Introduction to Lisp and Emacs

CMPU-365, Fall 2008, Luke Hunsberger

1 The World of Lisp

Programmers always need to have an accurate mental model of the operation of whatever computer they are programming. The complexity of their mental model depends in large part on the kind of programming language they are using. One of the significant advantages of the Lisp programming language is that it is based on a fairly simple computational model. This section introduces what I call the *abstract World of Lisp,* which serves as a context for describing Lisp's model of computation. This model is based on the Lambda Calculus invented by Alonzo Church in the 1930s. Internalizing Lisp's model of computation will make you an effective Lisp programmer in no time!

The World of Lisp is populated by Lisp entities. Some of these entities are *atomic* (or *primitive*)—in the sense that any internal structure they might have is not available to us for inspection. For example, in the World of Lisp, numbers and basic arithmetic functions are atomic entities. However, the range of atomic entities in the World of Lisp is fairly limited.

1.1 Primitive Lisp Entities

The World of Lisp is inhabited by Lisp entities, both primitive and compound. This section focuses on primitive Lisp entities, such as numbers, symbols and primitive functions. It is important to keep in mind that each Lisp entity has a unique type. For example, a Lisp entity cannot be both a number *and* a symbol. In other words, the World of Lisp entities is *partitioned* according to type.

1.1.1 Numbers (Primitive Lisp Entities)

Each number (e.g., the number *three*) is a primitive Lisp entity.

1.1.2 Symbols (Primitive Lisp Entities)

The World of Lisp includes a wide variety of *symbols*, including, for example, the symbols `xyz`, `abcdef`, `*` and `p_qr_s`. In Lisp, there is no *boolean* data type. Instead, as we'll see later on, the symbols `t` and `nil` play the roles of *true* and *false*, respectively. In addition, the symbol `nil` also plays the role of the *empty list* (i.e., a list containing no items).

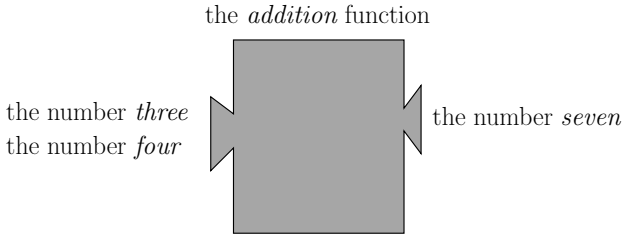
1.1.3 Primitive Functions (Primitive Lisp Entities)

Primitive functions (or procedures) are those whose internal structure is invisible to us.[1] The World of Lisp includes a basic toolbox of primitive functions such as the *addition*, *subtraction*, *multiplication* and *division* functions. Another primitive function is the extremely important *evaluation* function, which will be described later on.

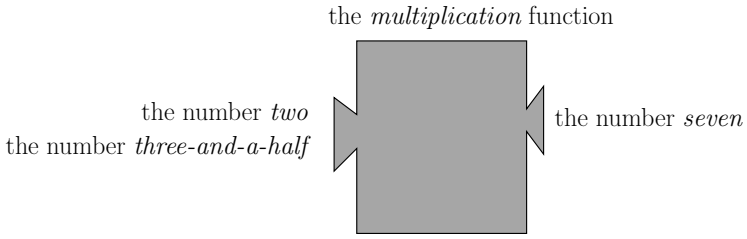
1.2 Function Application

Life would be pretty dull in the World of Lisp if nothing ever happened. Fortunately, there is one (and only one) way in which things happen: a function gets applied to some input arguments and generates a result. It is extremely important to keep in mind that the function, input arguments, and result are all Lisp entities. For example, the *addition* function is a Lisp entity that, when applied to the numbers *three* and *four* (which are also Lisp entities), generates the number *seven* (which is also a Lisp entity), as illustrated below.

[1] We shall use the words, function and procedure, interchangeably. Thus, the title of this section could just as well have been *Primitive Procedures*.

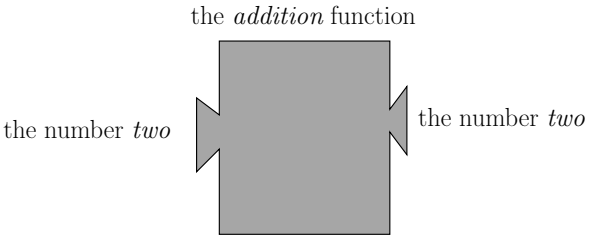


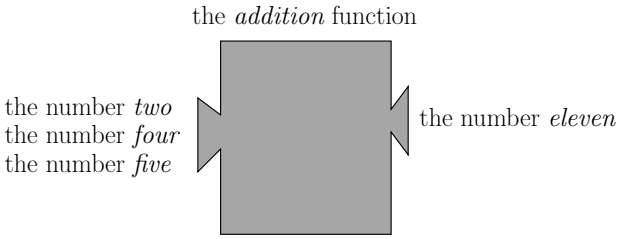
Notice that the *addition* function is shown as a *black box* whose internal structure is hidden from us. Similarly, the application of the *multiplication* function to the numbers *two* and *three-and-a-half* is illustrated below.



As always, the function, input arguments and output are all Lisp entities.

The *addition* function is only defined for numerical inputs. Thus, we are not allowed to apply the *addition* function to arguments of any other type (e.g., symbols or functions). However, the *addition* function can be applied to any number of numerical arguments, as illustrated below.





The *addition* function can even be applied to *zero* arguments—in which case, the output is the number *zero* (i.e., the additive identity).

Similarly, the *multiplication* function can be applied to any number of arguments. When applied to zero arguments, the *multiplication* function outputs the number *one* (i.e., the multiplicative identity).

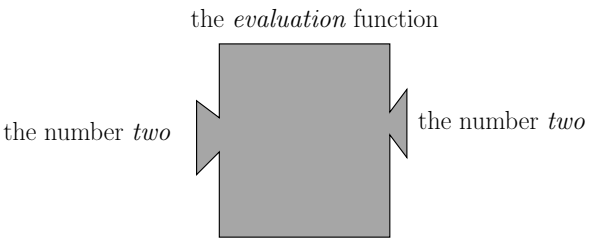
1.3 The *Evaluation* Function (Primitive Lisp Entity)

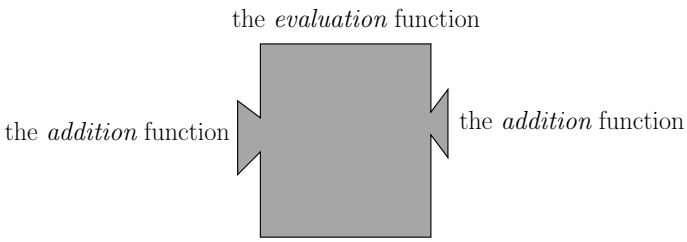
One of the primitive Lisp entities, the *evaluation* function, is so important that we dedicate an entire section to describing it. Like all primitive functions in the World of Lisp, *how* the *evaluation* function does what it does is hidden from us; however, we still need to know *what* it does.

The *evaluation* function takes a single Lisp entity as its only input. The result of applying the *evaluation* function depends on the type of Lisp entity that it is applied to. As with any function application, the result is a single Lisp entity.

1.3.1 Applying the *Evaluation* Function to Numbers or Functions

The *evaluation* function acts like the identity function when applied to numbers or functions, as illustrated below.





Since drawing all of these black boxes takes up so much space, from now on we'll use a simpler, text-based notation for the application of the *evaluation* function, as illustrated below.

$$\text{[Input Entity]} \quad \Longrightarrow \quad \text{[Output Entity]}$$

The double arrow ($\Longrightarrow$) is reserved solely for representing the application of the *evaluation* function to some Lisp entity (called the input) to generate some, possibly quite different Lisp entity (called the output).

Instead of saying that the *evaluation* function generates the output entity when applied to a certain input entity, we may say that the output entity is the result of *evaluating* the input entity (or that the input entity *evaluates* to the output entity). Keep in mind that when we say such things, we are talking about the application of the one-and-only primitive *evaluation* function.

