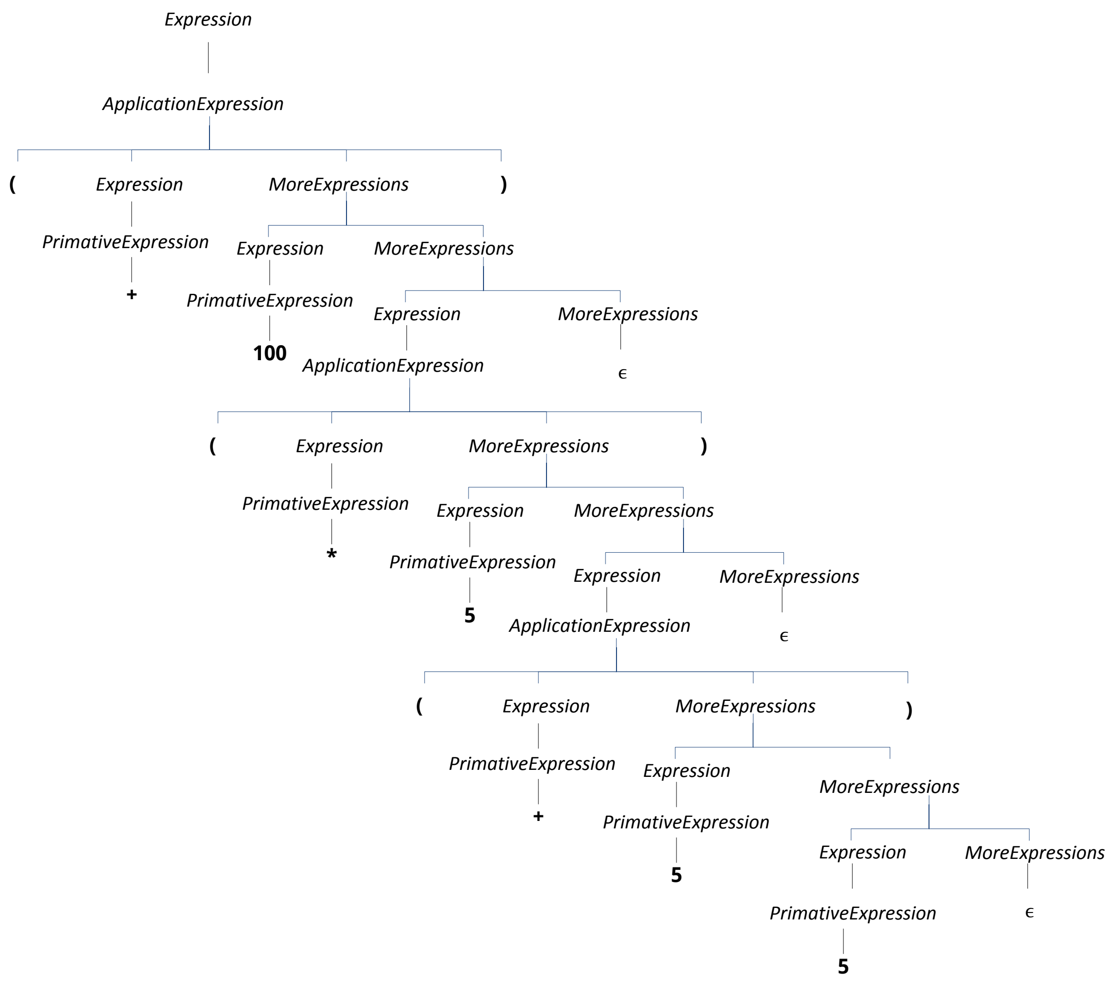
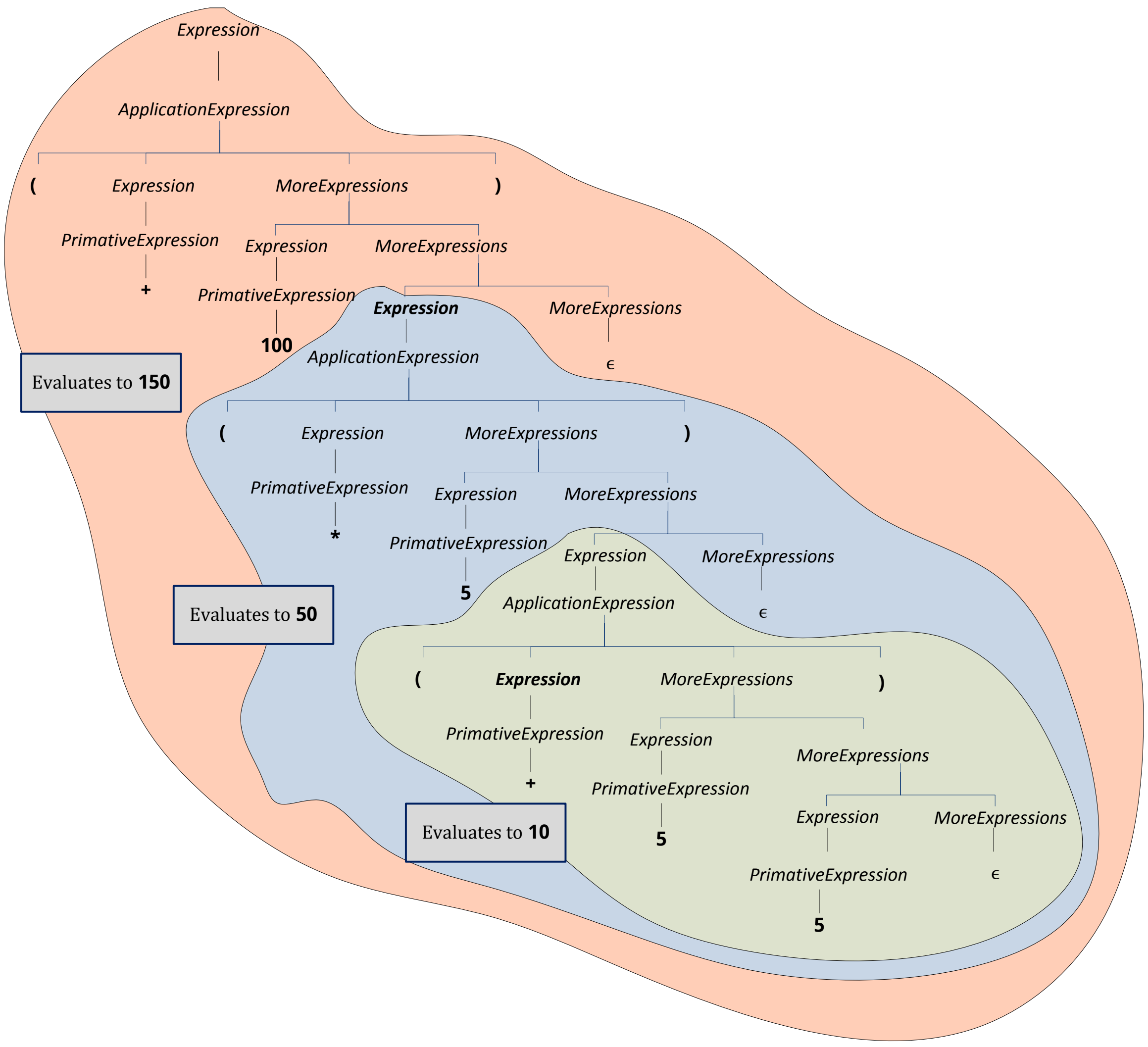
3 Programming

