Understanding Scheme In LilyPond

- Urs Liska

see:

- https://github.com/uliska/scheme-book
- https://github.com/yomannnn/scheme-book

Contents

1. Intro

1.1. Understanding Scheme In LilyPond

1.1.1. Urs Liska

GNU LilyPond is an extremely powerful and versatile text based music notation system with a strong focus on traditional craftsmanship and the aesthetics of manual plate engraving. LilyPond reads input files where the user has textually specified the *content* of a score and compiles them to graphical scores in PDF or SVG format (or additionally to non-graphical MIDI files). LilyPond's output can be tweaked to the least detail, but in fact even LilyPond's *behaviour* can be modified and extended right through to its inner gears! This is possible through LilyPond's built in extension language, Scheme

In a way, extending Lilypond with Scheme is like open heart surgery, and it is definitely not necessary for regular use and engraving scores. However, getting familiar with Scheme at least basically is always a good idea as one encounters its language constructs even in entry level LilyPond documents. Understanding these elements and their integration in LilyPond will make you feel more comfortable writing LilyPond files, and it opens up the horizon, even without ambitions to become a serious LilyPond "programmer".

For anybody except seasoned computer scientists this seems to be a thorny path, to which the LilyPond user mailing lists blatantly bear witness. The basic concepts of the language are different to what most users may know from more familiar script languages such as Python, JavaScript or even VisualBasic. The integration of two different languages (LilyPond and Scheme) adds to the complexity, and the advanced interaction with LilyPond's internals gives yet another layer on top.

The introductory chapters give an outline to the different aspects to the complexity of understanding Scheme in LilyPond. But in addition I have always felt that part of the problem is a lack of suitable learning resources. Unfortunately the official documentation is not exactly helpful in a smooth introduction to the world of extending LilyPond with Scheme, presumably because it's written too much from a developer's perspective. This was my motivation to start a category of Scheme tutorials on *Scores of Beauty*, the semi-official LilyPond blog. The idea behind these tutorials is to focus on a single task and explain it at a slow pace and sufficiently verbosely to give non-programmers the chance to get an idea *why* something is done a certain way, rather than just having them *copy* some ready-made code fragments.

When preparing a university course about Scheme in LilyPond I found myself writing the book that *I* would have needed several years ago. More and more it started heading towards a certain level of comprehensiveness, although it still doesn't claim to be a proper computer science textbook. In this book I'll cover a lot of Scheme topics, from the perspective of its use in LilyPond. What I'm trying to achieve is giving thorough explanations to the fundamental concepts and language structures. This tries to overcome or at least alleviate the obstacles that LilyPond users typically face when trying to approach Scheme. The book won't be "easy reading", but it is slow-paced enough to enable any

reader to get a firm understanding of how and why things have to be written in certain ways when using Scheme in LilyPond.

This book is for you if you

- want to be more comfortable with the scripting extension in LilyPond
- want to learn something new and powerful
- want to stretch your limits with LilyPond
- want to look under LilyPond's hood and experience the power of extending a program to its inner gears
- are simply a curious nature

Originally this material was conceived as an integrated part of The Plain Text And Music Book. However, as it grew so much I decided to extract it and release it as an independent web book. Of course the book is "work in progress", as thanks to its nature as an online book it's easily possible to make available what's already been written. The outline already gives an impression of what's planned, but experience tells that it will be both refined and extended during the actual writing process. If you find actual errors or sections that could be expressed more clearly, don't hesitate to contact me and/or open an issue at the book's Issue Tracker.

NOTE: This book is available on https://scheme-book.ursliska.de but currently it cannot be built, so the online version is not up-to-date against the state of the sources in this repository.

2. Intro - LilyPond's Scheme

2.1. LilyPond's Scheme

LilyPond uses *Scheme* as its extension language, OK. But it has to be noted that this is only half of the story, and in order not to get confused it's crucial to have a clear idea of what this actually means.

2.1.1. *Why* An Extension Language?

LilyPond adheres to the concept of compiling plain text input files. This makes it possible to include compiler instructions beyond the basic *contents* in the files, and these are written using an *extension language*. Programs could use arbitrary languages or even invent their own, but in LilyPond's case it was a natural choice to use Scheme, as it is the official extension language of GNU, GNU LilyPond's parenting organization and application framework.

LilyPond's internal architecture is also based heavily on Scheme, and as a consequence you can interact with LilyPond's internals the same way, as a developer working on LilyPond *or* a user writing input files. This makes the potentials of the extension language so incredibly powerful.

2.1.2. *What* Is Scheme?

Scheme is a programming language from the Lisp family of languages, which adhere to the paradigm of functional programming. This is very different from the concept of imperative programming which the majority of (non-professional) programmers is more familiar with and which is present in languages like Python, JavaScript or (Visual) BASIC, Java or the C family of languages. This makes getting in touch with Scheme challenging for many potential users.

2.1.3. *Which* Scheme?

It is important to note that *Scheme is not (necessarily) Scheme*, as there are many Scheme implementations around. In real life this means that when you search for "Scheme" solutions on the internet you have to expect results that may not be (completely) compatible with LilyPond. If you are not fully aware of that fact looking for help on the net can be quite off-putting.

The Scheme implementation used by LilyPond is the one included in Guile 1.8, which is the official application platform and extension language of the GNU operating and software system *(please note that Guile 1.8 is not the current version of Guile, so even web searches for "Guile Scheme" may bring up incompatible results)*. Therefore the official resource for any questions is the GNU Guile Reference Manual, especially the API reference. But it has to be said that this documentation is suited rather as a *reference* if you are already experienced with Scheme.

Apart from this the only trustworthy resources for Scheme *in LilyPond* are the LilyPond manual, this book, tutorials on Scores of Beauty or the lilypond-user mailing list.

2.1.4. *How* To Use Scheme in LilyPond?

Learning to use Scheme in LilyPond causes challenges on three layers:

- learning the language,
- integrating Scheme code in LilyPond code, and
- (advanced) interaction with LilyPond internals through Scheme.

The following pages will flesh this out somewhat more detailed, while the rest of the book will provide an idea about the "look and feel" of writing Scheme in LilyPond. Selected language characteristics are introduced slowly and thoroughly, while at the same time discussing how Scheme code can painlessly be mixed with LilyPond input code. The higher mysteries of advanced interaction with LilyPond internals are deferred to a later bookpart.

2.2. Learning the Language

The first and of course most fundamental challenge in learning Scheme in LilyPond is - Scheme itself.

Learners who are not used to functional programming tend to find Scheme confusing, with the most prominent aspect being the overwhelming number of parens *("parens" is a shortname for "parentheses" and is often used colloquially in computer literature although it's dubious if that term can be considered proper English. However, as it "flows" much better while the proper term is comparably awkward I will stick to the shortened form throughout this book)*. Cryptic error messages triggered by the slightest parenthesizing error are not likely to make the learning curve more pleasing. In my experience *evaluating expressions*, as explained in detail in a later chapter, is the first and most important concept that has to be thoroughly understood. Once that has been digested, everything else will be much easier to comprehend. I will try to make this learning process as smooth as possbile in the main bookpart about Scheme.

But there is another substantial cause for confusion, and while this book can't *eliminate* it the mere *knowlegde* of its existence will be of great help. Concretely, knowing that you're not alone will make you feel significantly less dumb and helpless. I'm talking about the fact that any Scheme keyword or function can have a number of different origins.

Most likely you will encounter existing Scheme code either by looking into arbitrary files or when investigating code that someone shared with you. Probably code shared in response to requests on the mailing lists is the single most common opportunity to get in initial touch with Scheme. The next natural step - after having managed to get such code to run at all - is usually the desire to modify that code in order to avoid having to ask the kind donor again. And this is where users are typically faced with incomprehensible heaps of code - and parens. Lacking the basic understanding of the fundamental ideas usually makes it unlikely to successfully change anything in the code. Figuring out what is going on in the code is made pretty complicated because there is no single place to look up the names, and web searches are generally not very helpful either.

What makes finding information about any given name or construct so difficult is that it can be defined in many different contexts:

- *Core Scheme * These are the easiest cases. But not every core feature is supported by each Scheme implementation.
- *Guile* (i.e. the actual Scheme implementation)
  Of course Guile is the reference for which Scheme elements are available in LilyPond
- *Guile modules*
  Not everything a Scheme system provides is available by default. Guile offers a huge number of modules that have to be included for its functionality to be accessible. LilyPond includes a number of such Guile modules automatically, but code that you see may depend on additional modules. As a consequence code that works in one context may trigger `unknown escaped string` errors in others.
- *LilyPond*
  LilyPond itself defines a large number of Scheme functions and keywords.
  Searching the net for these as "Scheme" keywords will probably fail
- *User code*
  Many names you encounter in Scheme code are functions or variables defined elsewhere in the file or in a user-provided library. In a way this is a clear case but experience tells that these names cause a whole lot of additional confusion if you can't really discern the previous four items either.

When looking at existing Scheme code you'll likely encounter *all* of these in one place, and this can be pretty confusing - in a way it feels like a system of equations with *at least* one variable too much. There is no once-and-for-all solution to this issue, but I can encourage you to keep calm and try to slowly disentangle everything step by step. And to ask. Probably it's not that you are too stupid for it, it's more likely that you're trapped in the described situation. So feel free to ask on the mailing lists, and try not to be satisfied by a solution that "just works" but ask until you have understood the underlying principle.

The bookpart about Scheme will cover Scheme concepts on their own, but always from the perspective of LilyPond.

2.3. Integrating Scheme in LilyPond

The second challenge is the integration of Scheme as an extension language in LilyPond input files. Actually this works quite smoothly because LilyPond's language itself is parsed in a way that is very close to Scheme. But on the other hand this affinity can be confusing to the user. I will give particular emphasis on disentangling these things by discussing Scheme always from the perspective of a LilyPond user. After the general Scheme bookpart I will cover the integration of Scheme *in LilyPond* in more depth, in a bookpart dedicated to the integration of Scheme in LilyPond.

2.4. Interacting With LilyPond Internals

The third challenge is the more advanced interaction with LilyPond's internals that is possible through Scheme. LilyPond can reveal information about every object's most secret properties, and the user can interact very closely with these internals. This is the part where the average LilyPond user tends to give up and will even be more confused than informed by the helping hands given by more knowledgeable people on the mailing lists. But on the other hand this is where the true power of extending LilyPond can finally be unleashed. And: this becomes much more accessible once the user has a firm understanding of the fundamentals of Scheme and its integration in LilyPond documents.

The third part of this book will introduce selected topics from this field, but is less targeted at being a comprehensive guide.

3. Scheme

3.1. Scheme Fundamentals

This main part of the book covers the fundamentals of Scheme as used in LilyPond documents. However, while it *does* give a basic explanation how Scheme code is integrated in LilyPond documents its focus is the language itself. In particular it doesn't give many examples that would make much sense in real-world documents, so if you work through the book in order you may find that part not to be real fun. However, I think that it is crucial to have a firm understanding of these fundamentals because it's exactly these that make using Scheme in LilyPond so (unnecessarily) confusing if not learned properly.

If it is an issue for you having to learn the concepts without the opportunity to try them out right away I have two alternative suggestions for you. You may jump directly to the Scheme in LilyPond part. But as I won't explain the fundamental concepts there again you will need to jump back and forth to look up anything you have difficulties with. This way you will only need to read through the explanations of what you really need, but on the other hand you run the risk of neglecting some fundamental understanding.

As another "offer" this part of the book provides a Music Function Primer, which gives you a few tools to use as building blocks in LilyPond documents. With these you can try out the Scheme concepts in actual score contexts. However, even if you intend to go that way I strongly recommend reading the next two chapters before.

3.2. Everything In Scheme Is An Expression

One fundamental idea learners have to internalize is the concept of *expressions*. While the difference may be very subtle it is one of the major walls new Scheme users tend to run into. In many languages a program consists of a sequence of *statements* that are *executed*, and function calls are such statements as well. In Scheme on the other hand virtually everything is an *expression* that *evaluates to* some *value*. Understanding this avoids confusion with regard to program flow, but above all it is the key to getting one's head around that thing with the countless parens.

We will investigate Scheme expressions using a *Scheme shell* or REPL (Read-Eval-Print-Loop), which - as the acronym suggests - reads in an expression, evaluates it and immediately prints the resulting value. For all but the most trivial use cases this shell is too limited, but for experimenting with expressions and data types it is pretty much ideal.

LilyPond provides such a shell whose most important advantage is that it provides exactly the environment LilyPond is running in. Firing it up with the command `lilypond scheme-sandbox` you can test how any given expression behaves in the LilyPond context:

```
$ lilypond scheme-sandbox
GNU LilyPond 2.19.39
Processing `/home/username/lilypond/usr/share/lilypond/current/ly/scheme-sandbox.ly'
Parsing...
guile>
```

At this `guile>` prompt you can enter any single Scheme expression, and its value will immediately be printed to the command line. In the following examples the remainder of the line starting with `guile>` is the expression that you will enter at the command prompt, while the subsequent line(s) are the printout of the expression's evaluation.

3.2.1. Literals

The most basic type of expression is the literal value. Try to enter a simple `5`:

```
guile> 5
5
```

Even at this basic stage we can say that you entered an expression, `5`, which evaluates to a value, `5`. Literals are also called *constants* or *self-evaluating expressions*. Try this with other literal values of different types:

```
guile> "I'm a string"
"I'm a string"

guile> -1.6
-1.6

guile> #t
#t

guile> '(1 . "Hello")
(1 . "Hello")

guile> '(1 2 3)
(1 2 3)

guile> red
(1.0. 0.0 0.0)
```

The first three expressions were a *string* literal, a floating point *number*, and Scheme's notation of the boolean *true* value. Don't worry, we'll dissect the strange notation of the last three expressions soon, in later chapters.

3.2.2. Procedure Application

OK, just having literal values is not really exciting, so let's type in something more interesting:

```
guile> (+ 12 17)
29
```

This is a simple addition of two numbers, expressed in Scheme's typical notation with the operator written first, before the operands. But actually this isn't the right perspective to understand the matter. It is not just an example of a somewhat unusual syntax but rather the core of how Scheme works.

3.2.2.1. Expressions Apply Procedures

What we just typed in is not a "command" or "statement" but an *expression*, an expression whose value after evaluation is `29`. Formally the expression we typed at the prompt is a *list*, Scheme's most basic and fundamental form of data. A list in Scheme is an arbitrary number of elements, grouped by parens. The current list has three elements: a `+` sign and two integer numbers, `12` and `17`. Whenever Scheme sees such a list it considers the first element a *procedure name* and tries to call that procedure. The remaining elements of the list are passed to the procedure, or - in Scheme-speak - the procedure is *applied* to the remaining elements. So in our example we apply the procedure `+` to the values `12` and `17`, which evaluates to `29`. Therefore the whole expression evaluates to `29` - and from the perspective of the program the whole thing *is* `29`.

This is the nucleus of how Scheme works and from which the whole language is built. In fact it's the core of the whole Lisp family of languages, "Lisp" being an acronym for "List Processing".

```
guile> (string-append "A" " " "B")
"A B"
```

In this example the list consists of four elements. The procedure `string-append` is applied to three strings, evaluating to a new string, concatenated from the three ones. This example is maybe a little bit easier to digest as `string-append` looks more like a procedure name than `+`, but basically it's the same thing as in the previous example. What we perceive as a procedure and its arguments is in fact a list whose first element is a procedure.

3.2.2.2. Applying Procedures to Multiple Arguments

It feels natural that `string-append` concatenates all of its "arguments", but as said we are talking of applying the procedure to all remaining list elements. In order to get a deeper understanding we can look at an example that is more different from what we are used to:

```
guile> (/ 100 2 2 5)
5
```

What is happening here? We have a procedure `/` and apply it to four remaining list elements consecutively: divide 100 by 2, divide the result (50) by 2, finally divide the result (25) by 5. This *is* quite different from most programming languages where one would have to explicitly use the operator for each subsequent division.

3.2.2.3. Procedure "Type"

```
guile> (1 "2" 3.14)
ERROR: Wrong type to apply: 1
ABORT: (misc-error)
```

Oh, what is wrong here, why should `1` be the wrong type to "apply"? As said Scheme will interpret the first element of a list as a procedure name and try to call that procedure. So obviously that first element must *be* a procedure. Using the REPL we can easily check what Scheme thinks of these procedures when passed as literals:

```
guile> string-append
#<primitive-procedure string-append>

guile> +
#<primitive-generic +>
```

Obviously procedures don't really have a "value", therefore the interpreter uses this special kind of `#< >` notation to tell us what it has read. `string-append` is a "primitive-procedure", and `+` is a "primitive-generic" (which more or less is the same as a procedure).

3.2.2.4. Procedure Names vs. Expressions

The error in the previous section is caused by mixing procedures and expressions. And it is typically encountered when the input code mistakenly uses the procedure invocation instead of its name. In order to demonstrate the difference we will show another example: a procedure (defined by LilyPond), entered without and with parentheses:

```
guile> ly:version
#<primitive-procedure ly:version>

guile> (ly:version)
(2 19 39)
```

In the first invocation `ly:version` is considered a literal, revealing that it is a procedure. In the second invocation the parens cause that procedure to be *called*, evaluating to a list with the version numbers for the current LilyPond version (so you'll likely get a different result when you try it out).

The final example is the real-world cause for the "Wrong type to apply" error that is also very confusing for beginners:

```
guile> ((+ 1 2))
ERROR: Wrong type to apply: 3
ABORT: (misc-error)
```

What is happening here? We have entered one layer of parentheses too much. Of course it's a simplistic example but when you are trying to modify real-world code this is something you will run into quite often.

So what happens (and will be explained in more detail in the next section) is: First the inner expression `(+ 1 2)` is evaluated and replaced with the result, leaving `(3)` to the interpreter. In the next step to evaluate the complete expression Scheme will try to invoke the procedure `3` because its the first element in the remaining list. *Now* the error message is completely understandable.

3.2.2.5. Summary:

Whenever you see an expression wrapped in parentheses you know that it is a list whose first element should be a procedure, which is then applied to all remaining list elements *(note that there is the concept of* quoting *that makes an important difference here. We will go into detail about this in a later chapter)*. The value this expression evaluates to is the result of the procedure application. This statement may seem trivial now, but keeping this in mind very firmly will help you significantly when having to disentangle complex nested structures in Scheme.

3.2.3. Nested Expressions

Every expression *evaluates to* something, and from Scheme's perspective this "something" then replaces the original expression. Expressions can *contain* other expressions, and we speak of "nested expressions" then. In that case Scheme subsequently evaluates the expressions from right to left (or from inside to outside).

In the following example of a more complex calculation the nested expressions are evaluated in turn and replaced with the resulting value:

```
guile>(+ (- 14 4) (* 3 (- 5 3)))
16
```

The following list shows the intermediate states of that nested expression up to its final evaluation to a simple value:

```
(+ (- 14 4) (* 3 (- 5 3))) ; (- 5 3) => 2
(+ (- 14 4) (* 3 2))       ; (* 3 2) => 6
(+ (- 14 4) 6)             ; (- 14 4) => 10
(+ 10 6)                   ; => 16
```

Already at this stage of an only slightly complex calculation the overall expression has *three* closing parentheses. It is clear that with more complex expressions this can quickly grow to a large number of nesting levels, especially when considering that procedures can be much more complex items than the simple operators we just had. Note that this expression still doesn't *do* anything useful yet, from the program's perspective it just represents `16` - so it will have to be integrated somewhere, resulting in even more nesting layers. Take the following expression as a contrived example.

```
guile> (let* ((init-value (+ (- 14 4) (* 3 (- 5 3))))
              (processed-value (* (+ init-value 4) (- init-value 3))))
        (display processed-value))
260
```

You can and should for now ignore everything you don't know about that (we'll soon cover it in the chapter about variable binding) but simply realize that within that "mess" we have inserted the previous expression and that it actually evaluates to a simple `16`, so the expression could as well have been written as

```
guile> (let* ((init-value 16)
              (processed-value (* (+ init-value 4) (- init-value 3))))
        (display processed-value))
260
```

Getting lost in nested parens is one of the most common - and potentially frustrating - experiences new Scheme users have. But when you strictly keep in mind that each pair of parens encloses one expression it is possible to keep the head over water.

3.2.4. Some Preliminary Words on Program Flow

So far we have only talked about single expressions. But of course these don't make up complete programs. A Scheme program consists of either a single expression or a sequence thereof. Whenever the complexity of the task suggests factoring out functionality in procedures this is done, but not through explicitly *calling* them but through expressions that implicitly invoke them, as we have discussed in this chapter. Of course users can define their own procedures, which will be covered later. But any procedure can be understood as a single expression that - again - *evaluates* to its "result".