Here are some more examples illustrating the trivial behavior of the *evaluation* function when applied to numbers or functions:

$$\begin{array}{rcl} \text{the number } \textit{zero} & \Longrightarrow & \text{the number } \textit{zero} \\ \text{the } \textit{multiplication} \text{ function} & \Longrightarrow & \text{the } \textit{multiplication} \text{ function} \end{array}$$

Since the *evaluation* function is itself a Lisp entity, there is nothing stopping us from applying the *evaluation* function to itself:

$$\text{the } \textit{evaluation} \text{ function} \quad \Longrightarrow \quad \text{the } \textit{evaluation} \text{ function}$$

Of course, if the *evaluation* function acted like the identity function for *every* kind of input, then it would not be very interesting. (It would just be the identity function.) The following sections describe two important special cases: symbols and non-empty lists.

1.3.2 Applying the *Evaluation* Function to Symbols

In Lisp, symbols are used to represent *variables*. In Math, variables frequently have values associated with them. For example, the variable x may have the value 3. So it is with Lisp. Thus, the evaluation of symbols is different from the evaluation of numbers or functions.

The evaluation of a symbol is based on *table lookup*. In particular, the *evaluation* function may be thought of as having a private table called the *Global Environment*.[2] The Global Environment contains a bunch of entries. Each entry pairs a symbol (which is a Lisp entity) with its corresponding value (which also is a Lisp entity). To evaluate a symbol, the *evaluation* function simply looks up the value associated with that symbol in the Global Environment.

For example, if the Global Environment contains an entry that associates the number *two* with the symbol `xyz`, then the result of applying the *evaluation* function to the symbol `xyz` will be the number *two*:

$$\text{the symbol } \texttt{xyz} \quad \Longrightarrow \quad \text{the number } \textit{two}$$

In this case, we would say that the *value* of the symbol `xyz` is the number *two*.

The value associated with a symbol in the Global Environment can be *any* type of Lisp entity—even compound Lisp entities that we have not yet encountered. Thus, the Global Environment might contain entries stipulating that the value of the symbol `pq` is the *addition* function; or that the value of the symbol `jk` is the symbol `nil`, which we would notate as follows:

$$\begin{array}{rcl} \text{the symbol } \texttt{pq} & \Longrightarrow & \text{the } \textit{addition} \text{ function} \\ \text{the symbol } \texttt{jk} & \Longrightarrow & \text{the symbol } \texttt{nil} \end{array}$$

If a symbol does not have a corresponding entry in the Global Environment, then the value of that symbol is undefined. Stated differently, the result of applying the *evaluation* function to that symbol would be undefined.[3]

Later on, we'll see how to add new entries to the Global Environment or modify existing entries. Doing so will enable us to use symbols as variables: adding a new entry is how we will *declare* a new variable; modifying an existing entry is how we will *assign* a value to a previously declared variable.

However, the Global Environment is typically initialized to contain a certain core set of entries. For example, the Global Environment *always* contains entries such that the following evaluations hold:

[2] Since the Global Environment is a private appendage of the *evaluation* function, it is not an official Lisp entity and, thus, is not available for direct inspection.

[3] In an actual Lisp session, attempting to evaluate such a symbol would cause an error.

the symbol `t` $\Longrightarrow$ the symbol `t`
the symbol `nil` $\Longrightarrow$ the symbol `nil`

Observe that these entries imply that the symbols `t` and `nil` evaluate to themselves. In addition, the Global Environment typically contains entries specifying values for symbols such as the following:

```
most-positive-fixnum

most-positive-double-float

*standard-input*

*standard-output*
```

The values for these symbols indicate certain features of the host computer.

Remember: The Global Environment is *not* an official Lisp entity! Instead, you should think of it as a little black book that the evaluation function uses to keep track of values associated with symbols.

1.4 Non-primitive Lisp Entities: Non-empty Lists

A non-empty list is a compound (i.e., non-primitive) Lisp entity. In particular, a non-empty list is an ordered sequence of Lisp entities. The following are valid non-empty lists in the World of Lisp:

- a list containing the number *three* and the number *four*
- a list containing the `+` symbol, the number *three*, and the number *four*
- a list containing two entities: (1) the symbol `eval`, and (2) a list containing the `+` symbol, the number *three* and the number *four*

The last example illustrates that a list can contain entities that are themselves lists.

Below, we distinguish three cases of non-empty lists:

(1) Default Case: A list whose first element is a symbol—but *not* one of the Lisp *keywords.*

(2) Special Form: A list whose first element is one of the Lisp *keyword* symbols (e.g., `defparameter`, `quote`, `lambda`, `let`, `if` or `cond`).

(3) A list whose first element is not a symbol.

The evaluation of lists belonging to the *default case* involves the application of a function to zero or more arguments—where the function and each of the arguments are Lisp entities. The evaluation of such lists is discussed in Section 1.5.

Lists whose first element is one of Lisp's keyword symbols are called *special forms.* The evaluation of special forms is considered in later sections, beginning with Section 3.

Most lists belonging to case (3) cannot be evaluated. There is at least one major exception, but even that is not widely used. Thus, it will be mentioned only briefly in a later section.

1.5 Applying the *Evaluation* Function to Non-Empty Lists: the Default Case

This section addresses the evaluation of lists whose first element is a symbol—but not one of Lisp's keyword symbols. Describing how such lists are evaluated requires saying a bit more about the Global Environment.

"Ordinary Values" and "Function Values"

Recall that, so far, the Global Environment has been thought of as containing entries, where each entry associates a value with a symbol. Well, it's time to reveal that each entry in the Global Environment really has an extra slot. In particular, each symbol in the Global Environment can also have a "function value" associated with it. As its name indicates, the entity serving as the "function value" for some symbol is required to be a Lisp function entity.[4]

[4]This contrasts sharply with "ordinary values" which can be any kind of Lisp entity (e.g., numbers, symbols or even functions).

⇒ To distinguish the two types of values that can be associated with a symbol, we shall call one the "ordinary value" and the other the "function value".

Previous sections have focused on the "ordinary value" associated with a symbol. In particular, when a symbol is fed as input into the evaluation function, the evaluation function generates as its output the "ordinary value" associated with that symbol in the Global Environment. For any symbol s, the shorthand notation, $s \Longrightarrow v$, is used to indicate that v is the "ordinary value" associated with s in the Global Environment.

⇒ If v is the "function value" associated with the symbol s in the Global Environment, then we shall use the following notation: $s \Longrightarrow_f v$, where the subscript f indicates that we are talking about the "function value", not the "ordinary value".

Suppose a *list* contains the symbol s as its first element. When that *list* is fed as input into the evaluation function, part of the process of evaluating that *list*—which will be described in detail below—involves fetching the "function value" associated with s in the Global Environment.

We shall assume that the Global Environment *always* contains a core set of entries that associate "function values" with certain commmonly used symbols. For example, it contains an entry stipulating that the `+` symbol has the *addition* function as its "function value" (i.e., that `+` $\Longrightarrow_f$ the *addition* function). Similarly:

the `*` symbol $\Longrightarrow_f$ the *multiplication* function

the `-` symbol $\Longrightarrow_f$ the *subtraction* function

the `eval` symbol $\Longrightarrow_f$ the *evaluation* function

For convenience, we say that `+` is the name of the *addition* function; `eval` is the name of the *evaluation* function; and so forth.

Evaluating Non-Empty Lists in the Default Case

Suppose $\mathcal{L}$ is a list that contains n Lisp entities, $e_1, e_2, \ldots e_n$, where the first entity, e_1, is a symbol—but *not* one of the Lisp keywords (i.e., $\mathcal{L}$ falls into the default case described earlier). Suppose further that the symbol, e_1, has a "function value", say, E_1, associated with it in the Global Environment (i.e., $e_1 \Longrightarrow_f E_1$). Note that E_1 must be a function entity. Then, the evaluation of the list $\mathcal{L}$ is derived as follows:

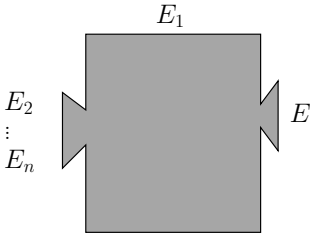
- First, evaluate the rest of the entities $e_2, \ldots, e_n$, resulting in new entities $E_2, \ldots, E_n$. If any of these entities are symbols, use their "ordinary values" when evaluating them.
- Next, *apply* the function E_1 to the entities $E_2, \ldots, E_n$. In other words, the entities $E_2, \ldots, E_n$ are the inputs used by the function E_1 to generate some output. Call that output entity E.