Exercise 3.1. Draw a parse tree for the Scheme expression (+ 100 (∗ 5 (+ 5 5))) and show how it is evaluated.

Solution. The full parse tree for the expression is:



The main subexpressions are evaluated as shown below to produce the final value of 150.



Exercise 3.2. Predict how each of the following Scheme expressions is evaluated. After making your prediction, try evaluating the expression in DrRacket. If the result is different from your prediction, explain why the Scheme interpreter evaluates the expression as it does.

a. 1120

Solution. 1120

b. (+ 1120)

Solution. 1120

c. (+ (+ 10 20) (* 2 0))

Solution. 30

d. (= (+ 10 20) (* 15 (+ 5 5)))

Solution. #f

The #f symbol represents the Boolean value *false*. Since the two operand expressions for the = have different values, the expression evaluates to #f.

e. $+$

Solution. `#<procedure:+>`

The symbol $+$ is a primitive expression that is the built-in procedure for addition.

f. $(+ + <)$

Solution. ⊗ `+: expects type <number> as 1st argument, given: #<procedure:+>; other arguments were: #<procedure:<>`

The interpreter produces an error since the $+$ procedure is only defined for operands that are numbers. Since the value of the first argument is a procedure, there is a type error.

Exercise 3.3. For each question, construct a Scheme expression and evaluate it in DrRacket.

a. How many seconds are there in a year?

Solution. For non-leap years, there are 365 days in a year, 24 hours in a day, 60 minutes in an hour, and 60 seconds in a minute: `(* 60 60 24 365)`

b. For how many seconds have you been alive?

Solution. The first number (e.g., `20` here) is the number of years you have been alive: `(* 20 60 60 24 365)`

c. For what fraction of your life have you been in school?

Solution. Let's assume you are 20 years old and you've been in school for 15 years and in the United States it is typical to have 180 school days in a year, and that each school day is 7 hours long. Then, we can compute the fraction of your life spent in school by dividing the number of hours spent in school by the number of hours lived: `(/ (* 15 180 7) (* 20 365 24))`. This evaluates to `63/584`. Note that math in Scheme is exact, and it is represented as a rational number. A more useful result is to convert it to a decimal number using the *exact->inexact* procedure:

```
> (exact->inexact (/ (* 15 180 7) (* 20 365 24)))
0.10787671232876712
```

Your answer, of course, will vary depending on how old you are, how many years you've spent in school, and the length of your school year and day.