3.2.4.1. "Empty" Expressions

It has to be said that there are expressions that do *not* evaluate to anything useful, and Scheme treats their value as `<unspecified>`. These correspond to *void functions* in languages like C++ or Java or *procedures* in languages that discern between *functions* (do return a value) and *procedures* (do not). At this point we don't have the means to show these, but it is a good idea to keep in mind that such "empty" expressions exist.

3.3. Including Scheme in LilyPond

The first thing to understand is how - on a very basic level - Scheme code can be used in LilyPond documents. So this is what we will first investigate.

Note: With LilyPond 2.19.22 some substantial *simplifications have been introduced in how Scheme can be integrated in LilyPond code. In case of doubt this book will strongly prefer the* new *syntax and exclusively explain this.*

3.3.1. Switching Between LilyPond and Scheme

In order to insert code in a different language the parser must discern between the two layers, which it does through the `#` sign that marks the transition from LilyPond to Scheme. Concretely, whenever the parser encounters this marker it will interpret the *immediately following code* as a *Scheme expression.*

So to insert any Scheme expression into a LilyPond document you have to prepend it with `#`. The best way to understand this is to see it in action. More or less *any* LilyPond user will already have assigned a Scheme value to an override:

```
{
  \once \override DynamicText.extra-offset = #'(2 . 1)
}
```

The # tells LilyPond's parser to parse one complete Scheme expression, which happens to be the *pair* `'(2 . 1)` (a later chapter will go into more details about Scheme's data types.)

3.3.1.1. Exceptions

LilyPond very much "thinks" like Scheme, and all input will internally be converted to Scheme. Therefore some data types can be entered literally, without explicitly switching to Scheme, and the following number assignments are equivalent:

```
{
  \once \override Stem.length = #7.5
  \once \override Stem.length = 7.5
}
```

This feels very natural, but it can cause some confusion once one starts thinking about it. The point is that it is actually the *LilyPond* parser that performs this transparent conversion.

The same is true for strings *(TODO: should it be explained what a "string" is?)* that can additionally be written with or without double quotes (*NOTE:* LilyPond/Scheme strings *have* to use *double* quotes, as single quotes have a completely different meaning). Again, the following assignments are equivalent:

```
\header {
  title = #"MyPiece"
  title = "MyPiece"
  title = MyPiece
}
```

The first one is the "regular" Scheme syntax: switching to Scheme mode, then writing a proper string with quotes. The other ones are simplifications made possible through LilyPond's parser. But in all three cases the variable `title` now refers to the *string* `MyPiece`. *Note:* the third version is only possible with single words. Something like `"My Piece"` *must* be enclosed in the quotes.

There are a few other data types where this is possible, but we will discuss them at a later point. For now you should only keep in mind that you have to use # to switch to Scheme syntax - but not *always*.

3.3.1.2. Displaying Scheme Values

Sometimes it is necessary, and for learning it is often enlightening, to print some Scheme values to the console. To start with, there are two ways to do so, using Scheme's `display` procedure or LilyPond's `ly:message` family of functions:

```
myVariable = "This is a variable"

#(display myVariable)
#(ly:message myVariable)
```

There are some notable differences between the commands:

- `display` can show *any* value while `ly:message` only processes strings (but this can be circumvented using the `format` procedure)
- `display` will not produce a newline, so that should always be done manually (through `#(newline)`)
- `ly:message` will print immediately while `display` only acts after parsing has been finished.

We will regularly use these methods for demonstrating parts of our code in the subsequent chapters.

3.3.1.3. LilyPond and Scheme variables

Another thing to note is that LilyPond variables are interchangeable with Scheme variables in LilyPond input files. Variables can be defined using either syntax:

```
% define a variable using LilyPond syntax
bpmA = 60

% define a variable using Scheme syntax
#(define bpmB 72)
```

We have two variables, `bpmA` and `bpmB`, one of which has been entered as a LilyPond, the other as a Scheme variable (as an exercise you can think about this expression in the light of what you learnt in the previous chapter). But internally they are now the same kind of variable and can be accessed in the same way:

```
{
  % assign a tempo using a literal value
  \tempo 8 = 54
  R1

  % assign tempos using the variables with LilyPond syntax
  \tempo 8 = \bpmA
  R1

  \tempo 8 = \bpmB
  R1
}
```

However, as they are stored as Scheme variables internally we can also refer to them using the Scheme syntax (i.e. switching to Scheme with `#` but *not* enclosing them in parens, as these variables are constants or self-evaluating expressions):

```
{
   % assign tempos by referencing variables using Scheme
  \tempo 8 = #bpmA
  R1

  \tempo 8 = #bpmB
  R1
}
```

Each of these ways to access the variables will work interchangeably, and it depends on the context which one should be used. This flexibility may seem confusing but it helps if you strictly remember that *all* values and variables are maintained as Scheme structures internally and that setting and accessing them can always be done through Scheme or LilyPond syntax.

3.4. Music Function Primer

This chapter will give you some building blocks that you can use to try out the Scheme features discussed in the current bookpart. They are provided "as-is", and you should be aware that you are not expected to really understand them at this point. I am talking about `music-`, `scheme-` and `void-` functions, which are discussed in-depth in a dedicated chapter. In this chapter I will only give you a brief overview, and you can use these functions to experiment with the Scheme concepts discussed in the remainder of this bookpart.

Music-, scheme- and void-functions are expressions just like anything else in Scheme. In light of the previous chapter you can say that music functions evaluate to a "music expression", scheme functions evaluate to any Scheme value, and a void function's value is `#<unspecified>`. That is, wherever you can write music (and overrides are "music" too) you can instead write a music function, wherever you can use a Scheme value (for example assigning values to overrides) you can instead write a scheme function, and void functions can be used whenever no return value is needed but something has got to be *done*.

All three forms basically look the same, and I will demonstrate this with a scheme function as an example:

```
mySchemeFunction =
#(define-scheme-function (argument-name)
   (string?)
   ; function body
   )
```

We use a variable name and assign it a (music) function. Following the keyword `define-scheme-function` is a list of parameter names (in the example just one), which is then followed by a list of *predicates* (see data types). For each parameter there must be one predicate specifying the expected data type, in this case `string?`.

The function body can consist of an arbitrary number of expressions, where the last one specifies the return value (which is then substituted in the LilyPond document).

You may have seen such functions where the parameter list started with (literally) `parser location`. This is not necessary anymore in LilyPond releases starting with the current 2.19 development line. The current stable release 2.18.2 still requires this, but this is a topic that I won't discuss anymore in this book.

3.4.1. Scheme Functions

As we've seen Scheme functions are created using the `define-scheme-function` keyword. They evaluate the Scheme value (of arbitrary type) that the last expression in its body evaluates to, and this value is then used in the LilyPond document:

```
mySchemeFunction =
#(define-scheme-function (name)
   (string?)
   (string-append name " | " name)
   )

\header {
  title = \mySchemeFunction "Doubled Title"
}
```

This will call the scheme function and assign `Doubled Title | Doubled Title` to the `title` paper variable.

3.4.2. Music Functions

Music functions are created using the `define-music-function` keyword. Instead of arbitrary Scheme values they are expected to return a music expression, so the last expression in the body must be "music". The easiest way to achieve this is to switch back to LilyPond mode using `#{ #}`. If it is necessary to access Scheme values from within that one has to - again - use the # hash sign:

```
myMusicFunction =
#(define-music-function (color)
```

```
   (color?)
   #{
     \override NoteHead.color = #color
     \override Stem.color = #color
     \override Flag.color = #color
   #})

{
  c'4
  \myMusicFunction #red
  d'8
}
```

The `#{ #}` is *one* Scheme expression, but as in any LilyPond music expression it can have many consecutive elements like the overrides in this example.

3.4.3. Void Functions

Void functions are created using the `define-void-function` keyword. Regardless of the value of the last expression in their body they do *not* return anything and can therefore be used virtually anywhere in a LilyPond file. Void functions are used when something has to be done or modified but we don't make use of the return value:

```
myVoidFunction =
#(define-void-function (a-value)
   (number?)
   (display (* 2 a-value)))

\myVoidFunction 4
```

3.5. Scheme Concepts

The following chapters introduce fundamental concepts of Scheme. To some extent this introduction is independent from LilyPond, as most of the concepts are general Scheme techniques. However, they are all taken from the LilyPond perspective.

We start with discussing *data types*, followed by a sequence of discussions of more complex concepts and techniques.

3.5.1. Data Types

When talking to a computer there's no room for ambiguity. Approaching a computational task with irony is almost bound to fail, as the programming language will always want to know what it really *is* you are talking about. Therefore in programming languages each value has a *type*, and this is not different in Scheme. If a function processes a number you have to *give* it a number, sometimes you even have to make a difference between, say, integer and real numbers, e.g. between `1` and `1.0`. And so on.

Scheme is extremely flexible and permeable with regard to type. If you have an expression like

```
(some-procedure arg1 arg2)
```

then Scheme will not check in any way what types the values `arg1` and `arg2` have. Any type mismatch will only become visible upon the procedure application within the expression:

```
guile> (+ "1" "2")
ERROR: In procedure +:
ERROR: Wrong type argument in position 1: "1"
ABORT: (wrong-type-arg)
```

You can compare this behaviour with other programming languages. In "strongly typed" languages like Java or C++ the function interface would already act as a shield against arguments with wrong types. The `"1"` and `"2"` wouldn't even reach the +, so to say. Other languages, like e.g. Python or JavaScript will try to make the best out of it. In the example they would both *concatenate* the two digits and return `"12"`.

So Scheme doesn't check the type of an argument passed into an expression but doesn't do anything to help with improper types either. This is an important characteristic because instead of the three literal elements the expression could equally consist of expressions themselves whose resulting type will only be known at runtime:

```
((return-a-procedure) (return-arg1) (return-arg2))
```

So Scheme is open for very elegant and concise dynamic programming tricks. Admittedly this is still pretty abstract. What you should remember at *this* point is that the type of an expression's elements is not enforced by Scheme, but that passing arguments of unsuitable type will cause errors during procedure application.

3.5.1.1. Predicates

Of course it's no good idea to simply throw values at a procedure and wait for errors to occur or not. In Scheme it is therefore very common to check the type of a value before using it. As a response the program will either reject the value or select suitable code to be executed. This will be discussed in a later chapter on conditionals.

Scheme does not have a (regular) way of revealing the type of something directly (which is part of the open characteristic). The approach taken instead is something like asking "is this object behaving like a certain type, does it have the right properties?" This is achieved using *predicates*. Predicates are procedures that take an object and return *true* if the object matches a certain definition or *false* otherwise. By convention the name of a predicate has a trailing question mark, so for example `number?` checks if a given value is a number etc.

```
guile> (string? "I'm a string")
#t

guile> (integer? 4.2)
#f

guile> (boolean? #f)
#t

guile> (boolean? "true")
#f
```

We will come back to the topic of predicates after having discussed writing procedures, for now we will continue with the discussion of a number of basic data types.

If at some point you might want to get more in-depth information about simple data types you should consult the reference in the Guile manual.

3.5.1.2. Numbers

Numbers are ubiquituous in computing, and according to the Guile reference there is a whole lot of aspects that can be discussed about them. However, for LilyPond users there is not that much required information to begin with. Scheme supports many different number types, but for LilyPond use, integers and real numbers are usually sufficient.

To check if a value is a number we can use the predicate `number?`

```
guile> (number? 4.2)
#t

guile> (number? "Hi")
#f
```

3.5.1.2.1. Integers

Integers, or whole numbers, are written as they are, `5`, `-12` or `1234`. If written in a LilyPond file they *can* be prepended with the `#` sign or written literally, so `#13` is equivalent to `13`

```
\paper {
  min-systems-per-page = 5
  max-systems-per-page = #7
}
```

If arithmetic operations are performed on integers the results are integer as well:

```
guile> (+ 123 345)
468

guile> (* 4 5)
20
```

The predicate for integers is `integer?`

3.5.1.2.2. Real and Rational Numbers

Real numbers are all the non-integer numbers, in computing often referred to as floating-point numbers. Rational numbers are a subset thereof, namely all numbers that can be expressed as a fraction of integers. Fractions are written as two integers separated by a slash, without any whitespace in between.

```
guile> (real? 1.20389175)
#t

guile> (fraction? 5/4)
#t

guile> (fraction? 1.25)
#f
```

3.5.1.2.2.1. Mixing Reals, Rationals and Integers

Above we said that arithmetic operations on integers produce integers again. However, if only *one* operand is a real number the whole expression gets converted to reals:

```
guile> (+ 3 1.0)
4.0
```

When integers are divided Scheme will express the result as a fraction that is shortened as much as possible, but as soon as one real number is involved the fraction is converted to a floating point number:

```
guile> (/ 4 2)
2

guile> (/ 10 4)
5/2
```

```
guile> (/ 4 3)
4/3

guile> (/ 4 3.0)
1.33333333333333
```

3.5.1.2.2.2. Exact and Inexact Numbers

Integers and fractions are always *exact* values that can be recalculated as often as desired, giving always the same result. Real numbers on the other hand are *inexact* as they are subject to an arbitrary *precision* as implemented by the programming language system. This means that when performing mathematical operations with real numbers one has to expect the possibility of rounding errors. Generally this is not an issue when using Scheme in LilyPond, but it should be noted that there *is* this issue.

3.5.1.2.3. Calculations With Numbers

In order to learn what operations can be done with numbers in Scheme it may be a good idea to familiarize oneself with the documentation on integers and reals. Diving into these pages *now* may also be a good test on how to handle the reference style of the GNU Guile Manual.

3.5.1.3. Booleans

Boolean expressions are expressions that represent a “true” or ”false” value. This sounds trivial, but in fact, although *any* programming language relies on having some boolean representation , there are significant differences in how they are actually handled.

Scheme has two explicit constants for true and false, namely `#t` and `#f`. They start with the `#` hash sign, therefore it has to be stressed that *in LilyPond* these constants have to be written with *two* hash signs, one for switching to Scheme and the other as part of the constant itself. This is consequent but often causes confusion:

```
guile>#t
#t

guile>#f
#f
```

but

```
\paper {
  ragged-bottom = ##t
  ragged-last-bottom = ##f
}
```

3.5.1.3.1. `#t` vs. ”A true value”

We have already encountered booleans in predicates, which are procedures that evaluate to `#t` or `#f`. However, beside the two boolean *constants* there is the concept of “true value” and “false value”. Expressions that “have a true value” do *not* necessarily “evaluate to `#t`”. Rather they are everything that is not `#f`. This distinction will become important when we talk about conditionals.

3.5.1.4. Strings

Strings are sequences of characters and more or less handle what we see as *text* in documents. From the perspective of data processing and programming languages dealing with strings is surprisingly complex, and working with strings is a fundamental task also when you’re writing Scheme in LilyPond. Guile provides a large number of functions for string processing, but we’ll only cover a

few specific aspects in this “data types” introduction. You can find all details about Scheme’s string processing in the Guile manual.

3.5.1.4.1. Writing Strings

In Scheme strings are always written using *double* quotes. Other languages understand single and double quotes interchangeably or with subtly different meanings, but single quotes are reserved for a completely different purpose in Scheme:

```
guile> "Hello"
"Hello"

guile> Hello
ERROR: Unbound variable: Hello
ABORT: (unbound-variable)

guile> 'Hello'
Hello'
```

The first example enters a *string* that evaluates to itself - strings are self-evaluating, literals, constants in Scheme. The second example interprets the input as a name for a variable that has not been defined yet. And the final example interprets the input as a “quoted symbol”, with the trailing single quote characteristically being part of the symbol. You will read more about both cases in symbols and quoting.

As you have seen in the section about including Scheme in LilyPond LilyPond’s parser treats strings specially and doesn’t require an explicit switch to Scheme mode through #. For all strings that don’t contain any special characters and that are single words you don’t even have to use quotation marks. As said the first three of the follwoing assignments are equivalent while for the last one the quotation marks are mandatory:

```
\header {
  title = #"MyPiece"
  title = "MyPiece"
  title = MyPiece
  title = "My Piece"
}
```

3.5.1.4.2. Escaping Special Characters

There are a number of special characters that can’t directly be inserted in strings, although Guile/LilyPond supports Unicode out-of-the-box. Inserting such non-standard characters is done using “escaping”: the parser sees an *escape character* and treats the immediately following content as a special object. In Scheme - just like in many other languages - this escape character is the \ backslash.

3.5.1.4.2.1. Quotation Marks

The most obvious character that has to be escaped is the double quote itself - as the parser would interpret this as the *end* of the string by default:

```
\header {
  title = "My \"Escape\" to Character Land"
  subtitle = "Where things can go "terribly" wrong"
}
```

In this example the title is escaped correctly while in the second example the quotes break the string variable, which you can already see from the syntax highlighting. LilyPond will in this case produce a pretty confusing error message but will at least point you to the offending line of the input file.

Note: this is only true for straight double quotes. From a typographical *perspective it is often better to use typographical (or "curly") quotes anyway *(e.g. `“English”`, `„Deutsch“` or `«Français»`). These don't need escaping.

3.5.1.4.2.2. The backslash

So if the backslash is used to indicate an escape character how can a backslash be used as a character in text? Well, in a way that's self-explaining: through escaping it. To print a backslash you have to escape it - with a backslash:

```
\markup "This explains the \\markup command."
```

3.5.1.4.2.3. Arbitrary Escaped Characters

There is a list of other special ASCII characters in the reference from which you're most likely to come across the *newline* `\n` and the *tabulator* `\t` escape sequences.

However, there's a last item we want to discuss here: arbitrary special characters. The reference page linked just above states that using the `\x` escape sequence it is possible to address characters numerically. But while this only works for the very limited set of ASCII characters and doesn't support Unicode in general, this isn't generally possible for use in LilyPond strings. There are cases where it is possible while others don't work, but it can be said that it is generally not recommended. Instead there are two options to include special characters, apart from the option of directly including the special characters in the input files - if all participating modules support that.

It is possible to encode special characters using their Unicode code point either as hexadecimal or as decimal numbers. Alternatively LilyPond provides a number of ASCII escape sequences which can be made available from within a `\paper` block:

```
\paper {
  #(include-special-characters)
}
```

Both options as well as a list of escape sequences (which are modeled after HTML escape sequences) can be found in LilyPond's documentation

3.5.1.5. Symbols

Symbols are a double-edged thing in Scheme. While seeming completely natural they can cause substantial confusion for beginners.

The Guile reference states that *"Symbols in Scheme are widely used in three ways: as items of discrete data, as lookup keys for alists and hash tables, and to denote variable references."* This sounds pretty complicated, but I hope to make that become clear soon.

On a first level you can see symbols as "names for something". Symbols are quite similar to strings in so far as they are sequences of characters - but written without additional quotation marks. And in some contexts in LilyPond they can even be used interchangeably. But still they *are* something different.

First of all symbols are not self-evaluating. Earlier we learnt about self-evaluating expressions in Scheme, so for example `4` evaluates to the value `4`, and `"Hello"` evaluates to the string with content `Hello`. A symbol on the other hand doesn't evaluate to itself but always denotes something else - it

is a symbol *for something*. When Scheme encounters the symbol `Hello` it will treat it as the reference to a *variable* whose *name* is `Hello` and will then evaluate to the *value* of that variable.

```
guile> 4
4
guile> "Hello"
"Hello"
guile> Hello
ERROR: Unbound variable: Hello
ABORT: (unbound-variable)
```

In this case there is no variable with the name `Hello`, which triggers this error. But earlier we saw how that works when a respective variable exists:

```
guile> red
(1.0 0.0 0.0)
```

`red` is a symbol (defined in LilyPond) that evaluates to the value of a variable which represents a list of three numbers.

Often we need the symbol "itself" to pass along as data. To achieve this we make use of *quoting* - a way to tell Scheme that a symbol does *not* denote something else. There are two equivalent ways to express this:

```
guile> (quote red)
red
guile> 'red
red
```

`quote` is a procedure that takes one argument - a symbol -, and prevents the evaluation of that symbol. This is a somewhat confusing concept, and therefore we have a dedicated section on quoting.

There is much more to symbols than can be said in this short introduction, and we will come back to the topic whenever necessary. In need of more detailed information one can head for the section in the Guile reference.

TODO:
Presumably there are substantial aspects still missing from this page.

3.5.2. Compound Data Types

The data types we have seen so far was a selection of "simple" data types. These "atomic" data types can be seen as building blocks from which larger data types and objects can be composed. In the context of the "data types" chapter we are not going to discuss "objects" in the sense of real-world models but start with compound data types.