The value of the original list $\mathcal{L}$ is defined to be that entity E. In other words,

If $\mathcal{L}$ is a list containing the entities $e_1, e_2, \ldots, e_n$, where:

$e_1 \Longrightarrow_f E_1$;

$e_2 \Longrightarrow E_2$, ..., $e_n \Longrightarrow E_n$; and



then $\mathcal{L} \Longrightarrow E$.

Function position. If a symbol appears as the first entity of a list, it is treated specially by the evaluation function. In particular, the evaluation function fetches the "function value" associated with that symbol. In contrast, if a symbol appears as one of the other entities in a list, then the evaluation function fetches its "ordinary value". For this reason, the first position in a list is often called *function position.* If a symbol appears in function position, then the evaluation function fetches its "function value"; otherwise, the evaluation function fetches its "ordinary value".

Things that could go wrong. Many things could go wrong in the process of evaluating a non-empty list. For example, the first element of the list might be a symbol (e.g., `nil`) that does not have a "function value" associated with it. Or the function E_1 might expect a different number of arguments than are present in the rest of the list. Or the attempt to evaluate one of the entities $e_2, \ldots, e_n$ might be undefined (e.g., if one of them is a non-empty list falling into case (3) above). Or the application of the function E_1 to the arguments $E_2, \ldots, E_n$ might be undefined because, for example, the function expects numbers and it gets something else. In any of these cases, the result is undefined. Thus, none of the following lists can be evaluated:

a *list* containing the symbols `nil` and `t`

a *list* containing only the `eval` symbol

a *list* containing: (1) the `eval` symbol, and (2) a list containing the `eval` symbol

a *list* containing the `+` symbol, followed by the symbols `t` and `nil`

It is important to understand that each of the above lists is a valid Lisp entity: each one is a *list*. It's just that these lists are not valid inputs for the primitive *evaluation* function. That is, they cannot be evaluated.

Example. Here's an example of the default case of evaluating a non-empty list where things work out. Let $\mathcal{L}$ be the list containing the following entities:

e_1: the `+` symbol, e_2: the number *one*, e_3: the number *two*, e_4: the number *three*

The "function value" associated with the `+` symbol is the *addition* function. The rest of the Lisp entities are evaluated as follows:

$e_2 \Longrightarrow E_2$: the number *one*

$e_3 \Longrightarrow E_3$: the number *two*

$e_4 \Longrightarrow E_4$: the number *three*

Since, in this case, E_1 is the *addition* function, it can be applied to the arguments E_2, E_3 and E_4 (i.e., the numbers *one, two* and *three*). This results in the output *six*, which is itself a Lisp entity. The value of the original list $\mathcal{L}$ is thus the number *six*.

Notice that because the *addition* function is a primitive function, its operation is invisible to us. We observe the inputs going in and the output coming out, but we do not get to see *how* the output is generated.

One More Example. Let $\mathcal{L}$ be the list containing two entities:

e_1: the *eval* symbol, and

e_2: the `most-positive-fixnum` symbol

Since `eval` is a symbol, but not one of the Lisp keywords, this list belongs to the default case. Furthermore, the `eval` symbol's "function value", courtesy of the Global Environment, is the *evaluation* function. In addition, the second entity in the list is the `most-positive-fixnum` symbol, whose "ordinary value", again courtesy of the Global Environment, is the number *536870911*.

$e_1 \Longrightarrow_f E_1$: the *evaluation* function

$e_2 \Longrightarrow E_2$: the number *536870911*

Next, the entity E_1 (i.e., the *evaluation* function) is applied to the entity E_2 (i.e., the number *536870911*). Like all numbers, this number evaluates to itself. Thus, the result of evaluating the original list $\mathcal{L}$ is simply the number *536870911*.

1.6 Summary

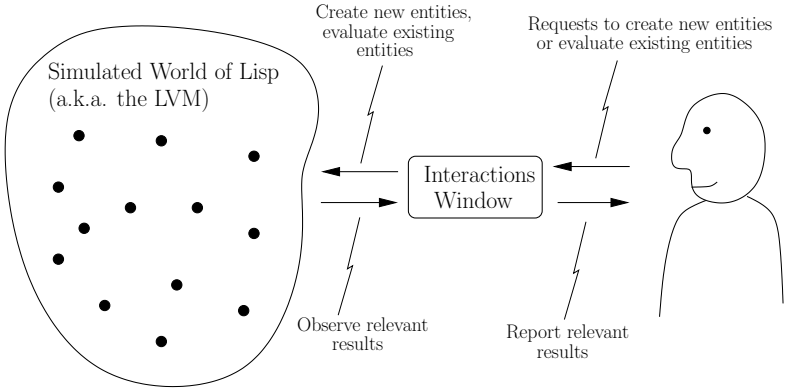
This section introduced the abstract World of Lisp. That world contains entities of various types (e.g., numbers, symbols, functions and lists). Entities of the function type can be *applied* to other Lisp entities. One of the primitive function entities, called the *evaluation* function, plays an important role in the World of Lisp. When applied to numbers or functions, the *evaluation* function acts like the identity function. However, when applied to symbols and non-empty lists, the *evaluation* function acts quite differently. For symbols, it uses a lookup table called the Global Environment. The evaluation of non-empty lists belonging to the *default case* involves fetching the "function value" of the symbol in function position and applying that function to the values of the rest of the elements in the list. The default case of evaluating non-empty lists is the primary mechanism whereby Lisp programmers can cause a function to be applied to a bunch of inputs.

2 Introduction to Emacs, Lisp and the Lisp Virtual Machine!

This section introduces the piece of software known as *Emacs*. Emacs originally started out as a simple text editing program; however, it has evolved amazingly over the years. Here, we are interested in Emacs's ability to use one of its window panes (or *buffers*) to act as a kind of intermediary between the programmer and a simulated world of Lisp.[5] For fun, we'll refer to the simulated world of Lisp as the Lisp Virtual Machine (LVM).[6] Because the Emacs/Lisp buffer enables the programmer to effectively *interact* with the LVM—albeit through an intermediary—we'll henceforth call that buffer the *Interactions Window* (IW).[7] It is important to keep in mind that we do not interact directly with the simulated World of Lisp (or the LVM); instead, we use the Interactions Window as an intermediary between us and that simulated world. In particular,

- by typing character sequences into the IW, we effectively ask the IW to make things happen in that simulated world (i.e., to make the LVM do something); and
- the IW reports back to us the results of the LVM's computations (e.g., by displaying a sequence of characters on the screen).

As illustrated below, only the Interactions Window gets to interact directly with the simulated world (i.e., with the LVM).



As a result, we never get to "touch" Lisp entities directly. For this reason, it is extremely important that we maintain an accurate mental model of what's going on in that simulated world. The rest of this section describes how we can effectively and reliably make things happen in that simulated world, using the Interactions Window as an intermediary.

2.1 Character Sequences are not Lisp Entities!

The first thing to note is that we interact with the IW by typing character sequences and by reading character sequences that are displayed on the screen. Thus, our interaction with the IW is solely in the realm of character sequences.

[5] Instructions for setting up Emacs with a buffer simulating the world of Lisp can be found on the cs365 web site.

[6] Recall that the simulated computer on which all Java programs are run is called the Java Virtual Machine (JVM).

[7] This term comes from drScheme's Interactions Window for the Scheme programming language. Scheme is quite similar to Lisp. However, Lisp came first (in the 1950s, as opposed to Scheme in the 1970s).

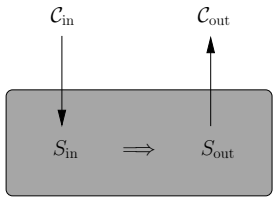
Typically, when we type a sequence of characters, $\mathcal{C}_{\text{in}}$, into the Interactions Window, and then hit the *Enter* key, and the following things happen:

(1) the character sequence is translated into a Lisp entity, S_{in};

(2) the evaluation function is applied to S_{in}, generating an new entity S_{out} (i.e., S_{in} *evaluates* to S_{out}); and

(3) that new Lisp entity, S_{out}, is translated into a character sequence $\mathcal{C}_{\text{out}}$, which is then displayed on the screen as the *result*.

This process is illustrated below.



