model, an **interpreter**.

You can, and you ought to, use the stepper when you don't understand how a new language construct works. The sections on **Computing** suggest exercises for this purpose, but you can make up your own examples, run them through the stepper, and ponder why it takes certain steps.

Finally, you may also wish to use the stepper when you are surprised by the result that a program computes. Using the stepper effectively in this way requires practice. For example, it often means copying the program and pruning unnecessary pieces. But once you understand how to use the stepper well this way, you will find that this procedure clearly explains run-time errors and logical mistakes in your programs.

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 40 and reasons along the lines explained above. Not all syntactically legal programs have meaning, however.

> For a nearly full list of error messages, see the last section of this intermezzo.

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 syntactically legal expression `(/ 1 0)`, which, as you know from mathematics, has no 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 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 example, computing the value of the above expression 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 sub-expression 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 remember when they 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:

> *Always choose the outermost and left-most nested expression that is ready for evaluation.*

While this guideline may look simplistic, 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 BSL omits `or` and `and` 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. They are **not** 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 because we don't wish to divide by `0` accidentally. Now let's 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 sub-expressions, and doing so would trigger an error. Instead `and` simply does not evaluate the second expression when the first one is `#false`, meaning, `and` *short-circuits* evaluation.

It would be easy to formulate evaluation rules for `and` and `or`. Another way to explain their meaning is to translate them into other expressions:

> To make sure *expr-2* evaluates to a Boolean value, these abbreviations should use `(if expr-2 #true #false)` instead of just *expr-2*. We gloss over this detail here.

`(and exp-1 exp-2)` is short for

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

and

`(or exp-1 exp-2)` is short for

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

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

Exercise 123. 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 of 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. 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.

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

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

```
(define RADIUS 5)
```

the variable is just a short-hand for the value. Wherever DrRacket encounters `RADIUS` during an evaluation, it replaces it with `5`.

For a 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 makes use of whatever definitions precede this constant definition. Hence,

```
(define RADIUS 5)
(define DIAMETER (* 2 RADIUS))
```

is equivalent to

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

This process 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 process 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 careful to use functions in constant definitions only when you know they are defined.

Exercise 124. Evaluate the following program, step-by-step:

```
(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))
```

Check your computations with DrRacket's stepper.

Structure Type Definitions

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

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

A structure type definition is a third form of definition. The keyword distinguishes it from both 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 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 a `define-struct` is to introduce a class of values that is distinct from all existing 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* 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 `c` is defined.

For example, the definition of `point` adds 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 sees `(point-y (make-point 3 4 5))`, it replaces the expression with `4` while `(point-x (make-point (make-point 1 2 3) 4 5))` evaluates 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.

Exercise 125. Discriminate the legal from the illegal sentences:

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 126. Identify the *value*s among the following expressions, assuming the definitions area contains these structure type definitions:

```
(define-struct point [x y z])
(define-struct none  [])
```

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

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

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

4. `(make-none)`

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

Explain why the expressions are values or not.

Exercise 127. 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)`

Check your predictions in the interactions area and with the stepper.

```
    def-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-member-of expr expr ...)
             | (check-range expr expr expr)
             | (check-error expr)
             | (check-random expr expr)
             | (check-satisfied expr name)
```

Figure 43: BSL, full grammar

BSL Tests

Figure 43 presents all of BSL plus a number of testing forms.

The general meaning of testing expressions is easy to explain. When you click the *RUN* button, DrRacket collects all testing expressions and moves them to the end of the program, retaining the order in which they appear. It then evaluates the content of the definitions area. Each test evaluates its pieces and then compares them with the expected outcome via some predicate. Beyond that, tests communicate with DrRacket to collect some statistics and information on how to display test failures.

For details, read the documentation on these test forms. Here are some illustrative examples:

```
; check-expect compares the outcome and the expected value with equal?
(check-expect 3 3)

; check-member-of compares the outcome and the expected values with equal?
; if one of them yields #true, the test succeeds
(check-member-of "green" "red" "yellow" "green")

; check-within compares the outcome and the expected value with a predicate
;  like equal? but allows for a tolerance of epsilon for each inexact number
(check-within (make-posn #i1.0 #i1.1) (make-posn #i0.9 #i1.2) 0.2)

; check-range is like check-within
; but allows for a specification of an interval
(check-range 0.9 #i0.6 #i1.0)

; check-error checks whether an expression signals (any) error
(check-error (/ 1 0))

; check-random evaluates the sequences of calls to random in the
; two expressions such that they yield the same number
(check-random (make-posn (random 3) (random 9))
              (make-posn (random 3) (random 9)))

; check-satisfied determines whether a predicate produces #true
; when applied to the outcome, that is, whether outcome has a certain property
(check-satisfied 4 even?)
```

All of the above tests succeed. The remaining parts of the book re-introduce these test forms as needed.

Exercise 128. Copy the following tests into DrRacket's definitions area:

```
(check-member-of "green" "red" "yellow" "grey")
(check-within (make-posn #i1.0 #i1.1)
              (make-posn #i0.9 #i1.2)  0.01)
(check-range #i0.9 #i0.6 #i0.8)
(check-random (make-posn (random 3) (random 9))
              (make-posn (random 9) (random 3)))
(check-satisfied 4 odd?)
```

Validate that all of them fail and explain why.

BSL Error Messages

A BSL program may signal many kinds of syntax errors. While we have developed BSL and its error reporting system specifically for novices who, by definition, make mistakes, error messages need some getting used to.

Below we sample the kinds of error messages that you may encounter. Each entry in one of the listings consists of three parts:

- the code fragments that signal the error message;
- the error message; and
- an explanation with a suggestion on how to fix the mistake.