Compound data types are types that have more than one atomic member. For example if you imagine a data type representing a "point" in a two-dimensional space it will have two members, namely values on both axes. In Scheme we would express this as a *pair*, which is Scheme's most fundamental compound data type.

We will take a close look at *lists* and *pairs* now, and after that investigate how *custom data types* can look like in Scheme.

3.5.2.1. Lists and Pairs

Lists and *pairs* are the fundamental entities in Scheme programming, which is pretty natural as Scheme is a member of the LISP family of languages whose name is derived from *LIS_t* P_rocessing.

We have already touched lists in the course of discussing expressions, but now we are going to have a much closer look at them. Lists are constructed from pairs, so it's actually the pair that makes the foundation of Scheme's data structures.

There are so many walls one can run into with lists and pairs when one isn't perfectly clear about all the different quotes and backticks, dots and parens and how to apply all those. Therefore it is really well-spent time and energy to get one's head properly around them.

3.5.2.2. Creating Pairs

The basic unit of compound data types in Scheme is the *pair*, an item with two values. Pairs are used very often as properties for graphical score items in LilyPond, so everybody should be familiar with seeing and entering them:

```
{
  \override Beam.positions = #'(2 . 5)
  \once \override DynamicText.extra-offset = #'(-2 . 4)
  c'8 \p b a b
}
```

{% image caption="Pairs as overrides", href="assets/images/pairs-fullsize.png" %} /assets/images/pairs-small.png {% endimage %}

3.5.2.2.1. Writing Pairs as Literals

In this example the `positions` property of the beam and the `extra-offset` of the dynamics have been deliberately changed using pairs. In Scheme a pair as a literal value is written as two arbitrary values, enclosed in parens and separated by a dot. The parens are prepended with the `'` single quote, and in order to tell the parser to interpret the following as Scheme the whole expression is prepended with the # hash. When entering pairs like this - as literal values - there are a few things one should be really clear about.

Pairs are not limited to numbers - as in the previous example - but can hold *any* type of data. It doesn't matter whether there is whitespace between the parens and the values, but it is important that the separating dot is surrounded by whitespace because otherwise the results will be unexpected:

```
guile> '(2 . 3.2)
(2 . 3.2)

guile> '(red . "hello")
(red . "hello")
```

The first example is a pair consisting of an integer and a real number, the second of a symbol and a string. This is something one could stumble over: a few chapters earlier we have seen that `red` in LilyPond refers to a variable of type `color?` and evaluates to a list with three elements. So shouldn't that somehow be reflected here as well?

The secret lies in the leading single quote `'`. This "quotes" the following expression and prevents Scheme from evaluating the enclosed elements, instead they are taken literally. We have touched this already when discussing symbols but we will cover the concept of quoting in depth in a dedicated chapter.

```
guile> '( 1 . 3/4     )
(1 . 3/4)

guile> '(apple .2)
(apple 0.2)
```

```
guile> '(1. 3)
(1.0 3)

guile> '(red. 4)
(red. 4)
```

The first of these examples shows that spaces between the parens and the values are silently ignored. But the other three examples produce unexpected results in so far as when the whitespace around the dot is missing this is interpreted as belonging to one of the values. `.2` is implicitly completed to `0.2`, `1.` to `1.0` (thus converting an integer to a real number), and in case of the symbol the dot simply becomes part of that symbol `red.`. It is important to note that there is no additional dot left in the resulting expression: Scheme has created a *list* now instead of a *pair*. We leave this distinction for now and will get back to it in the next chapter.

3.5.2.2.2. Explicitly Creating Pairs

Directly writing literal pairs is not the only way to create them, in fact it is a shorthand that can be used instead of the "proper" way. Instead pairs can also be created using the procedure `cons`:

```
guile> (cons 1 3/4)
(1 . 3/4)

guile> (cons "Hi" 2.0)
("Hi" . 2.0)

guile> (cons red 5)
((1.0 0.0 0.0) . 5)

guile> (cons 1 2 3)
ERROR: Wrong number of arguments to #<primitive-procedure cons>
ABORT: (wrong-number-of-args)
```

`cons` is a procedure that is applied to two elements and evaluates to the pair consisting of these two elements. It is an error to provide a different number of elements than two, as can be seen in the last example.

Now when looking at the third example you can see that *this* time `red` is actually evaluated and the first element of the pair is the list we already know as the value of `red`. When creating a pair using `cons` the elements can really have arbitrary types. You can even call procedures, as in the following example that calls the procedure `random` returning a psudo-random real number:

```
guile> (cons (random 10.0) 4)
(5.2 . 4)
```

Here we use the expression `(random 10.0)` that applies the `random` procedure to the value `10.0`, in this case returning the random number `5.2`.

```
guile> (cons random 10.0)
(#<primitive-procedure random> . 10.0)
```

This time we passed the "naked" `random` procedure to `cons`. It is *not* invoked but rather stored in the pair as a procedure. Admittedly this is already a somewhat advanced usage but can come in pretty handy, and we want to present examples of the different ways pairs can handle literals, procedures and evaluated procedure applications. Hoping it will help with providing the whole picture we will close this off with a final example:

```
guile> (cons 'random 10.0)
(random . 10.0)
```

As we have seen earlier it is possible to "quote" symbols to prevent their evaluation to something extranous (a procedure in this case), so here we used `random` as a *symbol* to store it in the first element of the pair.

3.5.2.2.3. Closing Thoughts

This chapter started with a familiar example of *pair* usage in LilyPond, followed by a dissection of how pairs can be created in Scheme. Having read through to here you should by now be aware that the overrides from the example expect "a pair", but not necessarily this familiar way of writing them. In fact you can supply *anything* that properly evaluates to a pair of numbers, even procedures that calculate the overrides on-the-fly. To show that this is true we close this chapter with an example where both the beam positions and the extra-offset of the dynamics is determined by the `random` procedure. *(It has to be noted that this is only pseudo random and will look identical for every subsequent compilation.)*

```
{
  \override Beam.positions = #(cons (random 5.0) (random 5.0))
  \once \override DynamicText.extra-offset = #(cons (random 5.0) (random 5.0))
  c'8 \p b a b
}
```

{% image caption="Random pairs as overrides", href="assets/images/pairs-random-fullsize.png" %} / assets/images/pairs-random-small.png {% endimage %}

3.5.2.3. Accessing Pairs

In the previous chapter we have investigated the different aspects of *creating* pairs, now we'll look into retrieving values from pairs. This is not too complicated but it is necessary to have a firm understanding of it before proceeding to *lists*.

3.5.2.3.1. Preparing the Ground

To start off let us create a few pairs for later reuse (so we don't have to re-type them in the REPL), at the same time taking the opportunity to practise the creation of pairs:

```
guile> (define a (cons 1 2))
guile> a
(1 . 2)
```

(As an aside: with `define` *we bind a value to the symbol* `a`*, namely the value the expression* `(cons 1 2)` *evaluates to, which is the pair* `(1 . 2)`*. You may notice that the* `(define ...)` *expression itself does* not *evaluate to anything (as nothing is written to the console), but afterwards* `a` *evaluates to the pair. So later we can simply refer to the pair using* `a`*.)*

```
guile> (define b (cons '(3 . 4) "5"))
guile> b
((3 . 4) . "5")
```

Here we create a pair `(3 . 4)` to become the first element of our pair `b`. You can see that it is possible to *nest* pairs and to use arbitrary data types as the elemennts of a pair.

```
guile> (define c (cons (cons 6 7) "8"))
guile> c
((6 . 7) . "8")
```

Again we create a pair as the first element of `c`. This example shows how an element can be the result of a procedure application, in this case a nested `cons`. In order to read/understand such an

expression one should recall what we learnt about expressions in general and "resolve" it one step at a time: the inner `(cons 6 7)` evaluates to a pair `(6 . 7)`, and this pair is then used as the first parameter to the outer procedure `(cons '(6 . 7) "8")`, which is then bound to the name `c`. As material for later retrieval exercises we will create an even more extreme example of nested pairs:

```
guile> (define d (cons (cons (cons (cons (cons 1 2) 3) 4) 5) 6))
guile> d
(((((1 . 2) . 3) . 4) . 5) . 6)
```

The second element is `6`, the first element is a pair whose second element is `5` whose first element is a pair … The following (non-functional) rendering may be a helpful visualization of the structure:

```
(                          . 6)
 (                    . 5)
  (               . 4)
   (         . 3)
    (1 . 2)
```

Our final definition assigns a *procedure* to the first element and the evaluation of a *procedure application* to the second element of the pair. Presumably you will get a different random number if you try this on your computer, but as it is really pseudo-random you will get the same sequence of numbers whenever you restart your scheme-sandbox.

```
guile> (define e (cons random (random 100.0)))
guile> e
(#<primitive-procedure random> . 74.1503668178218)
```

Now we have five pairs `a` through `e` that we can work with, starting to retrieve the individual items from the pairs.

3.5.2.3.2. Retrieving Elements from Pairs

So we have learnt how to create pairs in different ways, from writing them as a literal to rather complex procedure applications. For writing property overrides in LilyPond this should be sufficient in most cases, but as soon as you want to actually *work* with Scheme it will be necessary to access the individual elements of pairs (and later lists). For this Scheme provides the basic functions `car` and `cdr`.

3.5.2.3.3. Basic Retrieval Procedures

The first element of a pair is retrieved using `car` and the second using `cdr`:

```
guile> a
(1 . 2)

guile> (car a)
1

guile> (cdr a)
2
```

Applying `car` to `a` evaluates to 1, with `cdr` the result is 2. One can also say "The 'car' of 'a' is 1, and its 'cdr' is 2." "cdr" can be spelled out as "could-er" to make it more speakable.

These two procedures are very fundamental to working with Scheme, and you will see them a lot in real code. While the concept is as simple as that it is important to have a really firm understanding of it, as you will see the first complications already in the next two sections.

3.5.2.3.4. Using Procecures Stored in a Pair

We had defined `e` to hold the procedure `random` as its first element, so retrieving this should give us the procedure. Testing in the REPL happens to confirm this assumption:

```
guile> (car e)
#<primitive-procedure random>
```

So if we have a procedure at our disposal, shouldn't it be possible to *use* it, just like a normal procedure, something like `(random 100)`? And in fact this works perfectly:

```
guile> ((car e) 100)
81
```

Now you may wonder about the nested parens here, but dissecting it slowly it should easily become clear. `(car e)` evaluates to the procedure `random`, so the expression `((car e) 100)` first evaluates to `(random 100)`, which is the regular syntax for applying procedures, and this application will eventually evaluate to the (random) value 81.

On a more general level what we see here is that whenever an element of a pair is not of a simple (or "primitive") data type it will be retrieved just as what it *is*, be it an object, a compound data type - or a procedure. We'll have a closer look at this in the next section.

3.5.2.3.5. Nested Retrieval

As we have seen elements of a pair can be pairs themselves:

```
guile> b
((3 . 4) . "5")
guile> (car b)
(3 . 4)
```

In order to retrieve the elements of this pair we can again apply the `car` and `cdr` procedures - to `(car b)`:

```
guile> (car (car b))
3
guile> (cdr (car b))
4
```

We can "serialize" the nested applications by saying that "3 is the car of the car of b" and "4 is the cdr of the car of b".

Scheme provides shortcuts for this type of nested pair retrieval, for the previous examples these would be `caar` and `cdar`. There are numerous variations available that can be looked up in the Guile reference. The meaning of these procedures can be "resolved" by considering each `a` in the name as "the car of" and each `d` as "the cdr of". So `cdar` can be resolved to " the cdr of the car of ", `caddar` would be "the car of | the cdr of | the cdr of | the car of" something. Of course the value that is passed to such a procedure must *have* a corresponding level of nesting, otherwise it will trigger an error:

```
guile> (caar b)
3
guile> (cdar b)
4
guile> (cadr b)
standard input:15:1: In procedure cadr in expression (cadr b):
standard input:15:1: Wrong type (expecting pair): "5"
ABORT: (wrong-type-arg)
```

The first two are the shorthands for the previous applications, but what has gone wrong with the third one? Let's resolve this expression manually to understand the error message. The original value is `((3 . 4) . "5")`, and what we are requesting is "the car of the cdr of" that value. The cdr of the initial value is `"5"`, and applying the `car` procedure to this fails for obvious reasons - as we are told explicitly: In order to retrieve the `car` of something this something has to be a pair, but when our `cadr` reaches the `car` there is only a simple string left from the orginal object.

Now let's finally investigate our multiply nested pair `d`: `(((((1 . 2) . 3) . 4) . 5) . 6)`. Using `car` returns yet another pair etc.:

```
guile> (car d)
((((1 . 2) . 3) . 4) . 5)
guile> (car (car d))
(((1 . 2) . 3) . 4)
```

Nesting cars and cdrs it is possible to retrieve any single element from the nested pair. For example the `4` is the `cdr` of what we have just arrived at, or "the cdr of the car of the car of" the original `d`. The shorthand should therefore be `cdaar`:

```
guile> (cdr (car (car d)))
4
guile> (cdaar d)
4
```

As an exercise you can retrieve each single integer from `d`, both with the nested procedure applications and the shorthands. Below are the solutions but you are strongly encouraged to try it out yourself *before* looking at them.

```
guile> d
(((((1 . 2) . 3) . 4) . 5) . 6)
guile> (cdr d)
6
guile> (cdr (car d))
5
guile> (cdar d)
5
guile> (cdr (car (car d)))
4
guile> (cdaar d)
4
guile> (cdr (car (car (car d))))
3
guile> (cdaaar d)
3
guile> (cdr (car (car (car (car d)))))
2
guile> (cdaaaar d)
ERROR: Unbound variable: cdaaaar
ABORT: (unbound-variable)
guile> (car (car (car (car (car d)))))
1
guile> (caaaaar d)
standard input:17:1: In expression (caaaaar d):
standard input:17:1: Unbound variable: caaaaar
ABORT: (unbound-variable)
```

There are two invocations that fail because Scheme (or rather Guile) doesn't have these defined as shorthands. In these cases you have to apply and nest the regular procedures. However, if you like to get your head around this additional complexity you can consider the folloiwng (closing) example: `caaaaar` is not defined but `caaaar` returns a pair. Therefore you can apply `car` to the result of `caaaar`. And you can even continue on that track, thus nesting the shorthands in arbitrary ways:

```
guile> (car (caaaar d))
1
guile> (caar (caaar d))
1
guile> (caaar (caar d))
1
guile> (caaaar (car d))
1
```

3.5.2.4. Creating Lists

As with pairs lists can be created either as literals or by using a creation procedure, `list`. In its literal form a list is written as a sequence of values separated by spaces. The creation is the same in plain Scheme and LilyPond:

```
guile> '(1 2 3 4)
(1 2 3 4)
guile> (list 1 2 3 4)
(1 2 3 4)
```

```
% top-level variable definitions
myList = #'(1 2 3 4)
myOtherList = #(list 1 2 3 4)
```

Just as with pairs there is the difference between the quoted and the regular syntax, which makes a difference as soon as the list elements are not exlusively self-evaluating anymore:

```
guile> '(red green blue)
(red green blue)
guile> (list red green blue)
((1.0 0.0 0.0) (0.0 1.0 0.0) (0.0 0.0 1.0))
```

In the first, quoted version the symbols are read as literal values whereas in the second version with the list constructor they are evaluated to their individual lists before being added to the outer list. The last expression eventually evaluates to a list consisting of three lists.

Also as with pairs the list elements can be of arbitrary data types, but this can only be achieved using the `list` procedure as otherwise everything will be "quoted". We won't discuss the following example in detail as this would be redundant.

```
guile> (list 'red 12 random (random 12))
(red 12 #<primitive-procedure random> 1)
```

3.5.2.4.1. Symbol Lists in LilyPond

There is a special kind of list that is needed regularly in LilyPond: a list in which all elements are symbols. LilyPond defines the predicate `symbol-list?` for this purpose, and we can check a given list against it:

```
guile> (symbol-list? '(1 2 3 4))
#f
guile> (symbol-list? '(a b c d))
#t
```

In order to simplify input in LilyPond files the parser accepts a practical shorthand notation:

```
% regular Scheme syntax
mySym = #'(red green blue)

% LilyPond-style Symbol list
myOtherSym = this.is.one.symbol.list

% As of 2.19.39 commas may be used instead of dots
myLastSym = red,green,blue
```

`mySym` and `myLastSym` are now identical and would be displayed just like regular lists. Note that there must not be spaces around the commas or dots.

However, there is a caveat when entering symbol lists like that in LilyPond input files, namely the parser must be able to unambiguously identify the symbols. Concretely the elements must avoid

- pitch names
- numbers
- special characters

which are perfectly acceptable in Scheme syntax but have a different meaning for the LilyPond parser.

Enter the following in a LilyPond file and study the resulting (somewhat re-formatted) error messages on the console output:

```
one = #'(this is a symbol-list)
failOne = this.is.a.symbol-list
```

```
error: syntax error,
       unexpected NOTENAME_PITCH,
       expecting UNSIGNED or SCM_IDENTIFIER or SCM_TOKEN or STRING
failOne = this.is.
                  a.symbol-list
```

The output indicates that the offending part is the `a` element, which is interpreted as a `NOTENAME_PITCH` instead of being one of the types the LilyPond parser allows here. If *any* element of the symbol list has the same name as a pitch name *in the currently active document language* the list must be entered using the more extensive Scheme syntax.

```
two = #'(this doesn@t work)
failTwo = this.doesn@t.work
```

```
error: bad expression type
failTwo = this.doesn
                    @t.work
```

Any special characters will make the LilyPond parser fail with a symbol list. However, it is noteable that underscores and hyphens *are* accepted.

TODO: Find out what “bad expression type” actually means.

```
three = #'(one false4 symbol-list)
failThree = one.false4.symbol-list
```

```
error: syntax error, unexpected UNSIGNED
failThree = one.false
                    4.symbol-list
```

While Scheme allows a symbol `false4` LilyPond stumbles over it because it tries to read the combination of characters and subsequent number as pitch and duration of a note - which doesn't make any sense in this context.

```
four = #'(one 4false symbol-list)
failFour = one.4false.symbol-list
#(display failFour)
```

```
error: syntax error, unexpected SCM_TOKEN, expecting '='

#(display failFive)
(one 4)
```

This one is a tricky error message as the input to `failFour` actually causes the LilyPond parser to wreak havoc. The error is raised not during the parsing of `failFour` but at the beginning of the next expression, and therefore it refers to an item that shouldn't be of any interest.. Obviously the parser reads `(one 4)` and then stops making any sense of the input. The error message is rather cryptic in this case as the item that is printed in the message is *not* the item that caused the error. And of course it can be a very different kind of error message depending on what is following in the input file.

However, it is somewhat strange that `4` should be read as a symbol. And indeed, type-checking (which you will only be able to do after the next chapter) reveals the `4` to be an integer. We'll discuss this with the next and final example.

```
four = #'(4 this seems to work)
wrongFour = 4.this.seems.to.work
```

This final assignment works without errors, however, it does *not* produce a symbol list:

```
#(display (symbol-list? wrongFour))
#f
```

Obviously it is possible to enter plain integer numbers in a list with dot notation, but they are then inserted in the list as integers, not symbols.

To conclude, it is a very convenient (and common) shorthand to enter symbol lists using LilyPond's dot (or comma) notation, but it has its issues and difficulties.

3.5.2.5. List Types and Internal Structure

This chapter gives some more in-depth details about how Scheme lists are constructed, and what implications this has on their use. Please note that this is a somewhat advanced topic and you will probably *not* come across it too early in your learning curve with Scheme in LilyPond. However, there *are* occasions where these characteristics can appear at the surface with nastily confusing situations which can best be handled with a proper understanding of the underlying structures.

Maybe it is a good idea to have at least a cursory glance at the chapter, just to know what it is about. Once you run into a corresponding issue you'll know where to search for further information.

3.5.2.5.1. How Scheme Lists Are constructed

In the previous chapter we saw that lists are *entered* as a sequence of elements separated by spaces. Lists are also *displayed* that way, but this is not how they are organized internally.