Keep in mind that we only see the character sequences, $\mathcal{C}_{\text{in}}$ and $\mathcal{C}_{\text{out}}$; we do not see the Lisp entities, S_{in} and S_{out}. (What does a Lisp entity look like anyway?) We can abbreviate this process as follows:

$$\mathcal{C}_{\text{in}} \quad \longrightarrow \quad [\quad S_{\text{in}} \quad \Longrightarrow \quad S_{\text{out}} \quad] \quad \longrightarrow \quad \mathcal{C}_{\text{out}}$$

where the single arrow ($\longrightarrow$) represents the translation process (between character sequences and Lisp entities), the double arrow ($\Longrightarrow$) represents the application of the primitive *evaluation* function (as described in Section 1), and the square brackets indicate that we don't get to see S_{in} and S_{out}.

2.2 The Character Sequences Used by the LVM (and Us)

In our daily lives, we frequently use character sequences to refer to both concrete and abstract entities. For example, the character sequence `DOG` is frequently used to refer to a dog. Similarly, the character sequence `34` is used to refer to the number *thirty-four*. Of course, this article is itself a bunch of character sequences representing all sorts of things. Well, actually, it is a piece of paper with ink marks on it. The ink marks represent characters which, in turn, form sequences of characters that represent other things. The point is: we are so used to using character sequences to represent or refer to things that we tend to take it for granted. When using the IW (or, more generally, when programming in Lisp), the failure to be aware of the difference between character sequences and the Lisp entities they represent can lead to frustrating states of confusion.

Like any programming language, Lisp is defined by a set of *syntax* rules and a set of *semantic* rules. The syntax rules stipulate the legal constituents of a Lisp program. A Lisp program is a (usually long) sequence of characters; a legal constituent of a Lisp program is a (usually much shorter) sequence of characters. The basic building block of a Lisp program is an *expression*. The semantic rules stipulate the translation from Lisp expressions to Lisp entities. In addition, the semantic rules define the evaluation function.

In a typical interaction, the programmer types a character sequence into the IW. The IW then does the following:

- Checks that the character sequence is a legal Lisp expression according to syntax rules
- Translates that character sequence into the corresponding Lisp entity according to the semantic rules
- Applies the evaluation function to that Lisp entity generating a new Lisp entity
- Translates that new Lisp entity into a character sequence using a somewhat looser version of the Lisp syntax rules.

Each kind of Lisp entity is represented by a different kind of Lisp expression (i.e., the syntax for different kinds of Lisp entities is different). Each section below focuses on the syntax for a single type of Lisp entity. Fortunately, many of the Lisp syntax rules will seem quite natural to you. The syntax that is peculiar to Lisp (and which you have probably not seen before) will be highlighted.

2.2.1 Representing Numbers

Lisp represents numbers using the same sorts of character sequences that you would use in any math class. For example, the character sequences `35`, `3.12` and `-87` represent the expected numbers (i.e., Lisp entities). Lisp also allows character sequences such as `3/4` and `5/8` to represent fractions. It should be kept in mind that each of these character sequences represents an indivisible Lisp entity that has no parts (as far as we are concerned). In other words, the character sequence `3/4` represents the number *three-quarters*, it does not represent the application of the division operator to the inputs *three* and *four*.

Given this sort of representation for numbers, here's an example of what happens when we type the character sequence `44` into the IW, followed by hitting the *Enter* key:[8]

`44` ⟶ [the number *forty-four* ⟹ the number *forty-four*] ⟶ `44`

Thus, when we type the character sequence `44` into the IW, it *appears* that the IW is simply echoing that character sequence back to us. In reality, the IW is doing a translation, an evaluation, and another translation, as indicated above.

The following listing shows the text that might appear in an actual IW session:

```
> 44
44
```

Notice that only the input and output character sequences are shown in the IW session, not the corresponding Lisp entities—whatever they would look like! In addition, in the above listing, the character `>` is used instead of the more cumbersome Lisp prompt that actually appears in a real IW session.[9]

Note that the IW is not obliged to use precisely the same representation of a number that you do. For example, if you type in the character sequence `0000000`, which is a silly, but legal representation of the number *zero*, the IW will reply with the character sequence `0`, as illustrated below:

`0000000` ⟶ [the number *zero* ⟹ the number *zero*] ⟶ `0`

Here's the corresponding IW session:

```
> 0000000
0
```

2.2.2 Representing Symbols

There are lots of character sequences that represent symbols in Lisp. If we were to precisely specify the syntax rules for representing symbols in Lisp, it would be a little messy. Fortunately, we don't have to. In this class, we will be using a convenient subset of the legal character sequences for symbols that will be more than sufficient for our purposes. In particular, any character sequence involving only letters represents a symbol. For example, the character sequence `xyz` represents a symbol. In addition, character sequences consisting of both letters and hyphens also represent symbols. For example, the character sequence `my-big-symbol` represents a symbol. In addition, the following commonly used character sequences represent symbols: `+`, `-`, `*` and `/`. We will introduce other ways of representing symbols, as needed, later on.

Because of the special way symbols get evaluated, when we type character sequences such as `xyz` and `most-positive-fixnum` into the IW, interesting things happen. For example:

`most-positive-fixnum` ⟶ [the symbol `most-positive-fixnum` ⟹ the number *536870911*] ⟶ `536870911`

In the above case, the character sequence `most-positive-fixnum` is first translated into a Lisp entity: the symbol `most-positive-fixnum`. Next, the symbol `most-positive-fixnum` is evaluated. Since the Global Environment is always initialized to contain an entry for this symbol, the evaluation of the `most-positive-fixnum` symbol is well defined and the IW does not generate an error.[10] The `most-positive-fixnum` symbol evaluates to the number *536870911*. Next, the IW needs to translate that Lisp entity (i.e., the number) into a character sequence so that we can see it on the screen. Not surprisingly, the IW uses the character sequence `536870911` for this purpose, as illustrated below.

[8] This article must use character sequences to represent Lisp entities, too! Thus, for example, the character sequence "the number *forty-four*" is used to represent a Lisp entity. This is unavoidable, since we have no way of actually putting that Lisp entity on display. We can only *refer* to it using character sequences.

[9] The actual Lisp prompt would typically include the name of the "package" with respect to which the expression was being evaluated. It might also include the line number. Since these items would only serve as a distraction, they are not shown here, or in subsequent examples.

[10] In general, if you ask the IW to do something that is undefined in the World of Lisp, it will report an error.

```
> most-positive-fixnum
536870911
```

Typing an arbitrary sequence of letters into the IW generally causes the IW to report an error because the corresponding symbol won't have an entry in the Global Environment and thus its evaluation is undefined, as illustrated below:

```
> fdajklfdajklfds
Error: Attempt to take the value of the unbound variable ‘fdajklfdajklfds’.
[condition type: UNBOUND-VARIABLE]
```

Later on, we'll see how to make the LVM insert new entries into the Global Environment for symbols of our choosing. For now, suppose we did whatever was necessary to make the LVM insert an entry into the Global Environment associating the number *thirty-four* with the symbol `x`. If we then typed the character sequence `x` into the IW, the following would happen:

`x` ⟶ [the symbol `x` ⟹ the number *thirty-four*] ⟶ `34`

In the IW, it would look like this:

```
> x
34
```

2.2.3 Representing Booleans

There is no boolean data type in Lisp. Instead, the symbols `t` and `nil` play the roles of *true* and *false*. When a Lisp session is first started, the Global Environment is automatically given an entry for each of these symbols. In fact, the *value* of the symbol `t` is `t`; and the *value* of the symbol `nil` is `nil`. Thus, when you type the character sequence `t`, the IW first translates it into the `t` symbol, then evaluates it (to itself), and finally reports back the character sequence `t`. Similar remarks apply to the `nil` symbol.

`t` ⟶ [the symbol `t` ⟹ the symbol `t`] ⟶ `t`

`nil` ⟶ [the symbol `nil` ⟹ the symbol `nil`] ⟶ `nil`

The following IW session illustrates these evaluations:

```
> t
T
> nil
NIL
```

2.2.4 Representing the Empty List

Lisp provides two different character sequences, `nil` and `()`, either of which can be used by programmers to represent the *empty-list* entity. However, Lisp always uses `nil`, as demonstrated below.

`nil` ⟶ [the `nil` symbol ⟹ the `nil` symbol] ⟶ `nil`

`()` ⟶ [the `nil` symbol ⟹ the `nil` symbol] ⟶ `nil`

```
> nil
NIL
> ()
NIL
```

2.2.5 Representing Functions

There is no character sequence that we can type into the IW that gets *directly* translated into a function entity. Too bad. So far, the only way we know for getting at function entities is to *refer* to them indirectly using symbols. For example, we can refer to the *addition* function using the `+` symbol, if we include it in an expression such as `(+ 3 4)`. Later on, we'll see how to define and refer to functions of our own design.

