1.0 Expressions

Dr. Scheme **evaluates expressions** so we will start by using the interactions window and asking Dr. Scheme to evaluate some expressions.

Numbers are examples of **primitive** expressions, meaning Scheme knows what they are and how to evaluate them ... basically they are built in.

Type the number **12** into the interactions window, and when you hit the enter/return key Scheme will evaluate the number **12** and print the result to the screen. You should notice that it simply printed **12** back to the screen, does that make sense to you? It should be clear that **12** is just **12**, **26** is **26** and so on.

Example 1.0:

```
> 12
12
> 26
26
> 72
72
>
```

1.1 Procedures

You have been doing simple math since early grade school and it typically takes the following form:

```
3 + 4
4 + 5 + 6
12 * 5
12 / 3
30 - 5
```

This is commonly referred to as **infix** notation because the **operators** (*, /, +, -) are in between the **operands**. So if you are going to add 3 numbers then you will have 2 occurrences of the **+** operator. In a more general case, for **x** number of operands you will need **x - 1** operators.

In Scheme we will call operators **procedures** because they do something, in particular it instructs Scheme what to apply to the operands which we will call **arguments**. However in Scheme procedures are NOT infix, they adopt what is called **prefix** notation. Which you may have guessed by now that the procedure comes before (pre) the arguments. This greatly simplifies the notation because you only have to specify the name of the procedure (the operator) once.

General Rule 1.0
Scheme expressions begins with a left parenthesis ‘**(**‘.

General Rule 1.1
Immediately following a left parenthesis you must specify the name of a procedure.

General Rule 1.2
After following the name of a procedure you list the arguments separated by spaces.

General Rule 1.3
Scheme expressions ends with a right parenthesis ‘**)**’

The previous infix examples will be written in the following way after applying General Rules 1.0 - 1.3. This will take some practice before you become proficient writing and reading expressions in this form. Now please go to the interaction window and evaluate all the given expressions.

```
(+ 3 4)
(+ 4 5 6)
(* 12 5)
(/ 12 3)
(- 30 5)
```

```
> (+ 3 4)
7
> (+ 4 5 6)
11
> (* 12 5)
60
> (/ 12 3)
4
> (- 30 5)
25
>
```

Please take this moment to play around and practice. Can you evaluate other math expressions, what about combining them into longer more complicated expressions? Also notice how Scheme answers you back, you ask it to ADD 3 and 4 `(+ 3 4)` and it answers you with 7.

When Scheme answers a request we call that a **return value** and in the above case it returned 7. By design Scheme only returns at most 1 value for any given expression, and always returns what the LAST expression returns when a sequence of expressions are evaluated. This leads us to combining expression together to answer more complex questions, **called means of combinations**.

Schemes model for combining expressions is simple, the prefix notation allows for nesting expressions. So given the infix equation **`2 * 5 + 4 - 2`** which can be translated into the following prefix form `(+ (* 2 5) (- 4 2))` or `(- (+ (* 2 5) 4) 2)`. Below several more examples are illustrated, and each is fully parenthesized meaning that order of operation is clear and there is no ambiguity. It is also important to realize that these nesting combinations can be as long and as complicated as they need to be. It does not matter for Scheme, however you may want to format them differently so that it is easier for us humans to read, this is known as ‘**Pretty Printing**’.

Example 1.1:

Infix	Prefix
`4 + 5 - 1 / 2`	`(- (+ 4 5) (/ 1 2))`
`10 * 2 / 4 + 5`	`(+ (/ (* 10 2) 4) 5)`
`10 + 5 / 5 - 3 * 4`	`(- (+ 10 (/ 5 5)) (* 3 4))`
`5 - 3 * 5 / 2 + 3 / 2 + 2`	`(+ (- 5 (/ (* 3 5) 2)) (/ 3 2) 2)`

Please take this time to enter these expressions into the interactions window and experiment with translating your own math expressions and checking them with a calculator to make sure you are getting the order of operations correct.

Let us examine what Scheme is doing briefly when it evaluates these expressions, when you enter the following into the interactions window:

```
(- (+ 4 5) (/ 1 2))
```

Scheme evaluates the inner most set of parenthesis first. In this example the `(+ 4 5)` and `(/ 1 2)` are both on the same level (the second set of parenthesis). So the previous line then becomes:

```
(- (+ 4 5) (/ 1 2)) -> (- 9 (/ 1 2)) -> (- 9 .5) -> 8.5
```

This is a powerful method of combination and evaluation which we will explore in more detail in the next chapter, but make sure you understand this basic concept now.

1.2 Pretty Printing

When entering length expressions Scheme does not get confused, however these will be hard for humans to read easily. There is a technique that uses line separation and indentation to make long expressions easier to read. Scheme ignores **white space** characters longer than one space, white spaces include newlines, spaces, and tabs.