A list in Scheme is a *pair* whose `cdr` is the rest of the list. The `car` of this rest is the next list element and its `cdr` the further remainder of the list. In fact this structure is also known as *linked list*, where the elements don't know about the list as a whole but are only linked to their subsequent element. As a consequence you can say that any list is also a pair but that a pair is not necessarily a list:

```
guile> (define lst (list 1 2 3 4))
guile> (define pr (cons 5 6))
guile> lst
(1 2 3 4)
guile> p
(5 . 6)
guile> (list? lst)
#t
guile> (list? p)
#f
guile> (pair? lst)
#t
guile> (pair? p)
#t
```

Let's inspect the list `lst` a little closer:

```
guile> (car lst)
1
guile> (cdr lst)
(2 3 4)
```

The `car` of the list is `1` and the `cdr` is the list `(2 3 4)`, the `car` of that remainder is `2` and the `cdr` the list `(3 4)`. The `car` of *this* remainder is `3` and the `cdr` - the list `(4)`.

This last information may be confusing, but on second thought it is quite natural: The `cdr` of a list element is a list, and so the last element has to be a list as well. But what *is* this one-element list? As we defined earlier a list is a pair whose `cdr` is a list. So let us inspect this one-element list individually:

```
guile> (define ll (list 4))
guile> ll
(4)
guile> (car ll)
4
guile> (cdr ll)
()
```

Now we see that the `car` of the last element is the value `4`, while the `cdr` is an empty list. This kind of list - where the last element's `cdr` is an empty list - is called a *proper list* (we'll see improper lists below). One consequence of this can be somewhat confusing: whenever you have to retrieve the last element of a proper list you don't really look for the "last element" but rather "the car of the last element". We will return to this issue in the next chapter.

3.5.2.5.1.1. Constructing Lists As Chained Pairs

If we take our definition literally that a list is a chain of pairs it should be possible to create a list that way as well:

```
guile> (define pl
  (cons 1
    (cons 2
      (cons 3
        (cons 4 '())))))
guile> pl
(1 2 3 4)
```

What happens here is that we define `pl` to be a pair of `1` and a pair of `2` and a pair of `3` and a pair of `4` and an empty list. The same is possible using the literal notation:

```
guile> '(1 . (2 . (3 . (4 . ()))))
(1 2 3 4)
```

3.5.2.5.1.2. Improper Lists

What happens if the last element of a list is *not* a pair with an empty list as `car` but a simple value? This is not hypothetical but easily achievable:

```
guile> '(1 . (2 . (3 . 4)))
(1 2 3 . 4)
```

This is Scheme's syntax for an *improper* list, where the last element's `cdr` is a value instead of a pair. The dot is an indicator of the last element's nature as a pair between the `3` and `4` instead of between the `4` and an invisible trailing element. Such improper lists can also be created using `cons` and written as literals:

```
guile> (cons 1 (cons 2 (cons 3 4)))
(1 2 3 . 4)
guile> '(1 2 3 . 4)
(1 2 3 . 4)
```

3.5.2.5.1.3. Concatenating Lists

Maybe this section is more about *working* with lists, but we insert it here because it provides more insight on how lists are structured internally.

The procedure `append` takes a list and appends another list at its end, as can be seen like this:

```
guile> (define lst (list 1 2 3 4))
guile> lst
(1 2 3 4)
guile> (append lst (list 5 6))
(1 2 3 4 5 6)
```

From the user's perspective we have a list `(1 2 3 4)` and add the elements `5` and `6` to it. But we have to understand this from what we have learnt above. What `append` *really* does is replace the empty list that is the `cdr` of the first list's last element with the second argument, here `(5 6)`. So what originally was `'(4 . ())` has now become `'(4 . (5 . (6 . ())))`. And as this seamlessly integrates with the construction of chained pairs the overall result is a coherent list `(1 2 3 4 5 6)`. Let's see what happens if we don't append a list but instead a pair:

```
guile> (append lst (cons 5 6))
(1 2 3 4 5 . 6)
```

The result is an improper list, and by now we can easily explain why this is the case: we have replaced the trailing empty list with a pair whose `cdr` is *not* a list but instead a plain value.

The same is true if we append a literal value to the list:

```
guile> (append lst 5)
(1 2 3 4 . 5)
```

This time we have directly replaced the `cdr` of the last element with a literal value, with the same effect of turning the list into an improper list.

Finally we check what happens when we try to append anything to the resulting improper list:

```
guile> (define lst2 (append lst 5))
guile> lst2
(1 2 3 4 . 5)
guile> (append lst2 6)
```

```
standard input:10:1: In procedure append in expression (append l2 6):
standard input:10:1: Wrong type argument in position 1 (expecting empty list): 5
ABORT: (wrong-type-arg)
```

This doesn't work, and also here we are now able to explain why: we said that `append` replaces the empty list in the last element's `cdr` with the appended value. But the improper list does *not* have such a trailing empty list, instead the `cdr` is `5`, and therefore `append` can't do its work.

3.5.2.6. Accessing Lists

Accessing lists and retrieving their elements is really similar to handling pairs - which seems quite natural as lists are built from pairs. In this chapter we will discuss the basic procedures to access list elements.

3.5.2.6.1. car/cdr Access

In the previous chapter we have already seen how the `car` and the `cdr` of lists can be retrieved, and the `cadr` (and friends) shorthands are available for lists as well:

```
guile> (define l (list 1 2 3 4))
guile> l
(1 2 3 4)
guile> (car l)
1
guile> (cdr l)
(2 3 4)
guile> (cadr l)
2
guile> (cddr l)
(3 4)
guile> (caddr l)
3
guile> (cdddr l)
(4)
guile> (cadddr l)
4
```

Adding `d`s in the `cXXr` procedure name selects increasingly "right" parts of the list, while using an `a` as the initial letter selects the corresponding *value* at that position.

One point of specific interest is the *last* element. As explained in the previous chapter *any* `cdr` variant retrieves a *list* (at least from a *proper* list), so `(cdddr l)` also returns `(4)`. In order to retrieve *values* from a list on always has to use the `a` as the first letter, equivalent to using "the car of whatever position in the list we are interested in".

As a mental exercise think about what the `cdar` of this list would be and why `(cddddr l)` returns what it returns (if you have read the previous chapter the second question should be clear).

3.5.2.6.2. Other Access Options

Scheme and Guile provide much more convenient ways to handle lists and their elements. However, I think these should better be discussed after you are familiar with more basic concepts. Therefore I have moved the more elaborate access methods to a separate chapter.

3.5.2.7. A Comparison of Pairs and Lists

In an earlier chapter we created a nested pair whose `car`s were again pairs:

```
guile> (define p (cons (cons (cons (cons (cons 1 2) 3) 4) 5) 6))
guile> d
(((((1 . 2) . 3) . 4) . 5) . 6)
```

This definition is remarkably similar to a definition of a list in the previous chapter:

```
guile> (define l (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 '()))))))
guile> l
(1 2 3 4 5 6)
```

Earlier we had visualized the structure of the nested pair in a pseudo-code manner, and now we compare that to the corresponding rendering of the list as chained pairs, and additionally the same for an improper list `(1 2 3 4 5 . 6)`.

```
# Nested pair
(                          . 6)
 (                    . 5)
  (               . 4)
   (         . 3)
    (1 . 2)

# Proper List
(1 .                                  )
     (2 .                            )
          (3 .                      )
               (4 .                )
                    (5 .           )
                         (6 . '())

# Improper List
(1 .                           )
     (2 .                     )
          (3 .               )
               (4 .         )
                    (5 . 6)
```

This visualization and comparison is provided as an opportunity to get a "picture" of the structure of different list/pair-like constructs in Scheme. You can also consider the definitions in the code above, and think about how unwieldy constructs in Scheme can be managed by taking them apart one piece at a time. This is something you should regularly take your time for, then you'll eventually become really familiar with Scheme and its "way of thinking".

3.5.2.8. Vectors

3.5.2.9. Custom Data Types

Now that we've discussed some of the basic data types it is important to understand that Scheme can be substantially extended with arbitrary "data types". As we have learnt earlier Scheme is very open about types, and therefore creating data types in Scheme is not like defining classes of objects with their properties and methods. Instead one writes *predicates* (see data types), basically providing a way to check if a given object matches the criteria for being "of a given type".

There is a large number of predicates available beyond the basic Scheme data types, and it is sometimes difficult to trace where a given data type is originating. Predicates can be defined in

- Scheme itself,
- Guile's Scheme implementation,
- LilyPond, or

- Custom (user/library) code

You may now want to have a look at the extensive list of predicates in LilyPond's documentation. To get our head around the idea of custom data types we will have a closer look at one data type that is defined by LilyPond: `color?`.

3.5.2.9.1. Dissecting a Custom Data Type

Applying a color to a score item is achieved by overriding its `color` property. (For more details about coloring please refer to the respective pages in LilyPond's Learning Manual and the Notation Reference.)

```
\override NoteHead.color = #red
```

The hash sign switches to Scheme, and `red` is given as a symbol referring to a variable. Using the Scheme REPL we can investigate what this variable evaluates to:

```
guile>red
(1.0 0.0 0.0)
```

`red` evaluates to a list of three floating-point numbers. If you know something about colors and guess that the three numbers represent the RGB values within a range of `0` and `1` you are right. If you want you can do more experiments by inspecting other colors in the REPL, e.g. `blue`, `green`, `yellow` or `magenta`.

We can use the predicate `color?` to check if a given value is a color.

```
guile> (color? red)
#t

guile> (color? '(1.0 0.0 0.0))
#t
```

It is not surprising to see that `red` is a `color`, but we also see that we can pass a literal value to the predicate. Experimenting with a number of values we can get closer to understanding what `color?` expects:

```
guile> (color? 'my-symbol)
#f

guile> (color? '(1 0 1))
#t

guile> (color? '(1 0 0.5))
#t

guile> (color? '(2/3 1 0.2))
#t

guile> (color? '(3/2 1 1))
#f

guile> (color? '(0 1 0 1))
#f
```

Obviously a color is a color when it consists of a list of (exactly) three real numbers in the range of `0 =< n =< 1`. The numbers can be written as integers, fractions or reals, as long as they are within the proper range. In a later chapter we will have a closer look at how `color?` is defined, which will confirm this assumption.

The manual suggest another way of specifying colors from the list of X11 colors:

```
\override NoteHead.color = #(x11-color 'LimeGreen)
```

Let's check this as well in the shell:

```
guile> (x11-color 'LimeGreen)
(0.196078431372549 0.803921568627451 0.196078431372549)

guile> (color? (x11-color 'LimeGreen))
#t
```

This time we call a procedure `x11-color` with the symbol `'LimeGreen`, and again it returns a list of three floating point numbers, which "is" a color.

Now that we know what a color *is* we can try out to use custom colors created ad-hoc:

```
{
  \override NoteHead.color = #'(0.7 0.3 0.8)
  c''4 c''
}

{
  \override NoteHead.color = #'(3/2 0.3 1)
  c''4 c''
}
```

Not surprisingly only the first example works as well and colors the noteheads in a nice violet. The second object violates the requirements of `color?` as the first number is greater than 1. As a result LilyPond prints a warning to the console and ignores the override: `warning: type check for 'color' failed; value '(3/2 0.3 1)' must be of type 'color'`.

Obviously it it possible to use anything as a color property that evaluates to something satisfying the `color?` predicate, whether it is a predefined color variable, an ad-hoc list or a call to a custom procedure. At the same time the type check (which is actively performed by LilyPond) acts as a safety net: it prints a warning and tries to continue the compilation, and so it prevents the program to crash upon erroneous user input.

So in this chapter you have learned about the characteristics of data types and predicates. They allow to specify how data has to be formed in order to be successfully processable by the program. Guile makes use of that feature to extend Scheme's functionality, and LilyPond does so as well, so in real-world code you may encounter a large number of different data types.

3.5.3. Three Different Levels of Equality

Comparing the equality of values is a regular task in programming, usually as the base for a decision. However, deciding whether two values are "equal" isn't as trivial as one might think, and different programming languages have different approaches to this question.

Scheme has three different concepts of equality, represented by the three equality operators `eq?` `eqv?` and `equal?`. We will see these three again in various incarnations, as variants to different comparison operators. The exact interpretation of each level of equality is left to the discretion of the "implementation", so everything described here is explicitly valid for Guile Scheme only and may differ from any information you might encounter regarding MIT Scheme or Racket etc.

Objects of different *type* are never equal in Scheme.

3.5.3.1. eq?

`eq?` checks for the *identity* of two objects, and the scope of that identity is very narrow in Scheme. Basically this predicate returns true if and only if the two arguments refer to the *same object in memory*, regardless of their values. This means that for a lot of comparisons `eq?` isn't applicable, but it is by far the most efficient predicate.

Most importantly *symbols* with the same name *are* identical, so `(eq? 'my-symbol 'my-symbol)` will always evaluate to `#t`. Strings with the same content are, on the other hand, *not* equal in the sense of `eq?`: `(eq? "my-string" "my-string")` will always evaluate to `#f`. Consequently you should try to use symbols wherever possible to *name* things, especially keys in association lists.

Booleans of the same value are identical, which can be used in decision processes: `(eq? #t (number? 2.4))` evaluates to `#t` because the evaluation of the `number?` predicate evaluates to `#t`, which is "identical" to the other `#t` argument.

Another object that is unique and can be compared with `eq?` is the end-of-list element `()`: `(eq? (cdr '(1)) (cdr '(2)))` evaluates to `#t` because the `()` exists only once.

Note: Numbers and characters *may* be `eq?` to the numbers and characters with the same content, but - according to the reference - you can't rely on that fact. So depending on the context both is possible: `(eq? 1 1) => #t` or `(eq? 1 1) => #f`.

3.5.3.2. eqv?

Numbers and characters should rather be compared with `eqv?`. This doesn't look for *identity* but for *equivalence* of the compared objects - for numbers and characters. For all other data types `eqv?` behaves the same as `eq?` and should regularly not be used: `(eq? '(1 . 2) '(1 . 2))` evaluates to `#f` even if the pairs have the same content.

3.5.3.3. equal?

For all cases except the ones mentioned above `equal?` should be used - which is the least strict but also the most "expensive" procedure. In the case of compound data types `equal?` checks for equivalent content recursively, walking over the individual elements and comparing their content. So all these expressions evaluate to `#t`:

```
(equal? "a-string" "a-string")
(equal? '(1 . 2) '(1 . 2))
(equal? (list 1 2 3 4) (list 1 2 3 4))
```

TODO: Investigate and discuss the difference between `=` and `eqv?` or **`string=?` and `equal?`.

3.5.4. List Operations

Lists are maybe the most important building blocks in Scheme, and processing lists is fundamental to working with Scheme. Having a better understanding now how lists are organized we will investigate a number of higher-level operations that can be done with lists.

The first topic to be covered is accessing lists and retrieving their elements beyond the basic `car` and `cdr` procedures. After that we'll investigate how to retrieve specific elements from lists and how lists can be modified. Finally we'll see how to process all elements of a list in sequence.

This chapter will *not* cover all list operations comprehensively, as this is the task of the official reference. However, in order to make efficient use of the reference it is necessary to have a good understanding of the concepts, and therefore I will cover a subset of the functionality at a slower pace.

3.5.4.1. Accessing List Elements

In addition to the basic procedures `car` and `cdr` Scheme provides a number of higher-level ways to access the different list items individually.

3.5.4.1.1.1. Number of Elements of a List

Often it is necessary to know the length, respectively the number of elements of a list. This is determined by the `length` procedure:

```
guile> (length '(a b c d))
4
```

Note that this expects a *proper list* to work, applying `length` to an improper list or a pair will result in an error.

3.5.4.1.1.2. Indexed access

It is possible to access the n-th element of a list using the `list-ref` procedure, which is passed first the list and then the requested index as arguments:

```
guile> (list-ref '(a b c d) 3)
d
guile> (list-ref '(a b c d) 4)
standard input:7:1: In procedure list-ref in expression (list-ref (quote #) 4):
standard input:7:1: Argument 2 out of range: 4
ABORT: (out-of-range)
```

There are two things to note about this: The index is "zero-based", that means that the fourth list entry will have the index 3. And it will result in an error when you try to access a non-existent index as in the second example. The highest possible index is *one lower than the length of the list*. So if the list has four elements as in our example the highest index is *3*.

In order to avoid this kind of error you can make use of the `length` procedure, once you have learned about conditionals and iteration constructs later in this book. As an example that works already now you can use it to access the last element of a list:

```
guile>(define lst '(1 2 3 4))
guile>lst
(1 2 3 4)
guile>
  (list-ref
    lst
    (- (length lst) 1))
4
```

In this example we calculate the index of the last element by subtracting one from the length of the list.

3.5.4.1.1.3. first/second Access

Guile defines a number of convenience accessor methods to retrieve list elements by their position specified in English words instead of the number:

```
guile> (define lst '(1 2 3 4 5))
guile> (first lst)
1
guile> (second lst)
2
guile> (fifth lst)
5
```

Such procedures are defined for the first ten elements of a list (i.e. `first` through `tenth`). Note that "first" here really means the first, so `(first lst)` is equivalent to `(list-ref lst 0)`.

In addition there is the `last` procedure that always retrieves the last element of the list, regardless of its index. As `list-ref` these functions implictly retrieve the `car` of an element. So unlike accessing elements through the `car` and `cdr` functions it is not necessary anymore to explicitly unfold the value using `car`. But unexpected results may occur when the list is an improper list. Please investigate the result of the following expressions yourself - if you have difficulties with that please refer to the explanation about list structure.

```
guile> (last '(1 2 3 4 5))
5
guile> (last '(1 2 3 4 . 5))
4
```

3.5.4.1.1.4. Accessing Parts of a List

Sometimes it's necessary to retrieve parts of a list, for example all elements starting with the third or up to the fourth. For this the procedures `list-tail` and `list-head` are provided.

```
guile>(define lst '(1 2 3 4 5 6))
(list-tail lst 2)
(3 4 5 6)
```

`list-tail` takes the list and the starting index as arguments. So in this example the list elements starting with index 2 are retrieved. Colloquially one can also say the arguments specifies the number of elements to be skipped.

```
guile>(define lst '(1 2 3 4 5 6))
(list-head lst 2)
(1 2)
```

With `list-head` the index argument specifies the index *to which but not including* the list elements are retrieved. However, again this can colloquially be expressed much clearer as the "number of retrieved elements".

If an actual sub-list is required the procedures can be stacked/nested:

```
guile> (list-head (list-tail lst 2) 2)
(3 4)
```

First the `list-tail` expression is evaluated, resulting in the list `(3 4 5 6)`, then this is passed to `list-head`.

3.5.4.2. Extending and Reversing Lists

The title of this section is a little bit misleading as the discussed procedures do not actually *modify* the lists but return a modified *copy* of the original list(s). However, they are nevertheless very useful in common programming tasks.

3.5.4.2.1. Appending Lists to Lists

`append` takes (at least) two lists and returns a new list that concatenates them:

```
guile>(define a '(1 2 3))
guile>(define b '(4 5 6))
guile>(define c '(a b c))
guile>(append a c b)
(1 2 3 a b c 4 5 6)
```

Note that the lists defined as `a`, `b` and `c` are not changed through these operations. In order to make use of the resulting list you will have to bind it to something else:

```
guile>(define c (append a b))
guile>c
(1 2 3 4 5 6)
```

3.5.4.2.1.1. Appending a Single Element to a List

Of course it is possible to append single elements to a list, and in fact this is a very common task. However, there is a caveat that can lead to a very typical error:

```
guile> (append '(1 2 3) 4)
(1 2 3 . 4)
```

Appending a single element seems to create an *improper list!* But if we take a step back and consider what actually happens when lists are concatenated it is clearly correct behaviour.

We know that lists in Scheme are chained pairs, with the `car` of each pair holding the data of the element and the `cdr` pointing to the next element's pair. The last element of a proper list is also a pair, with the `cdr` being the *empty list* `()`. What `append` actually does is pointing that trailing empty list to the appended list, making the first element of the second list a natural member of the first. Consequently, when we append the simple value `4` to a list the `cdr` of the last list element *becomes* `4` - making the first list an improper list.

Fortunately the "solution" is very easy - just something one tends to forget most of the time. "`append` takes (at least) two lists" is what I wrote at the top, and you have to take that literally: you need a *list*, even if it consists of only one element. So the proper way would have been to write

```
guile>(append '(1 2 3) '(4))
(1 2 3 4)
```

NOTE: As an exercise you can figure out how to achieve that result by not appending a *list* but a *pair*.