2.2.6 Representing Non-empty Lists

Non-empty lists are compound (i.e., non-primitive) Lisp entities. They are represented by character sequences beginning with a *left parenthesis* and ending with a *right parenthesis.* In between the two are character sequences representing the elements of the list, where the character sequences representing consecutive elements must be separated by one or more spaces. Thus, the valid character sequences for representing a list containing n elements have the following form:

$$\texttt{(}\ \langle \text{subseq}_1 \rangle\ \ldots\ \langle \text{subseq}_n \rangle\ \texttt{)}$$

where, for each i, $\langle \text{subseq}_i \rangle$ is a sequence of characters representing the i^{th} entity in the list.

Thus, for example, the character sequence, `(3 t x)`, represents a list containing the following Lisp entities: the number *three*, the symbol **t**, and the symbol `x`. Typing this sequence of characters into the IW would cause an error because the first element in this list is a number, not a symbol; thus, the evaluation of this list is undefined. The following example is one in which no error is generated:

`(+ 1 2)` $\longrightarrow$ [list containing: symbol `+`, number *one*, number *two* $\Longrightarrow$ number *three*] $\longrightarrow$ `3`

See Section 1 for the details about why this particular list evaluates to the number *three.* Here's the corresponding IW session:

```
> (+ 1 2)
3
```

2.3 Concluding Remarks

Now that we know what's going on behind the scenes, we can make our lives a lot easier by only writing down the character sequences. Of course, we must always keep in mind what's going on behind the scenes in the World of Lisp (i.e., we must have an accurate mental model). Below, are many of the examples seen earlier using the shorter, more convenient notation (that reflects what we see in the IW):

$\mathcal{C}_{\text{in}}$	$\rightsquigarrow$	$\mathcal{C}_{\text{out}}$
`44`	$\rightsquigarrow$	`44`
`0000000`	$\rightsquigarrow$	`0`
`t`	$\rightsquigarrow$	`t`
`nil`	$\rightsquigarrow$	`nil`
`()`	$\rightsquigarrow$	`nil`
`x`	$\rightsquigarrow$	`34`
`(+ 1 2)`	$\rightsquigarrow$	`3`

The wavy arrow ($\rightsquigarrow$) represents a translation, followed by evaluation, followed by another translation!

3 Special Forms

In Lisp, there is a special class of symbols called *keywords.* Examples of keywords include: `cond`, `defparameter`, `defun`, `if`, `lambda`, `let`, and `quote`. When the first element of a non-empty list is a keyword symbol, then that list is called a *special form.*[11] For example, each of the following character sequences represents a special form:

```
(defparameter x 3)

(quote (3 4 5))

(if condition then-clause else-clause)

(let ((x 4)) (+ x 8))
```

[11] Actually, some of these "special forms" are really *macros.* However, the distinction between macros and special forms is not important for this article.

The important thing about special forms is that they are *not* evaluated according to the default rule described in Section 1. Instead, a special form is evaluated according to a special rule that is specific to the type of that special form (which is determined by the keyword). There is one rule for evaluating `defparameter` special forms, another rule for evaluating `quote` special forms, and so on. Importantly, each `defparameter` special form is evaluated in the same way, just as each `quote` special form is evaluated in the same way. However, the rule for evaluating `defparameter` special forms is very different from the rule for evaluating `quote` special forms.

In fairly short order, you will be introduced to about a dozen different kinds of special form. For each kind of special form, you will have to learn the corresponding evaluation rule. However, you will use these special forms so often that their special modes of evaluation will become second nature after a short time.

Note. Recall that the first position in a list is called function position. Every other position in a list is called non-function position. Recall further that according to the default rule for evaluating non-empty lists, the first thing that is done is to fetch the "function value" for the symbol in function position. Next, each of the entities in non-function positions must be evaluated. In contrast, when evaluating a special form—which is also a non-empty list—the keyword in function position is used to determine how that list should be evaluated; keywords do not have "function values" associated with them. In addition, depending on the type of special form, it may happen that some of the entities in non-function positions are *not* evaluated.

The next sections introduce some of the more important special forms that you will use every day for the rest of your Lisp-programming life!

3.1 The `defparameter` Special Form

The `defparameter` special form is indicated by the `defparameter` keyword. As a character sequence, it has the form

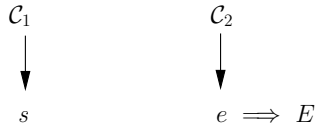
$$\texttt{(defparameter } \mathcal{C}_1 \ \mathcal{C}_2\texttt{)}$$

where $\mathcal{C}_1$ is a character sequence representing some Lisp symbol s, and $\mathcal{C}_2$ is a character sequence representing an arbitrary Lisp entity e, as illustrated below.

$$\mathcal{C}_1 \rightarrow s \qquad \text{and} \qquad \mathcal{C}_2 \rightarrow e$$

What happens when a `defparameter` special form is evaluated is illustrated below:

Evaluating the special form: `(defparameter` $\mathcal{C}_1\, \mathcal{C}_2$`)`



Global Environment Entry: | s | E |

where the thin arrows ($\longrightarrow$) represent translation from character sequences into Lisp entities, and the double arrow ($\Longrightarrow$) represents the process of evaluation, as described in Section 2.

After the translation from character sequences $\mathcal{C}_1$ and $\mathcal{C}_2$ into Lisp entities s and e, respectively, the entity e is evaluated, resulting in some entity E. (Notice that the symbol s is *not* evaluated.) Next, an entry is created in the Global Environment in which the Lisp entity E is associated with the symbol s. Notice that the primary purpose of a `defparameter` special form is a *side effect*: to create an entry in the Global Environment. The output value is less important. In any case, the output value is simply the symbol s.

Using the notation from Section 2, we have that:

$$\texttt{(defparameter } \mathcal{C}_1 \ \mathcal{C}_2\texttt{)} \quad \rightarrow \quad [\ \text{list containing } \texttt{defparameter} \text{ symbol, etc.} \quad \Rightarrow \quad \text{the symbol } s \ ] \\ \rightarrow \ s$$

This notation is not very helpful since it ignores the all-important *side-effect* of evaluating the `defparameter` special form (namely, the creation of a new entry in the Global Environment). Similarly, the wavy-arrow notation seen earlier does not help much:

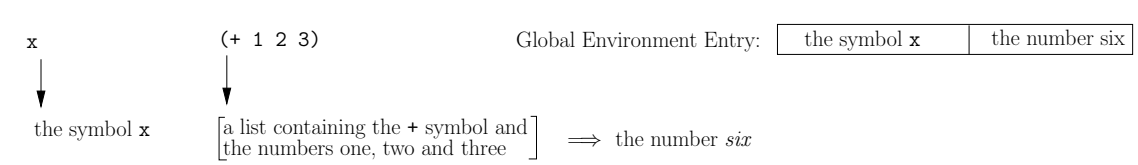
$$\texttt{(defparameter } \mathcal{C}_1 \ \mathcal{C}_2\texttt{)} \quad \rightsquigarrow \quad s$$

However, in the IW, we observe the following behavior:

```
> (defparameter y 38)
Y
> y
38
```

In this case, the `defparameter` special form caused the value `38` to be associated with the symbol `y` in the Global Environment. Afterward, the evaluation of the symbol `y` resulted in the value `38`.

Another Example. Typing the character sequence `(defparameter x (+ 1 2 3))` into the IW and hitting the *Enter* key would result in the number six being associated with the symbol `x` in the Global Environment, as illustrated below.



The output value for the expression `(defparameter x (+ 1 2 3))` is the symbol `x`. However, subsequent attempts to evaluate the symbol `x` would result in the value 6 being returned.

3.2 The `defvar` Special Form

The `defvar` special form is similar to the `defparameter` special form. The only difference is that if the symbol in question has already been declared (using either `defparameter` or `defvar`), then the `defvar` special form will *not* change the value associated with that symbol in the Global Environment. The following IW session illustrates the difference:

```
> (defvar z 19)
Z
> z
19
> (defvar z 819293)
Z
> z
19
> (defparameter z 1000)
Z
> z
1000
```

3.3 The `quote` Special Form

The `quote` special form is indicated by the `quote` keyword. As a character sequence, it has the form

`(quote` $\mathcal{C}$`)`

where $\mathcal{C}$ is a character sequence representing some arbitrary Lisp entity e, as illustrated below:

$$\mathcal{C} \rightarrow e$$

The evaluation of a `quote` special form is trivial. It just results in the Lisp entity e. Notice that e itself is not evaluated! Indeed, the whole point of the `quote` special form is to *shield* its argument from evaluation.