Typically pretty printing is used in the definitions window, and you will notice that Scheme will automatically handle the appropriate indentation. A translation for each of the previous prefix expressions in Example 1.1 are below:

```
(- (+ 4 5)
   (/ 1 2))
```

```
(+ (/ (* 10 2)
      4)
   5)
```

```
(- (+ 10
      (/ 5 5))
   (* 3 4))
```

```
(+ (- 5
      (/ (* 3 5)
         2))
   (/ 3 2)
   2)
```

Given this tool to improve human readability it is not always appropriate to use. However given a

long and complicated enough expression this will provide to be useful. The final decision on when to use this technique will ultimately be up to each and every programmer, but this is considered good practice to use. Over time you will learn when and when not to use pretty printing.

Another important concept to improve readability is to include **comments**. Comments are solely for humans, Scheme ignores them completely. Comment are started with a semi-colon '**;**' and include everything on the same line following the semi-colon. When dealing with more complex procedures and expression comments are often useful to remind the programmer what is going on in more straight forward terms. Well commented code will be much easier to go back and debug or modify at a later date.

1.3 Names

In order for Scheme or any programming language to talk about something or use something it has to have a name. Scheme uses names to reference items, suppose the following:

You are strolling through a store you see something you are interested in, but decide not to purchase it at this moment. You go home, then call the store only to order the item. Once a clerk gets on the phone you realized you did not get the name of the item you wanted. You tell the clerk "I saw a grill in the store that I liked and want to order it." The clerk however informs you that they carry more than 100 different types of grills and cannot conclude with the information you provided him which grill you are referring to.

This simple problem could be avoided if you just had the name of the grill or item you were referring to. So Scheme gives you the ability to assign things names using the **define** procedure. Define is an operator and it takes 2 operands, the name you are assigning and the expression you are assigning the name to.

```
(define ten 10)
```

Assigns the name '**ten**' to the expression 10, so do this in the interactions window. Notice that when you enter this expression in the interactions window nothing is returned. However Scheme now knows the name 10, and you can now use this name in other expressions **`(+ ten ten)`**. This would return the value 20. Using define to refer to complex expression provides a simple but powerful means of **abstraction**.

Example 1 .2

```
> (define pi 3.14)
> (define radius 5)
> (define circumference (* 2 pi radius))
> circumference
31.4
>
```

This **means of abstraction** allows one to reuse complex expressions without having to remember or repeat the details of the expressions. Programs are built by this simple tool, items are constructed and reused in other items. By repeating this process this encompasses an idea in engineering called **black box design**. A name however is just a symbol, and Scheme has memory which remembers that when you define a new name it tracks that the given symbol have the defined value. This memory will be referred to as the **environment**.

1.4 Exercises

1.4.1 Convert the following prefix expressions to the infix equivalent, and what would be returned by Scheme when it is evaluated:

a) `(+ 5 4 3)`
b) `(+ 3 4 5)`
c) `(- 3 4 5)`
d) `(- 5 4 3)`
e) `(+ (* 5 8) (- 10 5))`
f) `(* (- (* 2 (/ 10 2) 3) 1) 5 2)`
g) `(+ (* 5 4 (/ 10 5) (- 3 2)) (- 100 (* 20 4)))`

1.4.2 Translate the following infix expressions to the prefix Scheme form, and give the correct return value of each expression:

a) `(12 + 3) / 5`
b) `(10 + 5 - (2 * 5)) + 10 / 2`
c) `10 + (3 - 2) * 5 + 4 / 2`
d) 10^3 `+ 10 / 5 * (2 - 5)`
e) `5 * (2 + 3) / 5`
f) `(5 - 3) * 5 + (10 / 5) - 2`
g) `12 + 3 / 5 + 10 * 2 - 4`

1.4.3 What are the 4 general rules for constructing complex expressions.

1.4.4 Describe and name how to combine multiple expressions into one expression.

1.4.5 What means of abstraction can be used to hide the complexity of expressions, give and example.

1.4.6 How many return values can an expression have. *Hint: read carefully*

1.4.7 Describe the difference between operators and operands.

1.4.8 Use define to define the following symbols and value pairs:

a) `Name: eulers-number` `Value: 2.71828`
b) `Name: length` `Value: 12`
c) `Name: width` `Value: 40`
d) `Name: area` `Value: length * width`
e) `Name: height` `Value: 8`
f) `Name: volume` `Value: area * height`

1.4.9 Evaluate each of the named items defined in question **1.4.8** and give the value that Scheme returns for each.

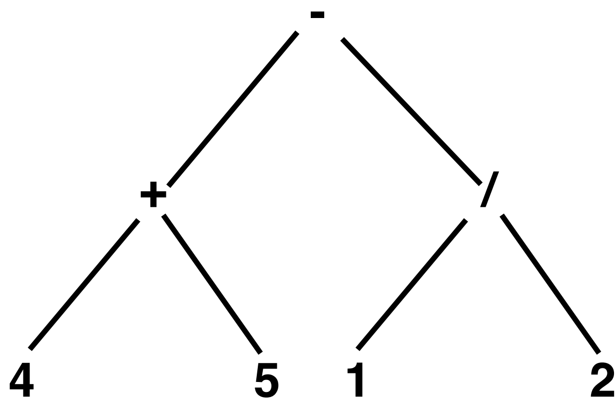
1.5 Substitution Model

The **substitution model** was informally introduced earlier, as a means for understanding the evaluation of compound (nested) expressions. We will formalized it here and offer several in-depth examples to clarify this concept - which is one of the most important features to understand in Scheme.

