

Principles of Programming Languages

Chapter 2: Syntax and Semantics with Scheme

In this chapter, we define a small but complete programming language. We demonstrate what are the elements necessary to define a programming language. We first present these elements informally - to define a subset of the Scheme language. This language is a simple functional language.

We then move on to define the elements formally in the form of:

- The syntax of the language (defining the set of expressions which belong to the language)
- The operational semantics of the language (defining how to map any expression in the language to a value in a recursive manner).

We implement these formal definitions into concrete programs - which together form an *interpreter* of the language.

This chapter brings us face to face with the most fundamental idea in computer programming: The interpreter for a computer language is just another program.

Interpreters are interesting because:

- They clarify *what programs do when they are executed*
- They illustrate how to build a wide class of programs which transform complex data from one form into another based on syntactic structure.

We use interpreters to implement the formal definitions of a programming language.

Object Language vs. Meta Language

The interpreter itself is written in a programming language, and it defines a programming language. To distinguish between these two programming roles, we use the following terminology:

- The language we define and describe is called the *object language*.
- The language we use to implement the interpreter is called the *meta language*.

We introduce a subset of Scheme to illustrate how to specify a programming language and how to implement a full interpreter because Scheme is a *small language* and a *simple language*. Yet, Scheme is an *expressive language* - it is Turing complete, and because it follows the functional programming paradigm, it allows the definition of powerful functional abstractions which make programming productive.

All of these elements stand in contrast with JavaScript - which has evolved into a *large language* with many primitives, a very complex syntax, and because it is a multi-paradigm language - supporting FP, OOP, procedural programming and more - JavaScript has a complex evaluation semantics.

We thus select to describe a *small and simple but full* language - a subset of Scheme - as the object language of this course.

We select to use a subset of TypeScript used as a Functional Language to implement the interpreter.

One of the advantages of this decision is that we exploit the TypeScript type system to encode the objects manipulated in the interpreter - abstract syntax trees and values. If the algorithm of the operational semantics is well understood, it can be implemented as a pure functional model in a code that is surprisingly short and elegant - and ported to any functional language.

Let us engage thus in developing a programming language bottom up, starting from small blocks and building up into a fuller version of a functional programming language.

How do we specify a programming language?
What are the elements that together define a programming language?

The key elements are:

1. **Primitives:** expressions whose evaluation is built-in in the interpreter and which are not explained by the semantics of the language. These include primitive operations (for example, arithmetic operations on number or comparison operators) and primitive literal values (for examples, numbers or boolean values).

2. **Combination means:** ways to create compound expressions and compound values from simpler ones.

3. **Abstraction means:** ways to manipulate compound objects (expressions or data values) as standalone units by giving them simple names.

The language definition is structured in two aspects:

1. **Expressions** are the words and sentences of the language.
2. **Values** are the results of the computation of expressions according to the evaluation rules of the language. Values belong to the semantic domain of the language.

To describe our object programming language, we first present all the types of expressions in the language (this is called the **syntax** of the language) on the one hand, and all the possible values that can be computed by the language on the other. The **syntax** is the input to the interpreter, the **values** are the output of the interpreter.

We will introduce the syntax and semantics of the Scheme subset in several iterations - starting from the simplest forms of expressions, then describing the rules to evaluate their value, then introducing more complex forms of expressions and their evaluation rules.

We distinguish:

- **Atomic** expressions
- **Compound** expressions

The following are the types of atomic expressions in Scheme:

- Literal numbers – 1, -1, 1.2
- Literal booleans – #t, #f
- Primitive procedures – +, -, *, /, >, <, =

Compound Expressions

The only syntactic form used to combine expressions into complex expressions in Scheme is to arrange them into parentheses:

(+ 45 78) ; ; => 123

(- 56 9) ; ; => 47

(* 6 50) ; ; => 300

(/ 25 5) ; ; => 5

These expressions are called **forms**. Their structure is to always refer to the leftmost sub-expression of the form as an *operator* and the rest of the sub-expressions as *operands*. Scheme compound expressions are always written in **prefix notation**.