Exercise 3.4. Construct a Scheme expression to calculate the distance in inches that light travels during the time it takes the processor in your computer to execute one cycle. (A meter is defined as the distance light travels in $1/299792458^{th}$ of a second in a vacuum. Hence, light travels at $299,792,458$ meters per second. Your processor speed is probably given in *gigahertz* (GHz), which are 1,000,000,000 hertz. One hertz means once per second, so 1 GHz means the processor executes 1,000,000,000 cycles per second. On a Windows machine, you can find the speed of your processor by opening the Control Panel (select it from the Start menu) and selecting System. Note that Scheme performs calculations exactly, so the result will be displayed as a fraction. To see a more useful answer, use (*exact->inexact Expression*) to convert the value of the expression to a decimal representation.)

Solution. My computer is 2.33 GHz. Hence, one cycle takes `(/ 1 2330000000)` seconds. Light travels `299792458` meters per second. So, in the time the processor executes a single cycle, light travels

```
> (exact->inexact (* 299792458 (/ 1 2330000000)))
0.12866629098712445
```

meters. This is equivalent to

```
> (exact->inexact (/ (* 100 (* 299792458 (/ 1 2330000000))) 2.54))
5.065602007367104
```

inches. This should give you some sense of how close the processor speed is to reaching physical limits.

Exercise 3.5. Define a procedure, *cube*, that takes one number as input and produces as output the cube of that number.

Solution.

```
(define cube (lambda (x) (* x x x)))
```

or:

```
(define (cube x) (* x x x))
```

Exercise 3.6. Define a procedure, *compute-cost*, that takes as input two numbers, the first represents that price of an item, and the second represents the sales tax rate. The output should be the total cost, which is computed as the price of the item plus the sales tax on the item, which is its price times the sales tax rate. For example, (*compute-cost* 13 0.05) should evaluate to 13.65.

Solution.

```
(define compute-cost
   (lambda (price rate)
      (+ price (* price rate))))
```

Another approach:

```
(define (compute-cost price rate)
   (* price (+ 1 rate)))
```

Exercise 3.7. Follow the evaluation rules to evaluate the Scheme expression:

```
(bigger 3 4)
```

where *bigger* is the procedure defined above. (It is very tedious to follow all of the steps (that's why we normally rely on computers to do it!), but worth doing once to make sure you understand the evaluation rules.)

Exercise 3.8. Define a procedure, *xor*, that implements the logical exclusive-or operation. The *xor* function takes two inputs, and outputs true if exactly one of those outputs has a true value. Otherwise, it outputs false. For example, (*xor true true*) should evaluate to false and (*xor* (< 3 5) (= 8 8)) should evaluate to true.

Solution. There are many possible ways to define *xor*. Here are two possibilities:

```
(define (xor a b)
   (if a (not b) b))
```

```
(define (xor a b)
  (or (and a (not b)) (and (not a) b)))
```

Exercise 3.9. Define a procedure, *absvalue*, that takes a number as input and produces the absolute value of that number as its output. For example, (*absvalue* 3) should evaluate to 3 and (*absvalue* −150) should evaluate to 150.

Solution.

```
(define (absvalue v)
  (if (< v 0) (- v) v))
```

Note that Scheme provides a built-in function *abs* that implements the absolute value function. Hence, the easiest way to define *absvalue* would be,

```
(define absvalue abs)
```

Exercise 3.10. Define a procedure, *bigger-magnitude*, that takes two inputs, and outputs the value of the input with the greater magnitude (that is, absolute distance from zero). For example, (*bigger-magnitude* 5 −7) should evaluate to −7, and (*bigger-magnitude* 9 −3) should evaluate to 9.

Solution. We use the *absvalue* procedure defined in the previous exercise:

```
(define (bigger-magnitude a b)
  (if (> (absvalue a) (absvalue b)) a b))
```

Exercise 3.11. Define a procedure, *biggest*, that takes three inputs, and produces as output the maximum value of the three inputs. For example, (*biggest* 5 7 3) should evaluate to 7. Find at least two different ways to define *biggest*, one using *bigger*, and one without using it.

Solution. Our first solution uses the *bigger* procedure from Example 3.3:

```
(define (biggest a b c)
  (bigger (bigger a b) c))
```

Another approach is to define an if expression:

```
(define (biggest a b c)
  (if (> a b)
    (if (> a c) a c)
    (if (> b c) b c)))
```

We prefer the first version since it is shorter and easier to understand. This is a simple illustration of the advantages of building up more complex procedures by combining simple ones.