Using the notation from previous lectures, we may abbreviate this as follows:

`(quote` $\mathcal{C}$`)` $\rightarrow$ [*list* containing `quote` symbol and e $\Longrightarrow$ e] $\rightarrow$ $\mathcal{C}'$

Notice that the returned character sequence $\mathcal{C}'$ may not be the same as the character sequence $\mathcal{C}$, but it must represent the same Lisp entity e.

For example, the character sequence `(quote x)` represents a list containing two symbols: the `quote` symbol and the symbol `x`. The result of evaluating this special form is simply the symbol `x`, as follows:

`(quote x)` → [*list* containing symbols `quote` and `x` ⟹ the symbol `x`] → `x`

Notice that the symbol `x` is *not* evaluated! Using the more compact wavy-arrow notation, we may summarize the evaluation of this `quote` special form as follows:

`(quote x)` ⇝ `x`

This is quite different from the default evaluation lisp for non-empty lists! Here's the way it looks in the IW:

```
> (quote x)
x
```

Here's another example:

`(quote (1 2 3))` ⇝ `(1 2 3)`

Notice that the list containing the numbers *one, two* and *three* has not been evaluated. Indeed, any attempt to evaluate such a list would cause the IW to report an error since the first element of that list does not evaluate to a function. This example illustrates the use of a list as a container for data rather than something we'd like to have evaluated. The `quote` special form comes in handy for such cases. Here's how it looks in the IW:

```
> (quote (1 2 3))
(1 2 3)
```

Important. For convenience, Lisp provides an abbreviation for character sequences representing `quote` special forms. In particular, character sequences of the form `'`*charSeq* and `(quote` *charSeq*`)` represent the *same* Lisp entity: namely, a *list* containing the `quote` symbol and whatever entity is represented by *charSeq*. For example, `'num` and `(quote num)` both represent a *list* containing the `quote` symbol and the `num` symbol. Although the abbreviation is useful, it requires care to remember that it is used to represent *lists*. Here are some examples in the IW:

```
> (quote (eval eval))
(EVAL EVAL)
> '(eval eval)
(EVAL EVAL)
> (quote 37)
37
> '37
37
> (quote xyz)
XYZ
> 'xyz
XYZ
> (quote (quote xyz))
'XYZ
> ''xyz
'XYZ
```

4 The `setq` and `setf` Special Forms

The `setq` special form is analogous to the "assignment operator" found in most programming languages. It enables you to change a symbol's "ordinary value" in the Global Environment. The `setf` special form has additional functionality that we won't make use of right away. For now, you may use `setq` and `setf` interchangeably.

A `setq` special form has the following syntax:

`(setq` $\mathcal{C}_1$ $\mathcal{C}_2$`)`

where:

- $\mathcal{C}_1$ is a character sequence denoting a Lisp symbol (i.e., $\mathcal{C}_1 \rightarrow s$, where s is a Lisp symbol), and
- $\mathcal{C}_2$ is an arbitrary Lisp expression (i.e., $\mathcal{C}_2 \rightarrow e$ for some Lisp entity e)

Notice that in the above, we have only indicated the Lisp entities s and e that $\mathcal{C}_1$ and $\mathcal{C}_2$ get *translated into*; we have not talked about anything getting *evaluated.*

Evaluating a `setq` special form. A `setq` special form, like all special forms, has its own mode of evaluation. A `setq` special form is evaluated as follows:

- First, the Lisp entity e is evaluated. Let E be the resulting Lisp entity (i.e., $e \Rightarrow E$).
- Second, E is assigned as the new "ordinary value" for the symbol s in the Global Environment.
- Third, the value E is the value of the `setq` special form.

Notice that the symbol s is not evaluated. Notice, too, that the second step is a *side effect.* It causes the previous "ordinary value" for s to be lost forever. This side effect is the main point of the `setq` special form. In contrast, the value of the `setq` special form (i.e., E in step 3) is frequently ignored by the programmer.

Here is an Interactions Window session demonstrating the use of the `setq` special form:

```
> (defparameter john 56)
JOHN
> john
56
> (setq john 100)
100
> john
100
> (setq john (+ 2 3))
5
> john
5
> (setq john (* john john))
25
> john
25
```

In the above example, `setf` could have been used instead since it has all the functionality of `setq`. More will be said about `setf` later.

5 Defining and Calling User-Defined Functions

This section introduces a number of special forms and built-in functions that facilitate the process of defining user-defined functions and subsequently applying them to inputs.

5.1 The `lambda` Special Form

The syntax for the `lambda` special form comes directly from Alonzo Church's *Lambda Calculus.* The purpose of the `lambda` special form is to enable us to define our own Lisp functions. Although a `lambda` special form—like any special form—denotes a particular kind of *list*, the result of *evaluating* a `lambda` special form is always a Lisp function entity. The subsequent behavior of that function entity—should it ever be applied to any arguments—is determined by the contents of the special form—which we can specify.

Syntax of a `lambda` special form. A `lambda` special form has the following syntax:

`(lambda (`x_1 x_2 ... x_n`)` b_1 b_2 ... b_k`)`

where:

- each x_i is an expression denoting a Lisp symbol;
- each b_i is a Lisp expression of any kind;
- $n \geq 0$; and
- $k \geq 1$.

The symbols denoted by $x_1, \ldots, x_n$ are referred to as *parameters*; the expressions $b_1, \ldots, b_k$ collectively comprise the *body* of the lambda expression. Notice that there may be zero or more parameters, but the body must contain at least one expression.

Each of the following character sequences is a valid instance of a `lambda` expression.

```
(lambda () 4)

(lambda (x) (+ 3 x))

(lambda (x y z) (* x y z))

(lambda (x) 44 86 32 'hike)
```

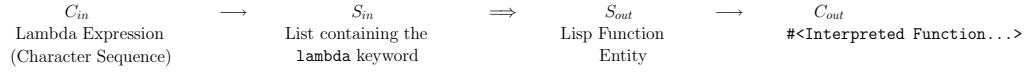
Evaluating a `lambda` special form. The result of evaluating a `lambda` expression is a Lisp function entity. Thus, evaluating each of the above expressions results in the creation of a new Lisp function entity, as demonstrated below.

```
> (lambda () 4)
#<Interpreted Function (unnamed) @ #x71caabd2>
> (lambda (x) (+ 3 x))
#<Interpreted Function (unnamed) @ #x71cc5cda>
> (lambda (x y z) (* x y z))
#<Interpreted Function (unnamed) @ #x71ced092>
> (lambda (x) 44 86 32 'hike)
#<Interpreted Function (unnamed) @ #x71d0b872>
```

In each case, the lambda expression itself denotes (i.e., is translated into) a *list.* And, of course, that list is a list, not a function. However, the *evaluation* of that list results in a Lisp function entity. The Interactions Window dutifully reports the result—a function—by displaying some weird textual information. Notice that this kind of textual information is not valid Lisp syntax. In fact, there is *no* valid Lisp syntax for directly representing Lisp function entities.

⇒ It is worth repeating: a `lambda` expression directly represents a list, not a function. Only when that list is evaluated do we get a function.

The following diagram summarizes the above discussion.



Up to this point, we haven't seen what these newly created Lisp function entities can do because we have not yet applied them to any arguments. Nonetheless, it is important to realize that a function entity can simply *exist* in the World of Lisp, even if it never ends up getting applied to any input.

In addition, you may be wondering just how the evaluation of a `lambda` special form gets from a list to a function. The details would require specifying how functions are represented internally by the LVM, which we don't really care about. Instead, we shall describe the resulting function entity by describing what happens should we subsequently apply that function to some inputs.