Forms can be nested recursively:

```
( + ( * 3 15 )  
    ( - 9 2 ) ) ; ; => 52
```

Expressions are evaluated by the interpreter and return a value. In Scheme, there are only expressions in the language. This is in contrast to JavaScript which contains expressions and **statements** (syntactic elements which, when evaluated, do not return a value, but simply execute a command).

The programming language provides means to **name objects**. This is a fundamental form of **abstraction**: using a simple name instead of using a complex value or a complex expression.

In Scheme, **define** is used to bind a name the value of an expression.

```
(define size 6)  
(* 2 size) ;; => 12
```

size in this context is called a **variable**. The relation between a variable and the value it denotes is called a **binding**.

Variables can be used as atomic expressions. They are evaluated to the value to which they were bound using **define**.

define is a form of *abstraction* because it allows the programmer to use names (variables) instead of complex operations.

In the syntax of a **define** operation, **define** is a **special operator** - it indicates that a special operation must be performed by the interpreter to evaluate the

(define <var> <expression>)
form.

The result of evaluating a **define** form is that the interpreter remembers that the variable is now bound to a value.

To evaluate the form
(define <var> <exp>):

1. Let `val` = Evaluate(<exp>)
2. Add the binding < <var>, val > to the global environment

The global environment is a function which maps variable names to values.

We have now presented different expression types with the corresponding evaluation rules for each type of expression. Let us call this language $L1$ and summarize the rules to evaluate all of the expression types in $L1$. We also summarize the set of all possible values computed by $L1$.

All the expression types presented so far are:

- number literal expression – `0`, `1`, `2`, ...
- boolean literal expression – `#t` and `#f`
- primitive operations expressions – `+`, `-`, ...
- variable expressions – `x`, `area`, ...

- Special compound expressions:
`(define <var> <exp>)`
- Non-special compound expressions:
`(exp0 ... expn)` where each `expi` is any expression that is not `define`.

This inductive definition corresponds to the set *Expression* of all possible expressions in the language. (The definition is inductive because we use the term *expression* to define what are compound expressions.)

We define the function
 $evaluate : Expression \rightarrow Value$ in an inductive
manner:

1. Variables are evaluated by looking up their value in the global environment.
2. Primitive atomic expressions evaluate to their pre-defined denoted value.

For each special form, a special evaluation rule exists. The special form
`(define <var> <exp>)`
is evaluated according to this rule:

1. Let `val = evaluate(<exp>)`
2. Add the binding `< <var>, val >` to the global environment.
3. The form returns a special `void` value.

Note that in this rule, the sub-expression `<var>` is **not** evaluated. We have only introduced one special form **define** so far - we will introduce more later.

Evaluation of Compound Non-special Forms

All compound forms are of the form
`(exp0 ... expn)`.

1. Let:

`val0 = evaluate(exp0)`

...

`valn = evaluate(expn)`

2. Apply the procedure `val0` to the values
`(val1 ... valn)`.

Looking at all the evaluation rules, we can summarize the set of all possible values that can be returned by an invocation of evaluate:

- Number values
- Boolean values
- Primitive operations

Example Programs in L1

5	:: => 5
(* 3 2)	:: => 6
(+ (* 3 2) 4)	:: => 10
(> 2 3)	:: => #f
(= 2 (+ 1 1))	:: => #t

Note that expressions in general are evaluated one by one. The order in which expressions are evaluated does not change their value.

Only in the case of the **define** form, there is a side-effect which makes the sequence of expressions significant:

```
(define radius 12)
(define pi 3.14)
(define area (* (* radius radius) pi))
(+ area (* 2 3)) ;; => 458.16
```

In order to make sense of a program that includes **define** forms, we must define a compound expression which is a sequence of expressions and its evaluation rule. We will describe the details of the evaluation rule for sequences including define-expressions in more details later.

Let us try to assess what programs can be written in $L1$ as defined.