3.5.4.2.2. Reversing Lists

As with the above `reverse` does not change the list itself but returns a new list with the same elements in reverse order. The behaviour is not surprising at all:

```
guile>(reverse '(1 2 3 4))
(4 3 2 1)
```

Apart from simply reverting the order of a complete list `reverse` can be made useful for accessing list elements from the end. For example the second-to-last element of a list with unknown length can be accessed through `(second (reverse '(1 2 3 4)))` (which would evaluate to `3`). The same result could be achieved using `(list-ref '(1 2 3 4 ) (- (length '(1 2 3 4)) 2))`, but apart from needing to refer to the list *twice* this may be semantically less straightforward, and often you'd prefer juggling with reversed lists. As another example you can retrieve all the elements of a list *except the last* through

```
guile> (reverse (cdr (reverse '(1 2 3 4))))
(1 2 3)
```

First we reverse the list, then we apply `cdr` (giving us all elements except the first one (i.e. the last one of the original list)), and finally we reverse the order again. As an exercise you should try achieving the same result using `list-head`.

3.5.4.3. Searching and Filtering Lists

Often it is necessary to choose elements from list based on certain criteria, or in other words: to search and filter lists. As lists are the core of Scheme as a LISP language there are of course readily available tools for that purpose.

3.5.4.3.1. Searching for Elements in a List

To determine if an element is present in a list Scheme provides the triple

- `memq`
- `memv`
- `member`

each representing one of the three equality modes. The functions are called as

```
(<function> <element> <list>)
e.g.
(memv 3 '(1 2 3 4 5))
```

Colloquially you could translate this into "check if `3` is a member of the list `'(1 2 3 4 5)`" - which it is -, but actually the function works somewhat differently. From the introduction to data types you may recall that for that type of question one uses *predicates*, procedures that return `#t` or `#f` according to the result of some test. But predicate names by convention have a trailing question mark, which `memq`, `memv` and `member` do not have. This is not an inconsistency but rather indicates that the function does *not* return `#t` or `#f`. Instead it returns *the remainder of the list, starting with the first matched element, or* `#f` *if the element is not found in the list.* The result of the above example would therefore be the list `(3 4 5)`.

The fact that `member` and friends return either `#f` or a "true value" can be exploited to create elegant solutions in conditionals.

3.5.4.3.2. Filtering Lists

Sometimes it is necessary to produce a list resulting of all elements in a given list that match (or do not match) given criteria. This can be achieved through applying `filter` or `delete` to a list

3.5.4.3.2.1. filter

```
(filter <predicate> <list>)
```

`filter` applies `<predicate>` to each element of `<list>` and creates a new list consisting of each element satisfying the predicate. `<predicate>` can be any procedure taking exactly one argument and returning a value. The predicate is "satisfied" whenever it returns a "true" value, that is anything except `#f`.

I will make this clearer through an example. The easiest case is applying a type predicate, keeping all elements that match a certain type:

```
guile> (filter number? '(1 2 "d" 'e 4 '(2 . 3) 5))
(1 2 4 5)
```

In this case `filter` applies the predicate `number?` to each element of the list `'(1 2 "d" 'e 4 '(2 . 3) 5)` and constructs the resulting list from all numbers in this list. Note that the pair `'(2 . 3)` is *not* a number although it consists of only numbers.

In this basic form `filter` is somewhat limited because it can only be used with procedures accepting a single argument, i.e. "predicates". Although there are other procedures that match this requirement it is rarely useful to use them in a `filter` expression. When interested in things like "all list elements that are numbers greater than 7" or "all list elements are pairs of numbers where the `cdr` is

greater than the `cdr`" you will want to use custom procedures, which you'll learn to write in Defining Procedures.

3.5.4.3.2.2. delete and delete-duplicates

`delete` is somewhat like the opposite of `filter` in so far as it returns a copy of a list with all elements that *do not* match the given criteria. The difference is that the match is not determined through a *predicate* but by comparing the elements with `equal?`.

```
guile>(delete 3 '(1 2 3 4 5 4 3 2))
(1 2 4 5 4 2)
```

The resulting list is the original list with all elements deleted that are "equal" to `3`.

`delete` has its companion procedures `delq` and `delv` which use `eq?` and `eqv?` as comparison predicate.

A related procedure is `delete-duplicates`, which returns a copy of the original list, stripped off any duplicate elements. The comparison is performed using `equal?`, so for example strings with the same content are deleted as well:

```
guile> (delete-duplicates '("a" 2 3 "a" b 3))
("a" 2 3 b)
```

3.5.4.4. Modifying Lists

The distinction between filtering/searching lists and *modifying* them is somewhat blurred, for reasons I will discuss more closely in a minute. You may notice that also the official reference pages about list modification and searching mix entries in a somewhat arbitrary way.

I will not cover all procedures on this page but try to explain the basic behaviour. But I strongly recommend you visit the two reference pages and familiarize yourself with what is available.

3.5.4.4.1. Changing a Single List Element

`list-set!` is the corresponding procedure to `list-ref`: `

```
(list-set! <list> <index> <new-value>)
```

changes the element addressed by the (zero-based) index argument to the new value. The trailing `!` in the name indicates (by convention) that `list-set!` actually modifies the list instead of returning a copy. The expression evalulates to the new value.

```
guile> (define a '(1 2 3 4 5))
guile> (list-set! a 1 'b)
b
guile> a
(1 b 3 4 5)
```

You can see that if the list is bound to a variable the change is also persistent.

3.5.4.4.2. Changing the Remainder of a List

With `list-cdr-set!` you can change the n-th `cdr` of a list to something else, usually another list.

```
guile> (define a '(1 2 3))
guile> (define b '(4 5 6))
guile> (list-cdr-set! a 1 b)
(4 5 6)
guile> a
(1 2 4 5 6)
```

What does happen here exactly? The procedure determines the list element at position `1`, which is the *second* element, `2`. Then it sets the `cdr` of this element to the list `b`, effectively appending the second list to the list head of the first.

The same list could equally have been cosntructed with

```
guile> (append (list-head a 2) b)
(1 2 4 5 6)
```

with the difference that the latter would have returned a *new* list.

3.5.4.5. Iterating Over Lists

Iterating over the elements of a list is an extremely common programming task. However, this is an area where Scheme is quite different from other languages, and its idioms are very elegant - once they are not confusing anymore. It is therefore important to really understand this topic in order to work efficiently without being constantly frustrated.

Most languages approach this iteration in a `for` loop. The basic approach would be counting an index variable and accessing each list element through this index. For example in "classic" C++

```
int v[5] = {4,3,2,1,0};
for (int i=0; i<5; i++)
{
    printf("%d: %d", i, v[i]);
}
```

Of course this works, but the loop construct is semantically unrelated to the actual list iteration because it uses an independent counter variable. Moreover you are responsible yourself for not exceeding the list index. Therefore languages provide a loop construct that is closer to the list, e.g. in Python

```
values = [4, 3, 2, 1, 0]
for v in values:
    print v
```

"Modern" C++ provides an equivalent construction:

```
int v[5] = {4,3,2,1,0};
for (int x : v)
{
    printf("%d", x);
}
```

This approach makes the elements of the list available in the body of the loop. This is closer to the list semantics, and it guarantees that the actual range of list elements is used. However, the *body* of the loop is still unrelated to the list, as you could do *anything* inside.

Scheme's approach is similar to Python's (and the second C++ variant) in actually iterating over the elements of a list passed as argument. But it goes one step further by *applying a procedure* to each element. There are two procedures available, differing in what they evaluate to, `map` and `for-each` which we will discuss in detail. However, as these concepts are mostly useful with custom procedures these discussion is post-poned to a later chapter.

3.5.5. Quoting

A very fundamental concept in Scheme is *quoting*. Unfortunately it is often made unnecessarily confusing for new users. If introduced slowly enough and digested thoroughly the idea isn't that

complicated after all. But instead users are mostly confronted with it through code pasted from some helpful snippet, and when trying to modify that unknown code they are left alone with sloppily placed comments on mailing lists or ready-made corrections.

Quoting is hidden beneath the keywords `quote`, `quasiquote`, `unquote` and `unquote-splicing` and their slightly confusing shorthands `'` (single quote), `` ` `` (backtick), `,` (comma) and `@` (at), combined with picky requirements regarding the placement of parens.

Being thrown back to one's own experimentation and to trying to adapt other's code is a recipe for frustration in this case. Therefore we'll proceed slowly and dissect this valuable item in the Schemer's toolkit step by step.

3.5.5.1. Preventing Evaluation

As we have seen in an earlier chapter Scheme has the concept of self-evaluating expressions, which is applicable for most simple data types: `5` evaluates to `5`, `"foo"` to `"foo"`, and `#t` to `#t`.

However, symbols are different as they are by default names referring to something else and evaluating to that else's value. `red` evaluates to the color definition list `(1.0 0.0 0.0)`, `random` evaluates to a procedure. However, sometimes we need to work with the names themeselves, regardless of some entity the refer to or not. We might want to express something like "hey, we want violet", no matter how violet is constructed as a color and whether it is actually defined.

In order to achieve this we can *quote* any Scheme value to prevent it from being evaluated. This is done using the `quote` procedure which is applied to a single object:

```
guile> red
(1.0 0.0 0.0)

guile> (quote red)
red
```

In this case Scheme doesn't care that there is a variable `red` referring a list of three values and representing the color "red". Scheme will also ignore that there is no color "violet" in

```
guile> violet
ERROR: Unbound variable: violet
ABORT: (unbound-variable)

guile> (quote violet)
violet
```

Quoting can not only be applied to symbols but (as said) to *any* Scheme value, for example procedures:

```
guile> random
#<primitive-procedure random>

guile> (quote random)
random
```

In all these cases we are dealing with names just as names, without any notion of a content the names might be referring to.

3.5.5.1.1. Shorthand Notation

Using `quote` is the explicit way to quote objects, but far more often you will encounter and use the shorthand notation with a prepended single quote.

```
guile> 'red
red

guile> 'violet
violet

guile> 'random
random
```

However, you should still be familiar with at least reading the explicit form, as Scheme may use it to format expressions that you may display for learning or debugging purposes..

3.5.5.2. Creating Quoted Lists and Pairs

3.5.5.2.1. Lists

As we have seen earlier it is not possible to directly write a list object because Scheme will try to apply the first element as a procedure to the remaining elements:

```
guile> (1 2 3)
standard input:98:1: In expression (1 2 3):
standard input:98:1: Wrong type to apply: 1
ABORT: (misc-error)
```

Scheme tries to read `1` as a procedure and apply it to `2` and `3`, which obviously fails. The "wrong type" in the error message refers to `1` being a number and not a procedure. The regular syntax to create that list is to use the `list` procedure in the first place:

```
guile> (list 1 2 3)
(1 2 3)
```

Numbers are self-evaluating, but if we pass symbols to `list` they are evaluated before being added to the list:

```
guile> (list red green blue)
((1.0 0.0 0.0) (0.0 1.0 0.0) (0.0 0.0 1.0))
```

But what if we want to store the symbols as *names*, that is we want the result to evaluate to `(red green blue)`? Of course we can quote the symbols, using either `quote` or the shorthand notation:

```
guile> (list (quote red) (quote green) (quote blue))
(red green blue)

guile> (list 'red 'green 'blue)
(red green blue)
```

This may become quite unwieldy in practice, therefore Scheme provides a shortcut and allows the quoting of the expression as a whole:

```
guile> (quote (red green blue))
(red green blue)
```

Please note that the three elements are surrounded by nested parens: `quote` quotes one single object, and that is the whole list here. It is also possible to use the shorthand notation, and this is what you will likely see and use most:

```
guile> '(red green blue)
(red green blue)
```

Or in LilyPond syntax:

```
myList = #'(red green blue)
```

3.5.5.2.2. Pairs

The same is true for pairs. You can't enter them literally but you have to either use the `cons` procedure or quote them through `quote` or `'`:

```
guile> (1 . 2)
standard input:82:1: In expression (1 . 2):
standard input:82:1: Wrong number of arguments to 1
ABORT: (wrong-number-of-args)

guile> (cons 1 2)
(1 . 2)

guile> (quote (1 . 2))
(1 . 2)

guile> '(1 . 2)
(1 . 2)
```

(Please don't ask about the error message in the first example. Obviously this expression is so fishy that Scheme isn't even able to produce a meaningful error …)

And as with lists and the `list` constructor elements that are not quoted will be evaluated:

```
guile> (cons red random)
((1.0 0.0 0.0) . #<primitive-procedure random>)

guile> (cons red blue)
((1.0 0.0 0.0) 0.0 0.0 1.0)
```

3.5.5.2.3. Digression

But wait a minute, this last one does *not* look like a pair, isn't it?

Well, this is one of these moments where Scheme's syntactical structures can drive you crazy when you haven't *really* dug into the core. Although it isn't the topic of this chapter it seems appropriate to take the opportunity of a practical recap of what we have seen in the chapter about the internal structure of lists.

We can properly retrieve the `car` and `cdr` of this expression:

```
guile> (car (cons red blue))
(1.0 0.0 0.0)
guile> (cdr (cons red blue))
(0.0 0.0 1.0)
```

But should that expression not evaluate to something that looks like

```
((1.0 0.0 0.0) . (0.0 0.0 1.0))
```

?

OK, let's dissect it: we have a pair where both the `car` and the `cdr` are lists. And earlier we have seen a definition of just that: a "pair whose `cdr` is a list". This definition describes - a *list*. So our pair is just a special case: when the `cdr` of a pair happens to be a list the whole expression becomes a list as well. Sounds somewhat strange but obviously doesn't do any harm (if you don't take countless hours of scratching newbies' heads into account …).

We can verify that assumption quite easily. Usually a list is also a pair but a pair is *not* a list:

```
guile> (pair? '(red blue))
#t

guile> (list? '(red . blue))
#f
```

But here we can see that our pair *is* at the same time a list:

```
guile> (list? (cons red random))
#f
guile> (list? (cons red blue))
#t
```

But there's one more thing to it. If we explicitly create a list from red and blue it takes yet another form:

```
guile> (list red blue)
((1.0 0.0 0.0) (0.0 0.0 1.0))
```

What is that? Well, a *proper* list is a list whose last element is a pair with an empty list as its `cdr`. Let's see what the `cdr` of our last (second) list element is:

```
guile> (cdr (list red blue))
((0.0 0.0 1.0))
```

So why are we getting *two* nested paren levels? It gets more and more confusing ... Well, `blue` is a list, so the `cdr` of our initial expression should be a list, isn't it? And as we have seen earlier the last element in a list is not the element itself but a *pair* with the element and an empty list as its elements. So we can check the `car` and the `cdr` of that last expression:

```
guile> (car (cdr (list red blue)))
(0.0 0.0 1.0)
```

So *this* is the "blue" list we'd expect.

```
guile> (cdr (cdr (list red blue)))
()
```

And *this* is the empty list that makes it a "proper list".

So if we try to wrap that up we can say: `(list red blue)` creates a proper list with two elements that are both proper lists as well. `(cons red blue)` on the other hand creates a list whose *first* element is a list ("red") while "blue" is represented as three individual elements.

Being confronted with this kind of stuff can be pretty confusing, not only for the new user. But everything can be stripped down to some very basic fundamental concepts, so don't be frightened but try to dissect things one by one - and ask on the mailing lists for clarifications, and don't hesitate to keep asking until you have fully understood the case.

3.5.5.3. Unquoting

OK, we have seen the quoted and the explicit syntax to create lists and pairs, the difference being that in the `list` approach the elements are evaluated:

```
guile> '(red random 1)
(red random 1)

(list red random 1)
((1.0 0.0 0.0) #<primitive-procedure random> 1)
```

But quite often one needs a list where *some* elements are taken literally while others should be evaluated. Consider the previous example and imagine that with `random` we don't want to refer to the procedure of that name, but instead read it as, say, an option name, e.g. one out of a list of `'random`, `'sorted` and `'weighted`.

The most straightforward approach would be to create the list using `list` and individually quote the element that needs it:

```
guile> (list red (quote random) 1)
((1.0 0.0 0.0) random 1)
guile> (list red 'random 1)
((1.0 0.0 0.0) random 1)
```

But Scheme offers an alternative approach for this that you will encounter quite often:

3.5.5.3.1. Quasiquoting

In addition to `quote` Scheme has the procedure `quasiquote`. On first sight it does the same as `quote`, namely it quotes an object. As with `quote` there is also a shorthand notation available that is usually used in input files, the backtick `` ` ``.

```
guile> (quasiquote (red random 1))
(red random 1)

guile> `(red random 1)
(red random 1)
```

So here the list elements are quoted as well. The difference is that with `quasiquote` it is possible to *unquote* individual elements within the object.

3.5.5.3.2. Unquoting

Unquoting an element, that is, forcing this specific element to be evaluated, is achieved using `unquote` or its shorthand notation, the comma:

```
guile> (quasiquote ((unquote red) random 1))
((1.0 0.0 0.0) random 1)

guile> `(,red random 1)
((1.0 0.0 0.0) random 1)
```

In this example we have (quasi)quoted the expression as a whole but explicitly unquoted the `red` element. You can see that `red` is evaluated while `random` is read as a name.

Note that I have used the written and shorthand notations consistently within each expression, but that is *not* necessary, the shorthand forms can freely be mixed with the verbal ones, like e.g. `(quasiquote (,red random 1))`.

3.5.5.3.3. Unquoting a List

When unquoting an element that happens to be a list this list is inserted into the surrounding list as a single item, such as

```
guile> `(1 2 ,(list 3 4) 5)
(1 2 (3 4) 5)
```

However, there are many occasions where you will want the individual items of that list to be inserted in the resulting list. This can be achieved with the `unquote-splicing` syntax or its shorthand "comma-at" `,@`:

```
guile> `(1 2 ,@(list 3 4) 5)
(1 2 3 4 5)
```

To be more concrete, in a quasiquoted expression `unquote-splicing` takes any Scheme expression that evaluates to a list and inserts the elements individually in the surrounding list.

3.5.5.3.4. Nesting `quasiquote` Levels

It is possible to nest multiple levels of quoting and unquoting, but we won't discuss this in more detail as it is not regularly used in LilyPond. But you may want to keep that fact in mind, in case you encounter a complicated quoted expression and unquoting doesn't seem to do its job.

3.5.6. Association Lists

Now that we've become familiar with lists and pairs we can investigate a special incarnation of them: *association lists*, or *alists*. These are lists that associate *keys* with *values* and allow the retrieval of values by their keys, which is the same behaviour as what *dictionaries* do in other languages such as e.g. Python. Alists are for example used to store all the properties in LilyPond's objects.

Technically a Scheme alist is not some predefined type, but it should be seen the other way round: a list whose elements are pair representing a key-value relationship is considered an association list.

3.5.6.1. Inspecting alist Structure

In order to save some typing we start a session with defining an association list:

```
guile>
(define my-alist
 '(
    (color1 . red)
    (color2 . green)
    (color3 . blue)
  )
)
guile> my-alist
((color1 . red) (color2 . green) (color3 . blue))
```

This is not the commonly used Scheme layout but intends to give a better idea of the structure. Normally the expression would be written in a more condensed manner (note particularly that all the trailing parens are lined up at the end of the last line), for example:

```
guile>
(define my-alist
 '((color1 . red)
   (color2 . green)
   (color3 . blue)))
```

Inside the `define` I created a list with three elements using the `'()` syntax, and each of these elements is a pair associating a name for a color with a concrete color. This is a regular list of pairs, but one can say it is an association list because of that specific structure. As it *is* a regular list you can access its elements with the usual methods applicable to lists and pairs:

```
guile> (car my-alist)
(color1 . red)

guile> (cadr my-alist)
(color2 . green)

guile> (first my-alist)
(color1 . red)
```

```
guile> (cdar my-alist)
red
```

I strongly suggest you take the opportunity to practice by trying to retrieve every single element from this alist, taking specific care about extracting the elements from the last pair.

3.5.6.1.1. Types and Quotes

Of course the keys and values can have arbitrary types, as is the rule with Scheme pairs. But it is considered best practice to use *symbols* as keys, which has some advantages we won't discuss here. And this points us to an issue where we can start to see the practical use case for the different quoting methods we have discussed in the previous chapter.

`my-alist` maps the *name* (symbol) `color1` to the *name* `red`. However, if we make use of such an association list we usually want to map the name `color1` to the *color* `red`. The real-world use case of this might be that we have defined a certain type of thing (e.g. the composer name) to be formatted using "color1" and have the user specify a concrete color for that stylesheet. Or we might highlight editorial additions with a color and want to set that to black when the score is compiled for publication.

So we have to make sure that the *value* parts of the pairs have the right type, which is not possible using the syntax used above. But while all the options are available that we discussed in the *quoting* chapter I will only use the most common idiom here: *quasiquote* the whole list and *unquote* the values (note the backtick in front of the list's parenthesis):

```
guile> (define my-new-alist
`((color1 . ,red)
  (color2 . ,green)
  (color3 . ,blue)))
guile> my-new-alist
((color1 1.0 0.0 0.0) (color2 0.0 1.0 0.0) (color3 0.0 0.0 1.0))
```

If you are wondering why the three pairs look like lists instead of pairs you should go back and check out the "Digression" section in the chapter about creating quoted lists and pairs.