Scheme's main mechanism for problem solving is to break items and expressions down and substitute them with their solutions in a hierarchical model. Which eventually results in the answer to the problem presented to Scheme. Lets look at the example from earlier:

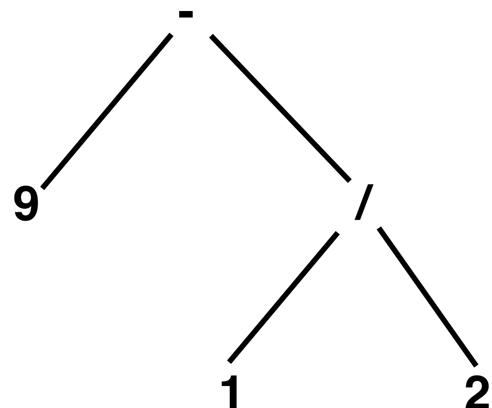
```
(- (+ 4 5) (/ 1 2))
```

This is actually a tree structure, you should try and visualize this in the following manor:

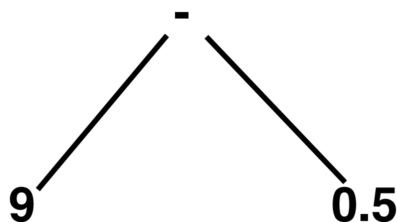


This tree structure to the left is the correct way to think of this compound expression. The primitive elements 4,5,1,2 are considered leaves of the tree. Evaluation occurs from the bottom most leaves and progresses up the tree. The first substitution in this example is the next tree.

Notice that the '**+**' operator is replaced or substituted with the result of applying the add procedure to its argument values 4 and 5.



Now if you were to try and evaluate the subtraction procedure specified at the root of the tree you have a problem. What are you supposed to subtract from 9? So in turn the division procedure needs to be evaluated first, before the subtraction can be executed.



Now you should be able to preform the subtraction procedure and you are simple left with 8.5. Since there is no more substitutions left the return value will be 8.5. This is a simple but powerful concept that we will build on if future chapters, however it is imperative you are able to deduce a tree structure from a given expression and in the correct order preform the substitution model.

We will end with this final example of the substitution model, please reflect on it until you understand it in its entirety.

Example 1.3

```
> (+ (/ 15 5) (* 2 (- 10 2) 1) (+ 2 (* 1 2 3)))
27
>
```

Substitution steps for **Example 1.3**

```
> (+ (/ 15 5) (* 2 (- 10 2) 1) (+ 2 (* 1 2 3)))
> (+ (/ 15 5) (* 2 8 1) (+ 2 (* 1 2 3)))
> (+ (/ 15 5) (* 2 8 1) (+ 2 6))
> (+ 3 (* 2 8 1) (+ 2 6))
> (+ 3 16 (+ 2 6))
> (+ 3 16 8)
> 27
```

The green highlighted text indicated the expression currently being substituted, and the bold black text indicated the substitution which was made for the previously highlighted green text.

1.6 Exercises

1.6.1 Show the detailed substitution method using illustrations of the trees for each step of the process for the below compound expressions:

a) `(/ (+ 4 4) (- 8 4))`
b) `(* (+ 1 2) (- 3 4) (+ 1 2 (* 1 2)))`
c) `(+ (/ (* 10 2) 4) 5)`
d) `(- (+ 10 (/ 5 5)) (* 3 4))`
e) `(+ (- 5 (/ (* 3 5) 2)) (/ 3 2) 2)`
f) `(+ (/ 15 5) (* 2 (- 10 2) 1) (+ 2 (* 1 2 3)))`

1.6.2 Give all the substitution steps 1 line at a time as in **Example 1.3** for question **1.6.1** parts **a** through **e**.

1.6.3 Rewrite the expression listed in question **1.6.1** parts **a** through **f** using the '**Pretty Printing**' concepts of newlines and indentation.

1.6.4 What symbol indicates a start of a comment in Scheme.

1.6.5 Why might the programmer want to use '**Pretty Printing**' and comments in their code.

1.6.6 Enter in `(1 2)` into the interactions window and ask Scheme to evaluate it. What does it say, and what does it mean (why is it saying what it does).

1.6.7 Enter in `(+ - * /)` into the interactions window and ask Scheme to evaluate it. What does it say, and what does it mean (why is it saying what it does).

1.6.8 Enter in `(define + -)` into the interactions window and ask Scheme to evaluate it. What does it say, and what does it mean (why is it saying what it does).

1.6.9 Enter in `(+ - * /)` into the interactions window and ask Scheme to evaluate it. What does it say, and what does it mean (why is it saying what it does).

1.6.10 Enter in `(define add +)` into the interactions window and ask Scheme to evaluate it. What does it say, and what does it mean (why is it saying what it does).

1.6.11 Immediately after preforming what is asked in question **1.6.10**.Enter in `(add 1 2)` into the interactions window and ask Scheme to evaluate it. What does it say, and what does it mean (why is it saying what it does).