On the side of the restrictions - we have:

- Few primitives - and no way to define other functions besides primitive functions (no *functional abstraction means*).
- The computed values can only be numbers or booleans - there are no way to build compound values (no *value composition means*).

- We can define global variables and no scoping mechanism
- There are no control structures

On the positive side:

- We can build expressions as deeply nested as required
- We can give names to complex expressions so that they can be reused to avoid repetition

Programs in $L1$ **always**
terminate.

L2: User Defined Procedures and Conditional Expressions

Let us introduce two new types of expression into *L1* - leading to a new language we will call *L2*. We choose to add together user defined procedures and conditional expressions - because these two language facilities *work well together*. The reason is that we will develop recursive functions - and when we write a recursive function, it helps to be able to test for the base case vs. recursive case.

`lambda` is a special operator which can be used in a special form of type `lambda`. The syntax is:

```
(lambda (<var> ...) <exp> ...)
```

For example:

```
(lambda (x) (* x x))
```

This expression is a procedure expression. It has three sub-expressions:

- The special operator `lambda`
- The list of parameters - all of which are variables.
- The body of the procedure - which is a list of expressions.

When this expression is evaluated, it creates a value whose type is called a **closure**. We will denote such values as
`<Closure (x) (* x x)>`.

Closures: Composite or Atomic Values?

A closure value contains multiple parts - the parameters and the body. But there are no accessors to take apart these components from the value. This leads to an interesting distinction:

- From the programmer perspective, a closure is an atomic value.
- From the interpreter perspective, a closure is a compound data structure with accessible sub-components.

When a `lambda` expression is evaluated, the body is **not** evaluated.

To give a name to a procedure, we use the existing `define` mechanism:

```
(define square (lambda (x) (* x x)))
```

We will see later in the course that the capability to name procedures is a *big deal* - as it allows the definition of recursive functions - and, in particular, it changes the expressive power of the programming language.

When a `lambda` expression is computed, we obtain a closure. Closures can then be applied to values. How?

The way a closure

`val0 = <Closure (p1 ... pn) <exp1> ... <expk>>`

is applied to values

`(val1 ... valn)`

is according to the following rule:

Compound Procedure Application

1. Replace all occurrences of p_1, \dots, p_n in the expressions $\langle \text{exp}_1 \rangle, \dots, \langle \text{exp}_k \rangle$ of the body of the procedure with the corresponding values $\text{val}_1, \dots, \text{val}_n$.
2. Evaluate all resulting expressions.
3. The value returned by the application is the value of the last expression $\langle \text{exp}_k \rangle$.

We introduce a third special form to the syntax of the language, in addition to **define** and **lambda** together with its specific evaluation rule to enable conditional evaluation.

The syntax of a Scheme conditional expression is:

```
(if <exp> <exp> <exp>)
```

For example:

```
(if (> x 2) x (* x 2))
```

`if` is a special operator - it has a special evaluation rule. The three other sub-expressions are called the test-part, then-part, and else-part of the compound if-expression. They can recursively be any type of expression.

Example: abs

```
(define abs  
  (lambda (x)  
    (if (> x 0) x (- x)))))
```

if-expressions can be nested as needed to define complicated decisions:

```
(if (= x y)
    0
    (if (> x y)
        1
        -1))
```

Conditional Expressions

Scheme also includes an additional special form called **cond** which allows a more general form of conditional expression:

```
(cond (<p1> <e11> ... <e1k1>)  
      (<p2> <e21> ... <e2k2>)  
      ...  
      (else <en1> ... <enkn>))
```

The sub-expressions of the **cond** form are called *clauses* - each clause starts with a predicate-expression and is followed by consequence-expressions.

Evaluation Rule for If-expressions

To evaluate an if-expression
(if <test-exp> <then-exp> <else-exp>):

1. Let $p = \text{evaluate}(\text{<test-exp>})$
2. If p is true, return
 $\text{evaluate}(\text{<then-exp>})$
3. Otherwise, return
 $\text{evaluate}(\text{<else-exp>})$

NOTE: When evaluating an if-expression, the `<test-exp>` is always evaluated, but only one of `<then-exp>` or `<else-exp>` is evaluated. This is in contrast to what happens when we evaluate a non-special form - where all the sub-expressions are always evaluated.