But of course we don't want to only create alists but to actually *do* something with them, which we'll discuss in the following chapters.

As processing alists is so central to working with Scheme it is no wonder that a number of dedicated procedures exist for adding, updating, retrieving and removing entries from alists are available. They are not hard to understand in principle, but in my experience there are two issues one has to be very careful about: the seemingly arbitrary order of arguments that is expected by the different procedures, and the fact that some procedures modify the alists in-place while others just return new copies. This is something one has to learn very thoroughly - or be sure to always look it up when using the procedures.

3.5.6.2. Looking Up Values from Association Lists

Retrieving a value from an alist means looking up the key and returning the corresponding value. In our previous alist `my-new-alist` we want to get the `green` variable for the key `color2`.

As the alists are regular lists we could go ahead and figure out a way to iterate over the list, comparing each element's `car` with the given key and if we find something return the corresponding `cdr`. You are not ready to do that yet, but fortunately this isn't necessary anyway. As association lists are so fundamental that Guile provides a number of specific procedures that can be used directly.

3.5.6.2.1. Guile's alist Retrieval Procedures

Guile's procedures are documented in the reference, but looking at that page might be quite confusing at this moment.

There are two basic forms of procedures, and both come in three variants (and additionally everything is doubled for C and Scheme). The three variants differ in the method they use for determining a matching key, which Guile calls the "level of equality". This is a discussion I would like to spare you for now, and you can keep in mind that as long as you stick to *symbols* as alist keys you should use the "q" variants of the procedures, `assq` and `assq-ref`. If at one point you should need other types as alist keys you have to get familiar with Scheme's concept of equality and Guile's/LilyPond's implementation of it.

3.5.6.2.1.1. Different Return Targets

`assq` and `assq-ref` differ in what they return for the match: `assq` returns the `(key . value)` pair for a match while `assq-ref` returns just the value. Both return `#f` if the requested key is not present in the alist. Please note the annoying detail that the order of arguments is reversed: `assq` expects first the *key* while `assq-ref` wants the *alist* first:

```
guile> (assq 'color2 my-new-alist)
(color2 0.0 1.0 0.0)

guile> (assq-ref my-new-alist 'color2)
(0.0 1.0 0.0)

guile> (assq 'color4 my-new-alist)
#f

guile> (assq-ref my-new-alist 'color4)
#f
```

So which procedure should be used then, or which one in which situation? Generally it is more common that you want to use the *value* rather than the key-value pair, so this seems to indicate using `assq-ref` preferably. But while this is *often* true there is an important caveat: the behaviour with non-existent keys. As seen both procedures return `#f` if the key is not present in the alist, and in our example with the colors this probably doesn't cause any problems. However, the issue becomes crucial when `#f` is a valid value in the alist.

```
guile> (define bool-alist
       '((subdivide . #t)
         (use-color . #f)))

guile> (assq-ref bool-alist 'use-color)
#f

guile> (assq-ref bool-alist 'use-colors)
#f
```

Both invocations return `#f`, but only in one case this refers to the actual *value* of the entry, while in the other it is the result of the key not being present.

Please don't think this is only used to catch typing errors - as this example might suggest. It is very common to process alists where it is not known which keys are present. In such cases you *have* to use `assq` and unfold the resulting pair yourself:

```
guile> (assq 'use-color bool-alist)
(use-color . #f)
```

```
guile> (assq 'use-colors bool-alist)
#f

guile> (cdr (assq 'subdivide bool-alist))
#t
```

The pair with `#f` as its `cdr` indicates an actual *false* value while the plain `#f` refers to a missing key. Unpacking the `cdr` of the result is a minor hassle, but there still is an issue that you can't handle at the moment: when the return value is `#f` (i.e. the key is not present) you can't extract the `cdr` from that (just try out `(cdr #f)`). In order to properly handle the situation you will have to wait a little longer until you have digested conditionals.

3.5.6.2.1.2. Caveat: About the Uniqueness of alist Keys

Both `assq` and `assq-ref` return the value for *the first* occurence of the key in the alist. This is important because Scheme has no inherent way to guarantee that keys in alists are unique. If you think of the fact that alists are regular lists with a specific form then this is pretty clear - how *should* there a way for enforcing uniqueness?

In the next chapter you will see how one can take care of uniqueness when adding entries to an alist, but as long as you can't directly control an alist you have to expect duplicate keys.

```
guile> (define al
'((col1 . 1)
  (col2 . 2)
  (col1 . 3)))
guile> (assq 'col1 al)
(col1 . 1)
```

3.5.6.3. Modifying alists

As seen in the previous chapter Scheme provides commands to retrieve elements from association lists although this could also be done using regular list operations. The same is true for adding and setting elements in alists. While it would surely be a good exercise to go through the process in detail you are still lacking some basics, in particular conditionals, so we're sticking to Scheme's dedicated procedures (which you'll be using in general anyway).

3.5.6.3.1. Adding an Entry

There is one procedure that simply adds an entry to an association list:

```
acons <key> <value> <alist>
```

(as a shortname for "cons an alist"). This creates a pair of the given key and value and "conses" this pair to the alist, returning a new alist with the new key prepended:

```
guile> (define bool-alist
       '((subdivide . #t)
         (use-color . #f)))

guile> bool-alist
((subdivide . #t) (use-color . #f))

guile> (acons 'debug #f bool-alist)
((debug . #f) (subdivide . #t) (use-color . #f))

guile> bool-alist
((subdivide . #t) (use-color . #f))
```

From the last expression you can see that the original alist is not modified by this, instead a copy of the combined alist is returned. So if we want to keep the new alist for further use we have to bind it to some other variable (which we'll see in more detail in Binding) or use the following syntax to assign the result to the original variable:

```
guile> (set! bool-alist
        (acons 'debug #f bool-alist))

guile> bool-alist
((debug . #f) (subdivide . #t) (use-color . #f))
```

As said above `acons` simply prepends a new pair, without checking for existing elements with the same key. When you now try to retrieve the entry for such a duplicate entry `assq` will return the *first* pair with the matching key:

```
guile> (set! bool-alist
        (acons 'subdivide #f bool-alist))
guile> bool-alist
((subdivide . #f) (debug . #f) (subdivide . #t) (use-color . #f))

guile> (assq 'subdivide bool-alist)
(subdivide . #f)
```

As it is rather uncommon to have association lists with non-unique keys Scheme also provides a procedure to *add or update* a list element:

3.5.6.3.2. Adding or Updating List Elements

Scheme provides a set of three procedures to *add or update* entries in an alist. As with the procedures retrieving elements from alists the three variants refer to three levels of equality that are applied for matching the given pair to potentially existing elements. If the keys in the alist are symbols the procedure to use is

```
assq-set! <alist> <key> <value>
```

otherwise you may have to use `assv-set!` or `assoc-set!` and - again - familiarize yourself with the different levels of equality in Scheme. (Again: please not the different order of arguments as compared to `acons`.)

The procedure first checks if the alist already contains an entry with the given key. If it does it replaces the value for that key, so the original alist is updated:

```
guile> (assq-set! bool-alist 'debug #t)
((subdivide . #f) (debug . #t) (subdivide . #t) (use-color . #f))

guile> bool-alist
((subdivide . #f) (debug . #t) (subdivide . #t) (use-color . #f))
```

When the procedure does *not* find the key already present it prepends a new element, just as `acons` does:

```
guile> (assq-set! bool-alist 'color red)
((color 1.0 0.0 0.0) (subdivide . #f) (debug . #t) (subdivide . #t) (use-color . #f))

guile> bool-alist
((subdivide . #f) (debug . #t) (subdivide . #t) (use-color . #f))
```

But please note that - again as with `acons` in this case the original alist is *not* updated and a new copy returned instead. So again you have to self-assign the updated list:

```
guile> (set! bool-alist
        (assq-set! bool-alist 'color red))

guile> bool-alist
((color 1.0 0.0 0.0) (subdivide . #f) (debug . #t) (subdivide . #t) (use-color . #f))
```

More details and examples can be found on the reference page.

3.5.6.3.3. Removing Entries from an alist

The missing third part is how to remove existing elements from alists. Probably you won't be surprised by now that this can be achieved using the procedure

`assq-remove! <alist> <key>`

and its companions `assv-remove!` and `assoc-remove!`. You should understand that what `assq-remove!` actually does is making sure the given key isn't present in the returned alist. So if you give it a non-existent key the procedure simply does nothing, without any further notification:

```
guile> (assq-remove! bool-alist 'subdivide)
((color 0.0 1.0 0.0) (debug . #t) (subdivide . #t) (use-color . #f))

guile> bool-alist
((color 0.0 1.0 0.0) (debug . #t) (subdivide . #t) (use-color . #f))

(assq-remove! bool-alist 'some-other-key)
((color 1.0 0.0 0.0) (debug . #t) (subdivide . #t) (use-color . #f))
```

Please not that *this time* the procedure *does* modify the alist directly.

3.5.7. Binding Variables

One of the bread-and-butter tasks in programming is handling variables. We already have seen predefined and custom variable definitions, but now it is time to have a closer look at the topic.

We have seen that *symbols* can be used as self-evaluating *data* or as *references* to something else: the quoted symbol `'red` simply evaluates to that name while the symbol `red` evaluates to the list `(1.0 0.0 0.0)`.

```
guile> 'red
red

guile> red
(1.0 0.0 0.0)
```

The usual Scheme terminology for the latter case is to say that "the name `red` *evaluates to* that list" and that "the list is *bound* to the variable/name `red`".

In the case of `red` this binding is available because it has been created explicitly somewhere in the LilyPond initialization files, but when we need our own variable name, say `violet` evaluating to `(0.5 0.0 1.0)`, we have to "create the binding" ourselves.

Bindings can be created at top-level or locally, which we'll investigate in the following chapters.

3.5.7.1. Top-level Bindings

Like most programming languages Scheme has the concept of *scope*, which basically means that names are visible and invisible in/from certain places.

Top-level bindings are variable definitions outside of any expression, somewhere in the input files. Bindings created on top-level are globally visible in the LilyPond files, both from and in included files as well. But of course they are only visible *after* they have been created, and it is an error to refer to them earlier in the input file. There is a notable difference that will be discussed in the context of procedure definitions.

In the chapter about including Scheme in LilyPond we have already seen top-level bindings (or "global variables" as they are often named in other languages). As demonstrated there bindings can be created in LilyPond and in Scheme syntax, and both produce equivalent bindings

```
% This is written in a LilyPond file
variableA = "Hello, I'm defined in LilyPond"
#(define variableB "And I'm defined in Scheme")
```

Now we have *bound* two strings to the variable names `variableA` and `variableB`. Structurally the bindings are equivalent and both variables can equally be referred to using Scheme and LilyPond syntax (`\variableA` vs. `#variableA`). The only difference is in the rules for naming the variables, as one will be parsed by LilyPond and the other by Scheme/Guile. LilyPond is rather restricted with regard to identifier naming while Scheme more or less allows everything.
TODO: Reference to LilyPond naming options (including advanced options)
For example the following definition is perfectly valid in Scheme, while the LilyPond definition would fail:

```
#(define f9!^¡ "What is this?")

f9!^¡ = "What is this?"
```

Such a variable will *not* be available through LilyPond's backslash syntax, but you can always access the actual Scheme value using the # hash sign:

```
{
  c'1 \mark #f9!^¡
}
```

3.5.7.1.1. Special Scope in Scheme Modules

As said earlier ariables defined in *LilyPond* files this way are *globally* visible also from or in included files. However, defined in *Scheme* modules there are different rules of visibility or "scope". For now you are far from writing Scheme modules, so I'm only mentioning the fact.

By default, when you bind a variable in a Scheme file using `(define)`, it is only visible from within the same module, respectively file. In order to make it available from outside one has to use `(define-public)` instead. This is an important method to hide elements that should not be accessed directly from outside.

Additionally there is the concept of `(define-session-public)`, which *does* make an element publicly visible, but only for the current compilation out of a set of multiple files that are being processed.

3.5.7.2. Local Bindings

A much more interesting - but also complicated - topic is the *local* binding, which is done with the `let` expression and its relatives. The primary use case for local binding is the reuse of evaluations. When the result of a complex expression is used more than once it is more efficient and gives a cleaner input structure to create a local binding for the result and (re)use that.

Colloquially spoken you can say that `let` allows you to create one or more local variables and then evaluate one or more expressions. But in fact `let` represents exactly *one* expression with some local bindings and an expression *body*. The `let` expression itself evaluates to a value that can then be used externally.

With `let` we finally reach the point where the nested parentheses in Scheme can become pretty daunting, with lots of frightening and confusing error messages in the console. I remember very well how I desparately moved, removed and added random parens to make `let` expressions work, and once I managed to get them compile without errors I had no idea *why* and couldn't make use of the experience for future challenges. This was until I started understanding how consequently the expressions are structured and what each part is actually necessary for, and if you follow me through the next few chapters you will hopefully reach a comfortable level of familiarity pretty soon.

3.5.7.3. Local Binding with `let`

The basic form for local bindings is `let`, which we'll explore in most detail. The companion procedures `let*` and `letrec` can then be shown quite concisely. As an example we'll implement a color damper that damps the RGB components of a color by a given factor. In order to benefit from color highlighting and easier editing we will do that in LilyPond files and not on the Scheme REPL.

The typical use case for `let` expressions is within procedures where we would get the original data from the procedure arguments. But for the sake of our example we simply create a global variable that we can then reference:

```
% Create a pair with a color and a damping factor
#(define props (cons yellow 1.3))
```

`props` is now `((1.0 1.0 0.0) . 1.3)`, a pair with a list for the yellow color and a number for the damping factor. What we need to do with it is pretty simple: just create a new list with three elements where each element is the corresponding element from the `car` of `props` divided by the factor stored in the `cdr` of `props`. In order to later display the value (and to practice) we bind the resulting list to a top-level variable defined in LilyPond syntax:

```
dampedYellow =
#(list
  (/ (first (car props)) (cdr props))
  (/ (second (car props)) (cdr props))
  (/ (third (car props)) (cdr props)))

#(display dampedYellow)
```

which will print `(0.769230769230769 0.769230769230769 0.0)` to the console. (You could also use that variable to override a `color` property of a grob, by the way.)

So what's the problem with that? It's the redundancy of the calls to `car` and `cdr`, three times for each. Each time we need the color or the damping factor we have to access the original variable and extract the data from it. In the case of `car` and `cdr` this is not a real issue but in real-world code one will have expressions that are considerably more complicated to write *and* computationally more expensive.

The solution is to create local bindings of the values of `(car props)` and `(cdr props)` and then refer to these bindings within the body of the expression. This is done with the `let` expression that takes the general form

```
(let (<bindings>) exp1 exp2 ...)
```

or, displayed in a more hierarchical manner:

```
(let
  (
    <binding 1>
    <binding 2>
    ...
  )
  <expression 1>
  <expression 2>
  ...
)
```

The `let` expression has two parts, the *bindings* and the *body*, where each part must have *at least* one entry. The bindings are enclosed by parens as a whole, and each binding has the form `(name value)` where `name` is a symbol (the "name" of the local binding) and `value` any Scheme value. While the value could be of any type including literal values the whole point of it is to bind the results of complex expressions to simple names.

In our example we create two bindings, one for the color and one for the damping factors, which looks like this:

```
(let
  (
    (color (car props))
    (damping (cdr props))
  )
  <expression 1>
  <expression 2>
  ...
)
```

Now we have two "local variables", `color` and `damping` that are visible and accessible from within the *body* of the `let` expression.

This body of a `let` expression is an arbitrary number of expressions that is evaluated in sequence. Note that - different from the bindings - there is *no* extra layer of parens around the body. The value of the *final* expression will become the value of the `let` expression as a whole. In our example we only have *one* expression, the `list` creation. But instead of repeatedly calling `car` and `cdr` we can now use the local variables directly.

As this `list` expression is the only (and therefore last) in the body the whole `let` expression evaluates to the created list, and this value (with the damped color) is bound to the top-level variable:

```
dampedYellowWithLet =
#(let
    (
      (color (car props))
      (damping (cdr props))
    )
    (list
      (/ (first color) damping)
      (/ (second color) damping)
      (/ (third color) damping)
    )
 )
```

```
#(display dampedYellowWithLet)
```

which will print the same result as before.

The indentation of the previous example was pretty unidiomatic and intended to exemplify how we constructed the expression step by step. Usually one would use a more condensed way, for example

```
dampedYellowWithLet =
#(let ((color (car props))
       (damping (cdr props)))
   (list
    (/ (first color) damping)
    (/ (second color) damping)
    (/ (third color) damping)))
```

You may think that this expression is considerably more complex than our initial, direct application of `list`. But as mentioned the `car` and `cdr` expressions are comparably simple, and the "complexity ratio" will change dramatically with more complex expressions.

The nesting and parenthesizing levels in that code look daunting, but indeed they are the result of a completely consequent structure, and (in theory) there's no room for misunderstandings. However, the frustrating truth is that in practice (for the beginner) it *is* a miracle how and where to place the parens, and the error messages aren't really encouraging, to say the least. Therefore I have written a dedicated chapter discussing the typical error conditions we tend to produce.

3.5.7.4. Parenthesizing Errors with `let`

The typical way to mastering `let` expressions is paved with parenthesizing errors, which I can tell from experience. Therefore I think it is a good idea to not only explain how these expressions *should* be structured but to also inspect *wrongly* structured expressions.

This chapter discusses the parenthesizing errors most commonly made with `let` expressions, how to identify and how to fix them. For most of the examples we take the expression from the previous chapter as a reference:

```
dampedYellowWithLet =
#(let ((color (car props))
       (damping (cdr props)))
   (list
    (/ (first color) damping)
    (/ (second color) damping)
    (/ (third color) damping)))
```

3.5.7.4.1.1. Missing Final Closing Parenthesis

Although a decent editor should assist the user with matching parens it *still* happens regularly (for example through copy & paste) that the number of parens in an expression doesn't match. In this example I removed the closing paren of the whole expression:

```
dampedColorWithLet =
#(let ((color (car props))
       (damping (cdr props)))
   (list
    (/ (first color) damping)
    (/ (second color) damping)
    (/ (third color) damping))

.../main.ly:7:2: error: GUILE signaled an error for the expression beginning here
#
```

```
 (let ((color (car props))
error: syntax error, unexpected EVENT_IDENTIFIER
#
 (let ((color (car props))
.../

main.ly:15:1: end of file
```

This error message gives us two indicators for the type of error we made.

The primary hint is the `syntax error, unexpected EVENT_IDENTIFIER` message, which is always triggered when the total number of opening and closing parens in a Scheme expression isn't balanced. The Scheme parser (= GUILE) determines the expression by matching parens, and within that expression parens are regular tokens of the Scheme language. But when the final closing paren is missing Scheme can't make any sense of the *starting* paren - which is why the error message is pointing to the *start* of the expression. In that position Scheme can only understand the `(` as LilyPond's token for starting a slur - and therefore as the "identifier" for a "slur *event*" - which is obviously not correct in this place.

The other indicator is the `end of file`. Line 15:1 happens to the the end of the actual file used to create the example, so LilyPond is complaining about not being able to close all open expressions - which is exactly what we triggered by removing a closing paren.

So the error message clearly indicates that the Scheme expression starting in line 7 is not properly closed, that is at least one closing paren is missing, and your task is to identify the place where the expression *should* be completed. The first approach for a solution should then be to add a closing paren at the end of the expression.

3.5.7.4.1.2. Extra Closing Parenthesis

```
dampedColorWithLet =
#(let ((color (car props))
       (damping (cdr props)))
   (list
    (/ (first color) damping)
    (/ (second color) damping)
    (/ (third color) damping))))
.../main.ly:12:32: error: syntax error, unexpected EVENT_IDENTIFIER
    (/ (third color) damping)))
                               )
```

Again, we have the `unexpected EVENT_IDENTIFIER` message, but this time it is pointing to the *end* of the expression, and it isn't GUILE that "signaled" it. In fact Scheme could successfully read the full expression up to the closing paren and handed responsibility back to the LilyPond parser. Now LilyPond encounters the "end of a slur", which doesn't make any sense in this place.

Of course the solution is simply - and already suggested visually by the layout of the error message - to remove the extra paren.