5.2 Naming User-Defined Functions: Version 1

Suppose we want to understand the Lisp function entity that results from evaluating a `lambda` expression of the form:

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

We can begin by giving that function a *name*. We can do that by using the following `defparameter` special form:

```
(defparameter square (lambda (x) (* x x)))
```

Notice that we have assigned our desired function as the "ordinary value" of the symbol `square`.[12] Lisp has a built-in function, called `funcall`, that can be used to apply a function to one (or more) arguments. In particular, the symbol `funcall` has a built-in function as its "function value". Thus, we can use the expression

```
(funcall square 8)
```

to cause our new function to be applied to the input `8`. Notice that the default case of evaluating a non-empty list is being used here. First, the "function value" associated with `funcall` is fetched. Second, the symbol `square` and the number `8` are evaluated. Thus, the "ordinary value" associated with `square` is fetched—which is our new function. Then, our new function and the number `8` are fed as inputs to the built-in `funcall` function, which causes our new function to be applied to the number `8`. The result is `64`, as illustrated below.

```
 > (defparameter square (lambda (x) (* x x)))
SQUARE
> (funcall square 8)
64
```

Here's a similar example for a function we shall call, `evalPoly`:

```
> (defparameter evalPoly (lambda (x) (+ (* x x 3) (* x 5) 8)))
EVALPOLY
> (funcall evalPoly 2)
30
> (funcall evalPoly 10)
358
```

5.3 Applying User-Defined Functions

Here's a formal description of what happens when a user-defined function is applied to some inputs. Suppose the user-defined function was generated by evaluating the following `lambda` expression:

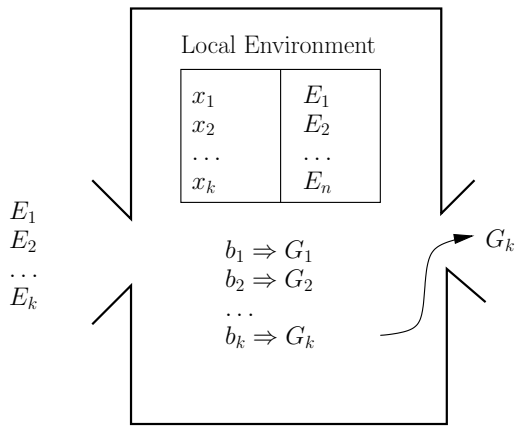
`(lambda (`x_1 x_2 ... x_n`)` b_1 b_2 ... b_k`)`

Suppose further that we have n Lisp entities: $E_1, \ldots, E_n$ that we want to feed as input to this function entity. Notice that the number of Lisp entities matches the number of input parameters in the `lambda` expression. The application of this function to these Lisp entities causes the following to happen:

- First, a *Local Environment* is created.
- Second, each symbol x_i gets its own entry in that Local Environment, where the "ordinary value" for x_i is the corresponding entity E_i.
- Third, each of the expressions $b_1, \ldots, b_k$ in the body of the lambda expression to be evaluated, in turn. In the course of evaluating them, any occurrences of the symbols $x_1, \ldots, x_n$ (i.e., the parameters) are evaluated with respect to the Local Environment, not the Global Environment. (Any other symbols are evaluated with respect to the Global Environment.) For example, an occurrence of x_3 in some expression b_i would evaluate to the corresponding input entity E_3.
- Finally, the output value is simply the Lisp entity that results from evaluating the last expression in the body (i.e., b_k).

[12] Since `square` does not currently have a "function value" assigned to it, we cannot use an expression such as `(square 6)` to apply our function to the number `6`. Later on, we'll see how to assign a function as the "function value" of a symbol.

This process is illustrated by the following diagram:



Example. Consider the following IW session:

```
> (defparameter cubit (lambda (x) (* x x x)))
CUBIT
> (funcall cubit (+ 2 5))
343
> (funcall cubit 10)
1000
```

When the expression, `(funcall cubit (+ 2 5))`, is evaluated, the built-in `funcall` function eventually applies our newly created `cubit` function to the input `7`. Focus on the application of our `cubit` function to the input `7`. It begins by setting up a local environment containing an entry for the single parameter, `x`. The "ordinary value" of this parameter in that local environment is `7`. Next, the single expression, `(* x x x)`, is evaluated with respect to that local environment. Thus, `(* x x x)` evaluates to `343`.

When the `cubit` function is applied to `10`, the local environment is created with a single entry for `x` with an "ordinary value" of `10`. In this context, the expression `(* x x x)` evaluates to `1000`.

5.4 Naming User-Defined Functions: Version 2

This section shows how we can name a user-defined function by assigning it to be the "function value" for a symbol of our choosing.

The built-in `symbol-function` function. Lisp provides a built-in function, `symbol-function`, that retrieves the "function value" for a given symbol, as illustrated below.

```
> (symbol-function '+)
#<Function +>
> (symbol-function 'eval)
#<Function EVAL>
```

Notice that the symbols `+` and `eval` had to be quoted in the above expressions to shield them from evaluation. We wanted the unevaluated symbols to be given as inputs to the `symbol-function` function.

Shorthand for fetching the "function value" for a symbol. Lisp provides a shorthand syntax for retrieving the "function value" associated with a symbol. In particular, if $\mathcal{C}$ is an expression denoting a Lisp symbol, then

`#'`$\mathcal{C}$

will retrieve the "function value" associated with that symbol. The following IW session gives some examples.

```
> #'eval
#<Function EVAL>
> #'+
#<Function +>
```

Setting the "function value" of a symbol using `setf`. Consider the expression `(symbol-function 'myFunc)`. It is evaluated by applying the built-in `symbol-function` function to the symbol `myFunc`, which retrieves the "function value" associated with the symbol `myFunc`. If we want to change the "function value" associated with `myFunc`, we can use an expression such as:

```
(setf (symbol-function 'myFunc) (lambda (x) (* x x))
```

This expression highlights some of the functionality of the `setf` special form that goes beyond what the `setq` special form can do. Recall that with `setq`, the second element of the list must be a symbol. With `setf`, the second element of the list can be an expression that looks like a function call. The above expression can be understood as an instruction to the LVM to change the behavior of the `symbol-function` function in the future. In particular, the expression, `(symbol-function 'myFunc)`, will subsequently evaluate to the squaring function defined by the `lambda` expression. Thus, subsequent attempts to fetch the "function value" for the symbol `myFunc` will return the indicated squaring function, as illustrated below.

```
> (setf (symbol-function 'myFunc) (lambda (x) (* x x)))
#<Interpreted Function MYFUNC>
> (symbol-function 'myFunc)
#<Interpreted Function MYFUNC>
> #'myFunc
#<Interpreted Function MYFUNC>
> (myFunc 10)
100
> (myFunc 5)
25
> (setf (symbol-function 'myFunc) (lambda (x) (* x x x x)))
#<Interpreted Function MYFUNC>
> (myFunc 10)
10000
> (myFunc 5)
625
```

We summarize this ability to change the behavior of the `symbol-function` function by saying that the `symbol-function` function is "`setf`-able" (or *settable*). User-defined functions can also be given this *settable* property.

5.5 The `defun` Special Form

The `defun` special form is provided to streamline the process of defining and naming user-defined functions. For example, the expression

```
(defun newFunc (x y) (* x (+ y x)))
```

is equivalent to the more complicated expression

```
(setf (symbol-function 'newFunc) (lambda (x y) (* x (+ y x))))
```

The following IW session gives an illustration, using `defun` to define a function called `newFunc` and the older, more complicated method to define a function called `altFunc`. The functions do the same thing.

```
> (defun newFunc (x y) (* x (+ y x)))
NEWFUNC
> (setf (symbol-function 'altFunc) (lambda (x y) (* x (+ y x))))
#<Interpreted Function ALTFUNC>
> (newFunc 10 15)
250
> (altFunc 10 15)
250
> (newFunc -2 3)
-2
> (altFunc -2 3)
-2
```

Using `defun` simply requires less typing. As with `lambda` expressions, the body of a `defun` expression can contain multiple Lisp expressions.

5.6 Applying an Unnamed User-Defined Function to Inputs

Consider the expression, `(lambda (x) (* x x))`. When typed into the Interactions Window, it evaluates to a function entity, as demonstrated below.

```
> (lambda (x) (* x x))
#<Interpreted Function (unnamed) @ #x71cd2a6a>
```

So far, the function exists, but hasn't been applied to any inputs. We can apply such a function to an input by placing that `lambda` expression in function position, followed by the desired input value, as illustrated below.

```
> ((lambda (x) (* x x)) 9)
81
> ((lambda (x) (* x x)) (+ 3 4))
49
```

The evaluation of the above expressions is carried out in a manner similar to the default rule for evaluating non-empty lists. The only difference is that the item in function position is a `lambda` expression, not a symbol. That is okay since a `lambda` expression evaluates to a function entity. That function entity is then applied to the value(s) generated by the expression(s) in argument position(s). In each of the above expressions, there is only one argument expression. In the first case, our squaring function is applied to `9`, generating `81` as output. In the second case, the function is applied to `7`, generating `49` as output.