The language we have defined so far is quite expressive. Let us define an example program demonstrating this: this program implements Newton's method for computing square roots.

Newton's method is stated as this algorithm:

If y is a non-zero guess for \sqrt{x} , then $\frac{y + \frac{x}{y}}{2}$ is a better approximation of \sqrt{x} .

To start this computation, we provide a non-zero guess like 1, and we need to decide when to stop guessing.

This algorithm is iterative:

- Is the current guess good enough? if yes return it.
- Otherwise, improve the guess and try again.

Interestingly - we can implement this iterative algorithm even though we have no construct in the language to iterate.

```
(define sqrt  
  (lambda (x) (sqrt-iter 1 x)))
```

```
(define sqrt-iter  
  (lambda (guess x)  
    (if (good-enough? guess x)  
        guess  
        (sqrt-iter (improve guess x)  
                    x)))))
```

Example Program in L2

```
(define abs (lambda (x) (if (< x 0) (- x) x)))  
(define square (lambda (x) (* x x)))  
(define epsilon 0.0001)  
  
(define good-enough?  
  (lambda (guess x)  
    (< (abs (- (square guess) x)) epsilon)))  
  
(define average  
  (lambda (x y) (/ (+ x y) 2)))  
  
(define improve  
  (lambda (guess x)  
    (average guess (/ x guess))))
```

This program illustrates many of the “good properties” we associated with the Procedural Programming paradigm in Chapter 1:

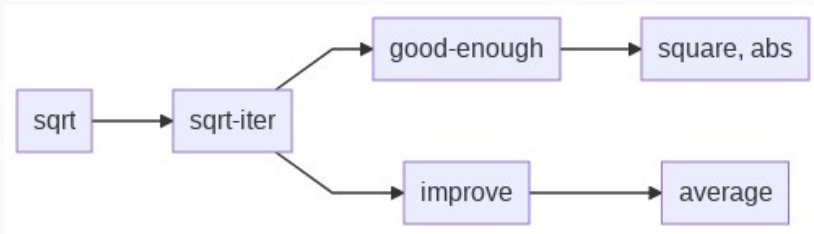
Example Program in L2

- Encourage the use of small units of codes, called procedures, which encapsulate well-defined commands.
- Procedures interact through well-defined interfaces published by each procedure (the contract of the procedure, including the signature of which arguments it expects, and which return value it returns).

We haven't discussed local variables (used inside each procedure without affecting the state of the program outside the scope of the procedures). We will see later in the chapter that even in L_2 , we have enough *semantic power* to define local variables, but we have not provided *syntactic constructs* to encourage the use of this feature.

Example Program in L_2

We have created a hierarchy of procedures, higher-level procedures call lower-level procedures:



So what is new in *L2*?

- We introduced the **lambda** expression that evaluates into the **closure** value
- We know how to apply a closure to a list of arguments
- We have if-expressions

Note the programming constructs which are absent from L2:

- No loop structures
- No mutation of variables
- No compound data structures
- No local variables

Can you prove that some programs in $L2$ may not terminate?

```
(define f (lambda (x) (f x)))
```

We observe that the computed values of $L2$ are atomic values (numbers, booleans) or closures (which are a compound value but which has no accessors).

To introduce compound data values in the language, we need:

- Constructors for compound values
- Literal expressions that denote compound values

In JavaScript, for example, compound values are constructed with the array and map constructors and denoted by the `[]` and `{}` notations.

In the minimalist spirit we have adopted so far, we will introduce into *L3* a single compound value constructor and the capability to use it recursively.

The Pair Compound Data Type

A pair is a minimal compound data type that combines two values together into a single new unit. The language supports this by introducing:

- A value constructor: this is called **cons**.
- Accessors to take apart a compound pair value: these are called **car** and **cdr**.
- A type predicate to check whether a value belongs to the set of pair values: **pair?**.
- An equality predicate for pairs **equal?**.