3.5.7.4.1.3. Missing Paren Closing the Bindings

One very common error can occur *because* the editor can assist the user with paren matching. While figuring out to balance opening and closing parens the user manages to miss one closing paren after the *bindings* and adds another, extra, one at the end so the overall balance of the expression is correct:

```
dampedColorWithLet =
#(let ((color (car props))
       (damping (cdr props))
   (list
    (/ (first color) damping)
    (/ (second color) damping)
    (/ (third color) damping))))
```

```
.../main.ly:7:2: error: GUILE signaled an error for the expression beginning here
#
 (let ((color (car props))
In file ".../main.ly", line 6: Missing expression in
(let ((color (car props))
(damping (cdr props))
(list (/ (first color) damping)
(/ (second color) damping)
(/ (third color) damping)))).
```

(Note that the `(let)` *expression in the error message is printed on one line, I have just wrapped it here for better display.)*

This error can be determined from the `Missing expression in (let ...)` explanation. As written in the previous chapter a `let` expression consists of a *bindings* expression and one or more further expressions that form the *body* of the `let`. If there is missing a closing paren at the end of the bindings the parser won't close that part of the expression properly and integrate the body in it as well. Of course after the closing paren there is no expression left to form the body of the `let`.

3.5.7.4.1.4. Extra Closing Paren After the Bindings

This error occurs as the opposite attempt to the previous one. In order to match the overall number of parens one adds an extra one after the bindings part and correspondingly removes one at the end of the expression.

```
dampedColorWithLet =
#(let ((color (car props))
       (damping (cdr props))))
   (list
    (/ (first color) damping)
    (/ (second color) damping)
    (/ (third color) damping))
```

```
...main.ly:7:2: error: GUILE signaled an error for the expression beginning here
#
 (let ((color (car props))
.../main.ly:9:4: error: syntax error, unexpected EVENT_IDENTIFIER

   (list In file ".../main.ly", line 6: Missing expression in
   (let ((color (car
   props)) (damping (cdr props)))).
```

As a result of this attempt the Scheme expression is already completed after the bindings, and so the remainder starting with `(list` is already back in the LilyPond domain. Therefore the `(` before `list` (Line 9:4) is the "offending" slur for LilyPond's parser - and the remaining lines aren't valid LilyPond input either (actually you could already *see* that from the missing syntax highlighting).

In addition Scheme complains about a missing body because it would expect one or more expressions between the two lost parens in line 8.

3.5.7.4.1.5. Missing Extra Parens Around the Bindings

In the following - correct - example there is only *one* binding in the `let` expression. In such cases the double parens around the bindings part look somewhat strange:

```
dampedColorWithLet =
#(let ((color (car props)))
   (display color))
```

Consequently it is a common error to leave out that seemingly redundant extra layer:

```
dampedColorWithLet =
#(let (color (car props))
   (display color))
```

```
.../main.ly:7:2: error: GUILE signaled an error for the expression beginning here
#
 (let (color (car props))
In file ".../main.ly", line 6: Bad binding color in expression
(let (color (car props)) (display color)).
```

So we have a "bad binding color" here. If dissected properly even that message makes sense.

As seen in the previous chapter the bindings part of a `let` expression has the form

```
(
  (name value)
  (name value)
  ...
)
```

where each `(name value)` expression is a single binding. In the above error example Scheme reads the first opening paren after `let` as the start of the bindings part and expexts a binding in the form `(name value)` to follow. Instead it encounters `color`, which obviously isn't a properly formed binding.

So the "bad binding" error indicates a missing opening paren at the beginning of the bindings part.

3.5.7.5. `let*` and `letrec`

Creating one or more local bindings with `let` is a powerful concept, but there are cases where it is limited by the fact that the bindings are only accessible in the body of the expression.

The easiest way to make the point is an example. For this we will continue to work with the expression from the previous chapters, factoring out yet another set of objects to bindings:

```
dampedYellowWithLet =
#(let ((color (car props))
       (damping (cdr props)))
   (list
    (/ (first color) damping)
    (/ (second color) damping)
    (/ (third color) damping)))
```

3.5.7.5.1.1. `let*`

Suppose we don't want to repeatedly access `color` in the expression body but rather have bindings for the RGB values that we can use like

```
(list
  (/ r damping)
  (/ g damping)
```

```
  (/ b damping)
)
```

Again, in this example case this doesn't make much sense, but when real-world objects are complex it can be a crucial simplification.

We might be tempted to write

```
#(let ((color (car props))
       (r (first color))
       (g (second color))
       (b (third color))
       (damping (cdr props)))
   (list
    (/ r damping)
    (/ g damping)
    (/ b damping)))
```

which seems like it might do the job: create three additional bindings making use of the `color` binding done previously. However, this doesn't work out:

```
.../main.ly:7:2: error: GUILE signaled an error for the expression beginning here
#
 (let ((color (car props))
Unbound variable: color
```

When we try to access `color` (in `(r (first color))`) that binding hasn't been created yet because all the bindings are only available when the bindings part of the expression has been completed.

For this use case there is the alternative `let*`. This works like `let`, with the difference that each binding is accessible immediately, already within the bindings part of the expression. Thus the following works:

```
#(let* ((color (car props))
        (r (first color))
        (g (second color))
        (b (third color))
        (damping (cdr props)))
    (list
     (/ r damping)
     (/ g damping)
     (/ b damping)))
```

In fact the necessity to use the `let*` form is quite regular, and therefore many people tend to write `(let*` before thinking of the actual use case. But while this works always one should consider that the added flexibility always comes at the cost of added computation, and while this may usually be negligible it should be good practice not to use up resources without need.

3.5.7.5.1.2. `letrec`

There is another form of local binding, `letrec`. This works like `let*` with the added convenience that *all* bindings are available for *all* other bindings, regardless of the order. With this expression it is possible to create bindings that are mutually dependent. This isn't required very often, therefore I won't go into more details about it, but you should at least have in mind that this option is available.

3.5.8. Conditionals

All programming languages provide constructs to choose code to execute based on some conditions. These constructs are known as *conditionals*, and Scheme is no exception to this rule. The keywords

provided by Scheme are `if`, `cond` and `case`, and they are accompanied by the *logical operators/ expressions* `and`, `or` and `not`.

There is a conceptual difference to conditionals in other languages, though. Colloquially one would phrase the *if* conditional as "*if* a certain condition is met *then* do the following". In Scheme, however, the conditional is a single expression, and depending on the tested condition this evaluates to one of its subexpressions. We will investigate this closer in the following chapters.

3.5.8.1. `if`

The `if` conditional is the most basic code switch in more or less any programming language. In Scheme it has the general form

```
(if test consequent alternative)
```

If the expression `test` evaluates to "a true value" then the subexpression `consequent` is evaluated, otherwise `alternative`, and the whole `if` expression evaluates to the value of the chosen subexpression. OK, let's clarify this with an example, but first of all we have to understand what "a true value" means.

3.5.8.1.1. "True Values"

Scheme knows the literals `#t` for "true" and `#f` for "false", which are returned by many comparing expressions and particularly all predicates:

```
guile> (> 2 1)
#t

guile> (string=? "Yes" "No")
#f

guile> (number? 1)
#t

guile> (list? "I'm a list?")
#f
```

However, in Guile Scheme *every object except* `#f` is considered to have "a true value". So the following expressions *all* have "a true value":

```
"Foo"
'bar
'(1 . 2)
'()
```

That means that whenever the expression `test` evaluates to *anything except* `#f` then `consequent` is evaluated, otherwise `alternative`.

3.5.8.1.2. Evaluating `if` expressions

So here is the first concrete example

```
guile>
(if (> 2 1)
    "Greater"
    "Smaller")
"Greater"

guile>
(if (> 1 2)
```

```
    "Greater"
    "Smaller")
"Smaller"
```

In the first case the test is the expression `(> 2 1)` which evaluates to `#t` (2 *is* greater than 1), therefore the `consequent` subexpression is evaluated. In this case it is the self-evaluating string `"Greater"`, but it could as well be a complex expression or a symbol referring to a variable. This string then becomes the value of the whole `if` expression, and therefore this is printed as the result.

In the second case the test expression evaluates to `#f`, therefore the whole expression evaluates to `"Smaller"`.

The most important thing to understand here is that the subexpressions *evaluate* to a value and don't necessarily *do* anything, and that the same is true for the whole `if` expression. This is what I meant with the different paradigm: in Scheme an `if` expression should be phrased colloquially as "depending on the result of the test this evaluates to one or the other" instead of "depending on the test do this or that". This is best demonstrated in a local binding (which is also a common use case for conditionals):

```
#(display
  (let* ((rand (random 100))
         (state (if (even? rand)
                    "even"
                    "odd")))
    (format "The random number ~a is ~a" rand state)))
```

We have a `let*` expression at the core of this example. It establishes two bindings, first a random integer and then its "state". The value bound to the `state` name is the result of an `if` expression, namely one of the strings `"even"` or `"odd"`, depending on the result of the application of the `even?` procedure. The *body* of the `let*` expression is the invocation of the `format` procedure which evaluates to a string. Therefore the value of the whole `let*` expression is the value of the `format` expression, which is then passed to the `display` procedure, which prints for example `The random number 61 is odd` to the console.

3.5.8.1.3. Special Cases

3.5.8.1.3.1. Unspecified Values

The example discussed above is the default case for `if` expressions, but there are a number of special cases you should know about - because they can be both confusing and useful. The first topic to discuss is the value of the subexpressions.

Depending on the test one subexpression will be evaluated, and its value becomes the value of the `if` expression. However, expressions do not necessarily evaluate to a value but can also be `<unspecified>`:

```
guile>
(if #t
    (display "true")
    (display "false"))
```

The subexpression `(display "true")` will print something to the console but doesn't evaluate to anything, and consequently the whole expression *also* has an unspecified value.

3.5.8.1.3.2. No alternative expression

The alternative expression can be omitted in an `if` expression, making it `(if test consequent)`. This will work, but when the test fails (i.e. the alternative expression is requested to be evaluated)

the value of the `if` expression will be unspecified. This may be acceptable or not, depending on the context. For example, if the subexpression is used to *do* something instead of *returning* a value there's no problem at all.

3.5.8.2. `cond`

The `cond` conditional is used when there are more than two options to handle. All the considerations about "true values" and the evaluation of subexpressions discussed in the previous chapter about `if` apply here as well, so if anything is unclear please refer to that chapter.

The general form of a `cond` expression is

```
(cond
  (clause1)
  (clause2)
  ...
)
```

Each *clause* starts with a test, and if the test succeeds the clause is evaluated and its value returned. The last clause may have the keyword `else` instead of a test, and if none of the previous tests succeeds this final else clause is evaluated.

3.5.8.2.1. Different Forms of clauses

I just wrote that the clauses are evaluated if their test succeeds, but that's a little bit sloppy. Actually there are three different forms of valid clauses, and the evaluation is different in each. The point was mainly that the first successful test determines which clause is responsible for the return value.

3.5.8.2.1.1. The Most Common Form

The most common form for each clause is

```
(test exp1 exp2 ...)
```

In this case one ore more expressions are evaluated and the value of the last expression is returned as the value of the `cond` expression:

```
#(display
  (let ((rand (random 100)))
    (cond
     ((> 50 rand)
      (display rand)
      (newline)
      "Random number is greater than 50")
     ((< 50 rand)
     (display rand)
     (newline)
      "Random number is smaller than 50")
     (else
      (display rand)
      (newline)
      "Random number equals 50"))))
```

First we generate a random integer and locally bind it to `rand`. In the `cond` expression we have three cases, expressed by two tests and an else clause. In each clause three expressions are evaluated, and the third - a literal string - is returned as the value of the `cond` expression, which is passed along as the value of the `let` expression as well. Note that the `cond` expression itself is surrounded by parens as well as each individual clause, but the expressions *within* the clauses are not.

3.5.8.2.1.2. Test Only

Another form for the clauses is

```
(test)
```

If a clause is expressed in this way a successful test will directly return its value, without evaluating further expressions. This is where the concept of "true values" comes into play: if the test returns a value that is considered "true" we can directly use it. In the chapter about retrieval from alists we learned about the `assq` procedure that will return either a key-value pair from an association list or `#f` if the given key is not present in the alist. This way we can directly return the resulting entry or pass control along to the next clause. The following example creates an alist of colors and then tries to retrieve a color through a `cond` expression (in real life we would get the alist from "somewhere" so we don't know which keys it contains):

```
colors =
#`((col-red . ,red)
   (col-blue . ,blue)
   (col-yellow . ,yellow))

#(display
  (cond
   ((assq 'col-lime colors))
   ((assq 'col-darkblue colors))
   ((assq 'col-red colors))
   (else `(col-black . ,black))))
```

`assq` will return `#f` for the first two tests because the given keys are not in the `colors` alist. However, the third test gives a match with the `'col-red` key, therefore the `assq` expression evaluates to the pair `(col-red 1.0 0.0 0.0)`. Since this is a "true" value it is returned as the value of the `cond` expression and consequently printed to the console. If none of the tests would succeed the `else` clause would create a pair with the same structure as the pairs returned by the other clauses, so anyone *using* the return value would surely get valid data.

3.5.8.2.1.3. Apply a Procedure to the Test Result

A final form for the clauses is

```
(test => proc)
```

In this case `proc` should be a procedure that takes exactly one argument. If the test returns a true value `proc` will be applied to this result. We can use this form to improve our previous example so that it only returns the actual color part of the alist entry:

```
#(display
  (cond
   ((assq 'col-lime colors) => cdr)
   ((assq 'col-darkblue colors) => cdr)
   ((assq 'col-red colors) => cdr)
   (else black)))
```

We perform the same test as before, but when it returns the pair this pair will be passed to the procedure `cdr` which will directly extract the second part of the pair. The `else` clause has been adjusted accordingly.

This form is actually a very nice form of "syntactic sugar" because it greatly simplifies that task. Of course we could get the same result - the color part extracted from the pair - with the other forms as

well, but in an unnecessarily complicated way. The relevant clause could for example be written like this:

```
((assq 'col-red colors)
 (cdr (assq 'col-red colors)))
```

Once we know that we're at the right key we can retrieve the value again and pass it along to `car`, which seems pretty inefficient. So we can avoid using `assq` twice by hooking in a local binding:

```
((let ((result (assq 'col-red colors)))
   (if result
       (cdr result)
       #f)))
```

Basically this makes use of the previous form, as the `let` expression evaluates to either the color or to `#f`. Apart from that hint I leave it to you to dissect this expression as an exercise to repeat the topic of local binding. But honestly, `((assq 'col-red colors) => cdr)` is much more elegant, isn't it?

3.5.8.3. Logical Operators: `not/and/or`

Often conditional switches are more complex than a simple true/false decision. Sometimes it is more convenient to ask if something is *false* than if it's *true*, and regularly multiple conditions have to be considered in relation. To achieve this Scheme offers the procedures `not`, `and` and `or`.

3.5.8.3.1.1. `not`

`not` is basically the inversion of a test: it returns `#t` if the expression it tests evaluates to `#f` and to `#f` in all other cases.

```
guile> (not #f)
#t

guile> (not #t)
#f

guile> (not '(1 2 3))
#f

guile> (not (< 2 1))
#t
```

The first two expressions are clear, `not` simply inverts the boolean literals. The third example shows that *any* object has a "true value" and is inverted to `#f`. The final example demonstrates that the value passed to `not` can (of course) be a complete expression which in this case evaluates to `#f`, which is then inverted by the `not`.

3.5.8.3.1.2. `and`

`and` takes any number of subexpressions and evaluates them one after another until any one evaluates to `#f`. In this case the `and` expression evaluates to `#f` as well. If *all* subexpressions evaluate to a true value the `and` returns the value of the last subexpression. This is an important point: `and` does *not* evaluate to `#t` or `#f` as one might expect, instead it evaluates to `#f` or an arbitrary value. You have to take that into account when you need to *use* that value:

```
guile>
(and
  #t
  (> 2 1)
```

```
  (odd? 2)
  "Hi")
#f
```

This expression holds four subexpressions that are tested sequentially:

- `#t` is obviously a true value
- `(> 2 1)` evaluates to `#t` as 2 *is* greater than 1
- `(odd? 2)` evaluates to `#f` as 2 is *not* an odd number
- "Hi" *would* be considered a true value, but it isn't tested anymore because `and` has exited already upon the previous subexpression.

```
guile>
(and
  '(1 . 2)
  (list? '())
  #t
  '(1 2 3))
(1 2 3)
```

In this `and` expression *all* subexpressions have true values, therefore the whole expression evaluates to the value of the last subexpression, the list `(1 2 3)`. So this is where you must *not* expect `#t` but rather *any* true value.

3.5.8.3.1.3. `or`

`or` behaves very similarly to `and`, except that it returns a subexpression's value as soon as that returns a true value. Only if none of the subexpressions return true values the `or` expression returns `#f`. And as with `and` you have to expect "a true value" and not `#t` for the successful subexpression.

In the case of `or` this can be used very straightforwardly to provide fallthrough values: check for a number of tests, and if all fail return the fallback value as the last subexpression. For example we can easily rewrite the test from the previous chapter using `or`:

```
colors =
#`((col-red . ,red)
   (col-blue . ,blue)
   (col-yellow . ,yellow))

#(display
  (or
   (assq 'col-lime colors)
   (assq 'col-darkblue colors)
   (assq 'col-red colors)
   (cons 'black black)))
```

3.5.8.3.2. Nesting of Logical expressions

When more than one condition have to be nested dealing with "operator precedence" is a confusing issue in many languages: which conditionals are evaluated first, do we therefore have to group them with brackets, etc.? Scheme's approach to this topic is very straightforward, and once you get the fundamental idea it is no magic anymore: Each conditional expression tests one single value, and that value can be the result of another logic expression. Period. From there you can create arbitrary levels of nesting.

What does the expression `(not (and #t #f))` return and why? We have a `not` which will invert the boolean state of the value it is applied to. That value is the expression `(and #t #f)`, which will

evaluate to `#f`. So the first evaluation step is to evaluate the inner expression, which leads to `(not #f)`, which eventually results in `#t`.

Let's inspect a more complex expression:

```
(or (not (> 2 1)) (and #t '() (> 4 5)) "Hehe")
```

This is an `or` expression with three subexpressions: `(not (> 2 1))`, `(and #t '() (> 4 5))` and `"Hehe"`. `or` will evaluate each of these subexpressions from left to right, and once *any* subexpression eavaluates to anything other than `#f` it will return that subexpression's value. So let's retrace that one by one:

```
(not (> 2 1))
(not    #t  )
#f
```

The first subexpression evaluates to `#f` so we have to continue. The second subexpression is an `and` expression which itself has multiple subexpressions: `#t`, `'()` and `(> 4 5)`:

```
#t
#t ; a true value

'()
() ; a true value

(> 4 5)
#f
```

The first two subexpressions of the `and` have true values, but the last one is `#f`, therefore the whole `and` subexpression evaluates to `#f`, and we have to continue with the last subexpression, `"Hehe"`. This subexpression has a true value, so finally our `or` returns that one and can be considered successful.

As a conclusion, nested logical expressions in Scheme are just like any other nested expressions: they have to be evaluated one by one, from inside to outside and in the case of `and` and `or` from left to right, and so everything can be resolved unambiguously.

3.5.9. Defining Procedures

Procedures are the driving force of programming languages because that's where the static data is actually processed. So far we have used a number of procedures that are provided by Scheme or have been defined by Guile or LilyPond. Now we're going to "roll our own", and that's about where the fun starts (i.e. where we start being able to achieve something useful).

Procedures are created using the `lambda` expression, that is: a `lambda` expression *evaluates to a procedure*. At the core of things even the fundamental language features are expressed as evaluating expressions! This also means that procedures are *values* just like every other value in Scheme. So procedures can be used like any other value, e.g. bound to different names, stored in pairs or whatever. We will discuss this in more detail in the following chapters.

There are different ways `lambda`-generated procedures handle arguments, which is important enough to warrant a dedicated chapter. Some words have to be spent on the binding of procedures to names, which is how procedures can actually be made useful. And finally we will have a closer look at a specific type of procedures: predicates. The main use of this chapter is to get some practise with writing procedures.

3.5.9.1. The `lambda` Expression

As said in the previous introduction `lambda` is an expression that creates - or *evaluates to* - a procedure. It has the general form

```
(lambda <formals> <expressions>)
```

`<formals>` is where the *arguments* are specified that the procedure will be applied to. There are three forms for this specification, and I will discuss the differences in the next chapter. In this chapter we're using the most common form.