Consider the following example, which is **the worst possible error message** you may ever see:

```
(define (absolute n)
  (cond
    [< 0 (- n)]
    [else n]))
```

```
<: expected a function call, but
there is no open parenthesis before
this function
```

A `cond` expression consists of the keyword followed by an arbitrarily long sequence of `cond` clauses. In turn, every clause consists of two parts: a condition and an answer, both of which are expressions. In this definition of `absolute`, the first clause starts with `<`, which is supposed to be a condition but isn't even an expression according to our definition. This confuses BSL so much that it does not "see" the open parenthesis to the left of `<`. The fix is to use `(< n 0)` as the condition.

The highlighting of `<` in the function definition points to the error. Below the definition, you can see the error message that DrRacket presents in the interactions window if you click *RUN*. Study the explanation of the error on the right to understand how to address this somewhat self-contradictory message. And now rest assured that no other error message is even remotely as opaque as this one.

So, when an error shows up and you need help, find the appropriate figure, search the entries for a match, and then study the complete entry.

Error Messages about Function Applications in BSL

Assume that the definitions area contains the following and nothing else:

```
; Number Number -> Number
; finds the average of x and y
(define (average x y)
  (/ (+ x y)
     2))
```

Hit the *RUN* button. Now you may encounter the error messages below.

```
(f 1)
```

```
f: this function is not defined
```

The application names `f` as the function, and `f` is not defined in the definitions area. Define the function, or make sure that the variable name is spelled correctly.

```
(1 3 "three" #true)
```

```
function call: expected a function
after the open parenthesis, but found
a number
```

An open parenthesis must always be followed by a keyword or the name of a function, and `1` is neither. A function name is either defined by BSL, say `+`, or in the definitions area, say `average`.

```
(average 7)
```
```
average: expects 2 arguments, but
found only 1
```

This function call applies `average` to one argument, `7`, even though its definition calls for two numbers.

```
(average 1 2 3)
```
```
average: expects 2 arguments, but
found 3
```

Here `average` is applied to **three** numbers instead of two.

```
(make-posn 1)
```
```
make-posn: expects 2 arguments, but
found only 1
```

Functions defined by BSL must also be applied to the correct number of arguments. For example, `make-posn` must be applied to two arguments.

Error Messages about Wrong Data in BSL

The error scenarios below again assume that the definitions area contains the following:

```
; Number Number -> Number
; find the average of x and y
(define (average x y) ...)
```

Remember that `posn` is a pre-defined structure type.

```
(posn-x #true)
```
```
posn-x: expects a posn, given #true
```

A function must be applied to the arguments it expects. For example, `posn-x` expects an instance of `posn`.

```
(average "one" "two")
```
```
+: expects a number as 1st argument,
given "one"
```

A function defined to consume two Numbers must be applied to two Numbers; here `average` is applied to Strings. The error message is triggered only when `average` applies `+` to these Strings. Like all primitive operations, `+` is a checked function.

Error Messages about Conditionals in BSL

This time we expect a constant definition in the definitions area:

```
; N in [0,1,...10)
(define 0-to-9 (random 10))
```

```
(cond
  [(>= 0-to-9 5)])
```
```
cond: expected a clause with a
question and an answer, but found a
clause with only one part
```

Every `cond` clause must consist of exactly two parts: a condition and an answer. Here `(>= 0-to-9 5)` is apparently intended as the condition; the answer is missing.

```
(cond
  [(>= 0-to-9 5)
   "head"
   "tail"])
```
```
cond: expected a clause with a
```

In this case, the `cond` clause consists of three parts, which also violates the grammar. While `(>= 0-to-9 5)` is clearly intended to be the condition, the clause comes with two answers:

`question and an answer, but found a clause with 3 parts`	`"head"` and `"tail"`. Pick one or create a single value from the two strings.

`(cond)`	A conditional must come with at least one `cond` clause and usually it comes with at least two.
`cond: expected a clause after cond, but nothing's there`	

Error Messages about Function Definitions in BSL

All of the following error scenarios assume that you have placed the code snippet into the definitions area and hit *RUN*.

`(define f(x) x)`	A definition consist of three parts: the `define` keyword, a sequence of variable names enclosed in parentheses, and an expression. This definition consists of four parts; this definition is an attempt to use the standard notation from algebra courses for the header `f (x)` instead of `(f x)`.
`define: expected only one expression after the variable name f, but found 1 extra part`	

`(define (f x x) x)`	The sequence of parameters in a function definition must not contain duplicate variables.
`define: found a variable that is used more than once: x`	

`(define (g) x)`	In BSL a function header must contain at least two names. The first one is the name of the function; the remaining variable names are the parameters, and they are missing here.
`define: expected at least one variable after the function name, but found none`	

`(define (f (x)) x)`	The function header contains `(x)`, which is **not** a variable name.
`define: expected a variable, but found a part`	

`(define (h x y) x y)`	This function definition comes with two expressions following the header: `x` and `y`.
`define: expected only one expression for the function body, but found 1 extra part`	

Error Messages about Structure Type Definitions in BSL

Now you need to place the structure type definitions into the definitions area and hit *RUN* to experiment with the following errors.

`(define-struct [x])` `(define-struct [x y])`	A structure type definition consists of three parts: the `define-struct` keyword, the structure's name, and a sequence of names in parentheses. Here the structure's name is missing.
`define-struct: expected the structure name after define-struct, but found a part`	

`(define-struct x` `  [y y])`	The sequence of field names in a structure type definition must not contain duplicate names.
`define-struct: found a field name`	

```
that is used more than once: y
```

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

```
define-struct: expected at least one
field name (in parentheses) after the
structure name, but found something
else
```

These structure type definitions lack the sequence of field names, enclosed in parentheses.