We thus extend the language with 5 primitive functions: `cons`, `car`, `cdr`, `pair?`, `equal?`.

The Pair Compound Data Type

In addition, Scheme defines a standard form to print pair values and literal expressions that denote pair values: A literal pair expression is denoted as '`<exp> . <exp>`'.

For example:

```
(define p1 '(1 . 5))    ;; literal pair expression  
(define p2 (cons 1 5))  ;; constructor invocation
```

```
(car p1) ;; => 1  
(cdr p1) ;; => 5
```

The Pair Compound Data Type

Pairs can be combined recursively into complex compound values:

```
(define p3 (cons p1 p2))  
(define p4 (cons p3 p2))
```

```
(car p3) ;; => '(1 . 5)  
(cdr p3) ;; => '(1 . 5)
```

The `cons` pair constructor can receive parameters of any type. The type of `cons` is thus described as:

`<T1, T2>(first: T1, second: T2) => Pair<T1, T2>`

The List Compound Data Type

In addition to the Pair data type, L3 introduces a recursive data type - called List. We first introduce it inductively:

- The *empty list* denoted '()' is the base case.
- A non-empty list is built by combining a value v_0 together with a list $(v_1 \dots v_n)$ to obtain a non-empty list $(v_0 \ v_1 \dots v_n)$ - which combines a *head* (v_0) and a *tail* which is a list.

Non-empty lists are constructed from a value and a list. The size that characterizes lists in this inductive definition is the length of the list: a list of size $n + 1$ is constructed from a value and a list of size n .

The definition of this inductive data type is a **disjoint union** between the empty list and non-empty lists.

Scheme implements List values by re-using the Pair data type for non-empty lists and a special value for the empty list. The additions to the language are:

The List Compound Data Type

- A new primitive value `'()` and a corresponding predicate `empty?`
- A special type of literal compound expressions for list values: `'(e1 ... en)`
- The predicate primitive `list?` to check that an object belongs to the List data type (either empty or non-empty).

The List Compound Data Type

```
(define l14  
  (cons 1 (cons 2 (cons 3 (cons 4 '())))))
```

```
l14                ;; => '(1 2 3 4)  
(list? l14)        ;; => #t  
(cdr l14)          ;; => '(2 3 4)  
(car (cdr l14))    ;; => 2  
(cons 10 l14)      ;; => '(10 1 2 3 4)
```

The List Compound Data Type

On the basis of this inductive definition, we can define functions over lists:

```
(define length
  (lambda (lst)
    (if (empty? lst)
        0
        (+ 1 (length (cdr lst))))))
```

As usual, recursive functions operating over an inductive type have a structure similar to the inductive definition of the inductive compound data type: because List is a disjoint union over Empty and Non-Empty lists, the structure of a function operating over lists will be:

The List Compound Data Type

On the basis of this inductive definition, we can define functions over lists:

```
(define <f>
  (lambda (lst)
    (if (empty? lst)
        <process empty list case>
        <process non-empty list case>)))
```

The List Compound Data Type

For example:

```
(define nth
  (lambda (lst n)
    (if (empty? lst)
        '()
        (if (= n 0)
            (car lst)
            (nth (cdr lst) (- n 1))))))
```

```
(nth '(1 2 3) 2) ;; => 3
```

```
(nth '(1 2) 2)   ;; => '()
```

The list constructor can also receive parameters of any types. In particular, we can be create list of pairs and recursive tree structures using the compound list data type.

Another list constructor is available - which avoids the need for nested calls to **cons**:

`(list <e1> ... <en>)`

evaluates to

`'(v1 ... vn)`

where **v_i** is the value of **<e_i>**.

So what is new in *L3*?

- New primitive procedures for handling pairs and lists:

`cons, car, cdr, pair?, list?`

- New atomic literal expression for the empty list: `'()`

- Literal compound expressions:
 - '(<lit> . <lit>)' for pairs,
 - '(<lit> ... <lit>)' for lists.

- Pairs
- Lists