The use of `lambda` expressions in function position in a list is not common; however, it will be used in the next section to demonstrate that the `let` special form can be defined in terms of the `lambda` special form.

5.7 The `let` Special Form

The `let` special form has the following form:

```
(let ((var1 val1)
      (var2 val2)
       ...
      (varn valn))
  expr1
  expr2
   ...
  exprk)
```

A `let` special form is evaluated as follows.

- First, a local environment is created.
- Second, each symbol var_i is given an entry in that local environment where the "ordinary value" for var_i is obtained by *evaluating* the corresponding expression, val_i.
- Third, the expressions $expr_1$, ..., $expr_k$ in the *body* of the `let` special form are evaluated in turn. In the process, whenever one of the symbols var_i is evaluated, the corresponding value from the local environment is used. For other symbols, the parent environment—which is often the Global Environment—is used. Thus, the local environment supercedes the Global Environment!

The following Interactions Window session demonstrates the evaluation of a `let` special form.

```
> (defparameter x 1000)
X
> (defparameter y 100)
Y
> (defparameter z 10)
Z
> (+ x y z)
1110
> (let ((x 3)
        (y 4))
    (+ x y z))
17
```

The first few expressions create entries in the Global Environment for the symbols `x`, `y` and `z`. The values for these global variables are used when evaluating the expression `(+ x y z)`. Next, a `let` expression is used to create a local environment containing variables named `x` and `y`. These local variables may have the same name as their global counterparts, but they are distinct from them. When evaluating the subsidiary expression `(+ x y z)` in this local context, the values for `x` and `y` are drawn from the local environment, which has precedence over the Global Environment. In contrast, since there is no local variable named `z`, its value is drawn from the Global Environment.

Deriving the `let` special form from the `lambda` special form. The `let` special form is a convenient abbreviation. It can be re-written in terms of a lambda expression, as illustrated by the following Interactions Window session.

```
> (defparameter z 1000)
Z
> (let ((x 3)
        (y 4))
    (* x y z))
12000
> ((lambda (x y)
     (* x y z))
   3
   4)
12000
```

Notice that in last expression, the `lambda` expression evaluates to a function which is then applied to the input arguments `3` and `4`.

Notice that the syntax of Lisp allows expressions to occupy multiple lines. This is quite useful when writing longer expressions. Emacs automatically indents sub-expressions to make longer expressions easier to read. Hitting the *tab* key will automatically cause the current line to snap to the appropriate amount of indentation.

Multiple expressions in the body of a `lambda` expression. The only way that intermediate expressions in the body of a function could have any impact is if they caused *side effects*. For example, in the following function, the built-in `format` function is used to print several lines of text prior to evaluating the last expression in the function's body.

```
> (defun verboseFunction (x y)
       (format t "Hi there~%")
       (format t "The value of x is: ~A~%" x)
       (format t "The value of y is: ~A~%" y)
       (+ x y))
VERBOSEFUNCTION
> (verboseFunction 3 9)
Hi there
The value of x is: 3
The value of y is: 9
12
```

Notice that, as always, the value returned by the function is simply the value of the last expression in the body of that function—in this case, the value of `(+ x y)`. The `format` function, and the *strings* that it operates on, are described in more detail in Paul Graham's book.

6 Defining a Useful `tester` Function

The following example defines a `tester` function that can be used to demonstrate the evaluation of Lisp expressions. The `tester` function takes a Lisp entity as its input, prints out a character sequence representing that entity, and then evaluates that entity and displays the result. The tester function returns the value `nil` because that is what the `format` function returns.

```
> (defun tester (expr)
    (format t "~A ====> ~A~%" expr (eval expr)))
TESTER
> (tester '(+ 2 3))
(+ 2 3) ====> 5
NIL
> (tester 'most-positive-fixnum)
MOST-POSITIVE-FIXNUM ====> 536870911
NIL
```

Notice how the `quote` special form is used to shield the input expression from immediate evaluation. Inside the body of the `tester` function, that expression is explicitly evaluated using the `eval` function.

7 Further Exploration of Lisp

The preceding sections have introduced you to the syntax and semantics of the Lisp programming language. The evaluation function plays a prominent role in the semantics. The evaluation of numbers and functions is trivial: they evaluate to themselves. The evaluation of symbols is based on table-lookup. The evaluation of non-empty lists depends on the form of the list.

- If the first element of the list is a symbol—but not one of the keyword symbols—then the default rule for evaluating non-empty lists is used. Accordingly, the "function value" of the symbol in function position is fetched, the "ordinary value" of the expression(s) in argument position is(are) fetched, and that function is applied to the resulting value(s). Thus, the default rule involves function application. The result of applying the function to the inputs generates the value of the original list.
- If the first element of the list is a keyword symbol, then the list is a special form. Each kind of special form has its own evaluation rule. A common feature of special forms, which clearly distinguishes them from the default rule, is that one or more arguments may be shielded from evaluation. The most obvious illustration of this is the `quote` special form, whose sole purpose is to shield its one argument from evaluation.
- If the first element of the list is a lambda expression, then that lambda expression defines a function. That function is then applied to inputs as in the default case.

It should be noted that this article has blurred the distinction between special forms and *macros*. In Paul Graham's book, he does not blur the two. Nonetheless, anything he calls a macro can be thought of as a special form. For example, `setf` is *really* a macro, but can be thought of as a special form. If you are interested, you can look at his chapter on macros to see how you can, in effect, define your own special forms.

Important! The Lisp language includes an amazing variety of built-in functions and macros. It also includes a handful of special forms. When you encounter one of these, it is *extremely* important that you first check whether what you are looking at is (1) a built-in function or (2) a macro/special form. If it is a built-in function, then you call it by using the default rule for evaluating non-empty lists—in which case, all argument expressions will necessarily be evaluated. On the other hand, if it is either a macro or a special form, you need to learn its unique evaluation rule. In particular, you need to know which arguments will be evaluated and which will be shielded from evaluation.

For example, there is a built-in function, called `eval`. Since it is a function, we call it using expressions of the form, `(eval` $\mathcal{C}$`)`, where $\mathcal{C}$ is some subsidiary expression. Since `eval` is a built-in function, the expression, `(eval` $\mathcal{C}$`)`, will be evaluated using the default rule. Thus, the expression $\mathcal{C}$ will necessarily be evaluated. Thus, if you want to shield it from evaluation, you need to explicitly quote it, as illustrated below.

```
> (eval (+ 2 3))
5
> (eval (quote (+ 2 3)))
5
> (eval (quote (quote (+ 2 3))))
(+ 2 3)
```

On the other hand, consider `or`. Looking it up in Paul Graham's book, we see that `or` is a macro; thus, it may not evaluate all of its arguments. The description of the evaluation rule for the `or` macro stipulates that it evaluates

each of its arguments, in order, until one of two things happens: (1) it finds one that evaluates to something other than `nil` or (2) it reaches the end of the list of arguments. In the first case, it returns that non-`nil` value; in the second case it returns `nil`. The following IW session illustrates this behavior.

```
> (or nil nil nil t nil nil)
T
> (or nil nil 3 nil nil)
3
> (or nil nil 3 4 5 nil)
3
> (or nil nil nil nil)
NIL
> (or nil nil 3 (/ 1 0) nil)
3
```

Notice that once it finds a non-`nil` value, it does not evaluate any additional arguments. This is particularly apparent in the last case, where the subsidiary expression, `(/ 1 0)`, is not evaluated. If it had been evaluated, it would have caused an error. Notice, too, that any non-`nil` value counts as boolean true; in contrast, only `nil` counts as boolean false.

The lesson to be learned from these examples is that you have the tools to guide your further exploration of the Lisp language, but you need to remember to use them! If you ever find yourself puzzled by the behavior of the Interactions Window as it evaluates some Lisp expression, return to basic principles of how expressions are evaluated! And remember that character sequences get translated into Lisp entities, which are then evaluated and translated back into character sequences. And remember to distinguish syntax from semantics (i.e., character sequences from Lisp entities)! Good luck!