`<expressions>` is an arbitrary number of expressions that is evaluated in sequence. As usual the value of the last expression will become the value of the procedure as a whole. But let's consider this with an example:

3.5.9.1.1. Creating a procedure

```
guile> (lambda (x) (+ x x))
#<procedure #f (x)>
```

The printed result (remember that the Scheme console immediately prints the *value* of the expression typed in) gives us three informations: first that it is a *procedure* that we've created, second that it does *not* have a name (the `#f`) and finally the expected argument list. What it *doesn't* tell us is how the procedure creation works. In order to investigate this we reformat the expression:

```
(lambda
  (x)
  (+ x x)
)
```

The first argument to `lambda` is `(x)`. This tells the parser that the procedure will accept exactly one argument, and this argument will be visible by the name of `x` in the body of the procedure. When the formals are written as a list like here then each element of the list represents the name of one actual parameter the procedure expects.

The body of the procedure consists of the single expression `(+ x x)` which simply takes the argument `x` and adds it to itself. The result of this "duplication" will become the value of the whole procedure.

3.5.9.1.1.1. Parameter types

Here we can see something in action that has been mentioned much earlier in the introduction to data types: Scheme does *not* impose any restriction on the data type that is passed to the procedure, in fact a value of arbitrary type may be locally *bound* to the name `x`. But as we use `x` in the expression `(+ x x)` it is clear that only types can be accepted which the `+` operation can be applied to. Scheme doesn't "guard" procedures against unsuitable parameters through type-checking, which has its pros and cons. The problem can be that possible errors occur within the procedure and not at its interface, which can make them harder to pinpoint. On the other hand this opens a lot of potential for "polymorphism", that is the possibility to write a single interface that behaves differently depending on the type of arguments that are passed into it. We will discuss this aspect in a later chapter.

3.5.9.1.2. *Using* the Procedure

Now we have created a procedure, but it doesn't *do* anything yet, so how can we make use of it? Correct, by *applying* it. Our expression *is* a procedure expecting one argument, so we can use it like one: `(procedure arg1)`. Usually when applying a procedure we refer to it by its *name*, but that

name doesn't do anything else than *evaluating to* the procedure itself, so for Scheme it doesn't make a difference if we use the name or its definition:

```
guile>
(
 (lambda (x) (+ x x))
 12
)
24
```

We have an enclosing pair of parens to denote the *procedure application*, then the definition of the procedure in the first position, followed by a number as the single argument. The expression correctly evaluates to 24, which corresponds to 12 + 12. You should clearly see that this whole `lambda` expression is in the place where we'd normally place a procedure name, like

```
guile>
(
 random
 12
)
7
```

Of course it rarely makes sense to create a procedure just for a single application, but for now we'll stick to that approach and dedicate a full chapter to the different ways of binding and reusing procedures.

3.5.9.1.3. Multiple Paramters and Expressions

Our first procedure accepted a single argument, and its body also consisted of a single expression. But of course multiple arguments and epxressions can be handled:

```
guile>
(lambda (x y)
  (display (format "X: ~a\n" x))
  (display (format "Y: ~a\n" y))
  (+ x y))
#<procedure #f (x y)>
```

This procedure will accept two parameters, which will be visible by the names `x` and `y` in the procedure body. This time the body evaluates three expressions in sequence: the first two expressions print the input arguments to the console while the third and last one evaluates the sum of the parameters and returns that as the whole procedure's value.

We can apply this procedure the same way as the previous one, although it starts to get awkward doing that in the Scheme REPL that doesn't forgive any typing errors:

```
guile>
((lambda (x y)
    (display (format "X: ~a\n" x))
    (display (format "Y: ~a\n" y))
    (+ x y))
  9
  12)
X: 9
Y: 12
21
```

In the last three lines we can see the printout from the `display` procedure and finally the *value* of the expression. *(You may make a mental note of the characteristic double paren at the opening of this expression.)*

3.5.9.2. Alternative `lambda` Signatures

In the previous chapter we discussed the creation of procedures with the signature

```
(lambda (<var> ...) <exp1> <exp2> ...)
```

In this each `<var>` represents a single argument that is expected by and then available within the procedure. But this isn't the only form of specifying a procedure's *signature* in Scheme, there are two other options available, allowing for more flexibility.

The following form allows to pass an arbitrary number of actual arguments to a procedure:

```
(lambda <var> <exp1> <exp2> ...)
```

All arguments that follow the `lambda` expression are then wrapped into a list that is bound to the name `<var>` within the procedure body:

```
guile>
((lambda var-list
    (display (car var-list))
    (newline)
    (length var-list))
 'one 'two 'three 'four)
one
4
```

The four arguments are wrapped in the list `'(one two three four)` which is available as `var-list` in the procedure. This way we can handle arbitrary numbers of arguments within a function. The first expressions in the procedure body print the first element of the argument list while the last expression determines the value of the whole expression, which is then printed on the last line of output.

Please note that a) when you pass a single argument to such a procedure you *still* have to unpack it from the list, and b) there is no extra pair of parens around the variable declaration or the procedure body - it's basically `lambda` with an arbitrary number of expressions, whose first represents the name to which the argument list will be bound.

A third form is actually a hybrid of the two others, accepting both a fixed number of named arguments plus a list of unnamed remaining ones.

```
(lambda (<var-1> ... <var-n> . <var-last>) <exp1> <exp2> ...)
```

The "formals" are given as an improper list here, while the starting entries `<var-1>` through `<var-n>` represent individual actual arguments, while `<var-last>` after the dot collects all remaining arguments in a list:

```
guile>
((lambda (x y . z)
    (* (+ x y)
       (length z)))
    1 2 3 4 5 6)
12
```

Of course the procedure is pretty useless, but it shows how the signature works. We have two actual arguments, `x` and `y`, which are taken from the first two arguments passed to the list: `x = 1` and `y = 2`. The remaining arguments are wrapped to the list `'(3 4 5 6)` and bound to the name of `z`. The body adds `x` and `y` (= 3) and multiplies that with the number of remaining arguments (4).

As said earlier it will rarely make sense to create a procedure using `lambda` for a single application as seen so far. I think each of these examples would have been realized simpler with "normal" expressions. Procedures are getting interesting when they are *bound* to a name and made *reusable*.

3.5.9.3. Binding Procedures

Procedures are *objects* like everything else in Scheme, and therefore procedures can be *bound* to names just like any other variable. Not all programming languages support this handling of procedures as first-class functions. This typical concept of *functional programming* languages allows to pass functions as arguments to and return them from functions. And just as with other objects we can bind procedures locally or globally.

3.5.9.3.1. Top-level Binding of Procedures

When a procedure has been bound to a global name it can be used from anywhere in the program, just like the built-in procedures or those provided by LilyPond. Basically we do exactly the same as with other top-level bindings of variables:

```
(define <name> <object>)
```

with the `<object>` being the `lambda` expression:

```
guile>
(define my-proc
  (lambda (x y)
     (+ x y)))
guile> my-proc
#<procedure my-proc (x y)>
```

The `lambda` expression creates a procedure adding its two arguments, and this procedure is now bound to the (global) name `my-proc`. So from now on we can *use* `my-proc` just like any other procedure (including errors …):

```
guile> (my-proc 2 3)
5
guile> (my-proc 2)
ERROR: Wrong number of arguments to #<procedure my-proc (x y)>
ABORT: (wrong-number-of-args)
```

3.5.9.3.1.1. Alternative Syntax for Top-level Binding

The above example was the very "literal" binding of a `lambda` expression to a name, and this can be done with either of the three forms of `lambda`. But there's also a shorthand notation that is equivalent but that you'll be likely to encounter (and use) more often.

The previous example could equally have been written as

```
guile>
(define (my-proc x y)
  (+ x y))
guile> my-proc
#<procedure my-proc (x y)>
guile> (my-proc 2 3)
5
```

The general form for this is that the following are equivalent:

```
(define <var-0> (lambda (<var-1> ... <var-n>) <exp1> ...))
(define (<var-0> <var-1> ... <var-n>) <exp1> ...)
```

<var-0> will be the name to which the procedure is bound while the remaining vars represent the actual arguments.

There are shorthands for the other two forms of lambda expressions as well, but I'll just mention them shortly, as the above seems to be the most common form by far.

The form accepting an arbitrary list of arguments as its corresponding shortcut like this:

```
(define <var-0> (lambda <var-1> <exp1> ...))
(define (<var-0> . <var-1>) <exp1> ...)

guile>
guile> (define
(my-name . args)
(length args))
guile> my-name
#<procedure my-name args>
guile> (my-name 'a 'b 'c)
3
```

Finally, the hybrid form looks like this:

```
(define <var-0> (lambda <var-1> ... <var-n> . <var-rest>) <exp1-> ...)
(define (<var-0> <var-1> ... <var-n> . <var-rest>) <exp1> ...)

guile>
(define
  (my-name arg-1 arg-2 . arg-rest)
  (length arg-rest))
guile> my-name
#<procedure my-name (arg-1 arg-2 . arg-rest)>
guile> (my-name 1 2 3 4 5)
3
```

3.5.9.3.2. Local Binding of Procedures

We can bind procedures in a let block to reuse not only *values* but actual *functionality*. Basically it is all the same as what we've seen about lambda expressions and variable bindings, so I think we're at a point where I don't have to go into that much depth anymore. However, we can take the opportunity to practise and demonstrate a few other things in context instead. What we can do for the first time is using data that has actually been passed as *arguments*. We will create a named procedure that accepts two values, a color and a damping value to provide shaded versions of colors that can be applied as LilyPond overrides.

```
#(define damp-color
   (lambda (color damping)
     (let ((damp-element
            (lambda (elem)
              (/ (elem color) damping))))
       (list
        (damp-element first)
        (damp-element second)
        (damp-element third)))))
```

```
{
  c'
  \override NoteHead.color = #(damp-color red 1.2)
  d'
  \override NoteHead.color = #(damp-color blue 2)
  e'
}
```

The procedure `damp-color` is created accepting two arguments, the color `color` and the factor `damping`. Inside this procedure we create a local binding with `let`, binding a `lambda` expression to the name `damp-element`. In the body of the `let` expression we create a list with three elements that make use of this locally bound procedure. This list will be the value of the `let` expression, and as this is the last expression in the procedure it will propagate to become the value of the procedure as well. Finally we use that procedure to produce overrides in the LilyPond domain.

There is one more "feature" hidden in this example (step back a moment and try to discover it yourself). When we invoke the local procedure we pass it `first`, `second` and `third` as arguments, which seems somewhat natural - but what *are* these?

```
guile> first
#<primitive-procedure car>
guile> second
#<primitive-procedure cadr>
guile> third
#<primitive-procedure caddr>
```

`first`, `second` and `third` are procedures, actually they are shorthands to built-in list accessor procedures. So what reaches the local procedure as `elem` is not actually a *value* but a *procedure*. This is what I referred to as *first class functions* at the beginning of this chapter. Inside the local procedure we are doing `(elem color)`, that is: we take the procedure that is not hard-coded but passed in as an argument and apply it to `color`, which is a value that we have obtained as the argument to the outer procedure.

3.5.9.4. Defining Predicates

In the discussion of custom data types we discussed the variety of types in Scheme is extended using *predicates*. Instead of defining "classes" or "prototypes" and instantiate objects as having a certain type predicates are used to determine if a given expression/value matches certain criteria. Generically speaking a predicate is a procedure expecting one argument that evaluates to `#t` or `#f` depending on criteria specific to the requested type. In this chapter we get back to that topic as an exercise that will help us get a better understanding of procedure definitions.

In that earlier chapter we introduced the `color?` predicate. Now we're in the position to investigate its actual definition which can be found in the file `scm/output-lib.scm` within LilyPond's installation directory:

```
(define-public (color? x)
  (and (list? x)
       (= 3 (length x))
       (every number? x)
       (every (lambda (y) (<= 0 y 1)) x)))
```

We define a procedure with the name of `color?` and one argument `x`. Appending a question mark to the name is the Scheme convention for predicates. (And it is created with `define-public` because

`output-lib.scm` is a Scheme *module*, and earlier I told you that definitions have to be explicitly made public within modules.)

The body of the expression is one single `and` expression, so the procedure will evaluate to a true value if all of the sub-conditions are met. If on the other hand *any* single subexpression evaluates to `#f` the predicate will also evaluate to `#f`. The (four) conditions that a value has to meet in order to be a "color" are:

- `(list? x)`
  The value has to be a *list*
- `(= 3 (length x))`
  This list must have exactly three elements (representing the red, green and blue components)
- `(every number? x)`
  All three elements must be (real) numbers
- `(every (lambda (y) (<= 0 y 1)) x)`

The last condition should be inspected more closely. `every` is like `and` but with lists. **TODO**: *Reference to subchapter of "list operations"*: it applies a procedure to a list of arguments, one after another, and returns either `#f` or the value of the application to the last list element. But what *is* the procedure that is applied here? it's a `lambda` expression, in other words: an unnamed local procedure:

```
(lambda (y) (<= 0 y 1))
```

which `every` will apply to each element of the `x` list. The local procedure expects a single argument `y` and will check if it is a number between (including) 0 and 1. This `<=` expression expects numbers and would trigger errors otherwise, but from the previous subexpression in the `and` we know that it *is* a number. `<=` will evaluate to `#t` or `#f` and not to an arbitrary value, so the `every` expression will do so as well - it is not possible that any other "true value" than `#t` will be the outcome. Therefore finally the value of the whole predicate will be either `#t` or `#f`. This is the specific requirement when writing predicates: they have to return real `#t` and not just true values.

3.5.9.4.1. Practising With Predicates

To get a better understanding of predicates and types let's write a few predicates as an exercise and investigate some characteristics.

3.5.9.4.1.1. Specifying Type More Narrowly

start with something really simple: checking for a positive integer number:

```
#(define (positive-integer? x)
   (and (integer? x)
        (> x 0)))
```

Again we have an `and` expression in the body, this time we first check if `x` is an integer number and then if it's greater than zero. Now let's see a somewhat more involved predicate, checking if a color is "reddish" (which we define as the "red" component being stronger than the sum of the "green" and "blue" parts):

```
#(define (reddish? col)
   (and (color? col)
        (>= (first col)
            (+ (second col) (third col)))))

#(display (reddish? red))
% => #t
```

```
#(display (reddish? blue))
% => #f
#(display (reddish? (list 0.7 0.35 0.4)))
% => #f
#(display (reddish? magenta))
% => #t
```

First we check if the tested object is a color in the first place, and of course we don't reimplement that check but use the existing `color?` predicate. As the second subexpression of the `and` we build the sum of the second and third list elements and compare that to the first list element.

3.5.9.4.1.2. Choice

A common situation is that values with one out of several types can be accepted in a certain place. For these cases there already are a number of `X-or-Y?` predicates available, and one can easily write custom predicates as well. Imagine for some obscure reason you expect a variable to be either a list, a color or a symbol, then you can create the following predicate:

```
#(define (list-or-color-or-symbol? x)
   (or (list? x)
       (color? x)
       (symbol? x)))
```

While the previous - "narrowing" - predicates used the `and` conditional this type of predicates tends to use `or` instead.

Another typical "choice" type of predicate would check if a value is part of a predefined list:

```
#(define (mode? x)
   (and (symbol? x)
        (or (eq? x 'major)
            (eq? x 'minor))))
```

This would return `#t` when (and only when) applied to `'major` or `'minor`.

3.5.9.4.1.3. Caveat: "True Values"

As a last example I'm going to show you a somewhat more involved example with a caveat: checking if an object is an association list that contains a specific key:

```
#(define (alist-with-color? x)
   (and (list? x)
        (every pair? x)
        (assq 'color x)))

#(display
  (alist-with-color?
   '((amount . 5)
     (color . red))))
```

Surprisingly, when we compile this code the output on the console isn't `#t` or `#f` but `(color . red)`. This is because the predicate procedure has the value its last expression has, and this is the `assq` in this case. From the discussion of association lists we recall that the return value is either `#f` or the retrieved *pair* - but what we need is a simple `#t` in this case.

This means whenever a test used in a predicate returns "a true value" it has to be wrapped in order to really return `#t` or `#f`. Which is fortunately very easy to do:

```
#(define (alist-with-color? x)
   (and (list? x)
```

```
        (every pair? x)
        (if (assq 'color x)
            #t
            #f)))
```

The `assq` is wrapped in an `if` expression, so if `assq` returns "a true value" the expression manually returns `#t` instead.

A good exercise to do on your own now would be to write a predicate `alist-with-key?` where you can additionally specify the key whose presence you'd like to check.

3.5.9.5. Handle Different Parameter Data types

At several places we have seen that Scheme doesn't enforce types for procedure arguments but that (naturally) certain procedure applications may only work with certain types, e.g. numbers or strings. So while the following procedure `double` can be invoked with parameters of arbitrary type everything except numbers will cause errors:

```
#(define (double x)
   (+ x x))
```

But with everything we know by now about types, predicates and local binding we can rewrite this procedure in a robust manner. Concretely, what we want to achieve is: if `x` is a number we create the double, if it is a string that represents a number then first convert it and then double it, but return it as a string again. Finally, if it is a string that can't be converted to a number then it should be "doubled" in the way some other languages understand it: concatenating the string to itself. Finally we need a fallback action if it is neither a string nor a number.

The requirement of two specific cases plus a fallback suggests to use the `cond` conditional:

```
(cond
  (number: just double it)
  (string:
    (if it can be converted to a number:
        convert, double, return back
    )
    (else concatenate)
  )
  (else: report an error)
)
```

Leaving out the more complex handling of the string this looks like this in LilyPond:

```
#(define (double x)
   (cond
    ((number? x)
     (+ x x))
    ((string? x)
     "not implemented yet")
    (else
     "'double' needs a number or a string as parameter")))
```

To implement the string handling we have to know the procedure `string->number` that will return a number - or `#f` if the conversion fails. This is exactly what we need. We could now do something like:

```
(if (string->number x)
    (string->number y)
    ... else clause)
```

but that looks inefficient because the conversion is actually processed twice. Now (as we have seen) this is exactly the use case for local binding: we bind the result of that procedure to a local variable, and if that has a true value we use it, otherwise do the alternative concatenation:

```
#(define (double x)
   (cond
    ((number? x)
     (+ x x))
    ((string? x)
     (let ((num (string->number x)))
       (if num
           (number->string (+ num num))
           (string-append x x))))
    (else
     "'double' needs a number or a string as parameter")))

#(display (double 3))
#(newline)
#(display (double "3"))
#(newline)
#(display (double "A"))
#(newline)
#(display (double '(2 . 3)))
```

If the string can be converted to a number this value will be doubled and converted back to a string, otherwise the original input string is taken and concatenated to itself. Now the invocations correctly return `6`, `6` (as a string), `AA` and the error string.

A perfect exercise would now be to extend that procedure to handle the last invocation as well: if the parameter is a pair holding two numbers we return a pair with the numbers doubled each.

This is the general approach to handle variable parameter types in Scheme. If you don't know what type the input parameter will have and your procedure body has to care about types then you can block or choose within the procedure body using built-in or custom predicates.

3.5.10. Iteration and Looping Constructs

Iteration and loops are fundamental programming tasks in any language. However, in Scheme iteration over lists is somewhat special, as discussed on list iteration, and the general looping constructs `do` and `while` are considered somewhat unidiomatic.

As list iteration and particularly recursion are very much limited without making use of custom procedures their discussion has been deferred to this chapter, after writing procedures had been introduced.

3.6. Mapping List Elements

3.6.1. Single Lists

`map` "maps" the elements of a list to a new list, creating a new list from the results of the application of a procedure to each element.

```
(map proc list)
```

is the basic form of `map`. `proc` is a procedure accepting a single argument. The value of `proc` after evaluation will be added to the resulting list. As with `filter` there are not many applications for the built-in procedures, so more examples with custom procedures will be given below.

```
guile>(map abs '(-2 4 -0.25 1))
(2 4 0.25 1)
```

`map` takes two arguments here, a *procedure* and a *list*. Note that the procedure `abs` (determining the "absolute" or positive value of a number) is passed *without* the parens, as the bare procedure. Now `map` iterates over all elements of the list, applies `abs` to them and creates a list composed of the evaluations of `abs`.

Somewhat more involved and interesting is using `car` and friends to dissect nested and association lists. If you type `all-grob-descriptions` in your Scheme sandbox you will be faced with the display of a very big nested list. Each list element is itself a list with the first element being a symbol representing the grob name and multiple pairs documenting the properties and default values. This is just the *first* list element describing `Accidental` (already formatted for better reading):

```
(Accidental
  (after-line-breaking . #<primitive-procedure ly:accidental-interface::remove-tied>)
  (alteration . #<procedure accidental-interface::calc-alteration (grob)>)
   (avoid-slur . inside) (extra-spacing-width -0.2 . 0.0)
   (glyph-name . #<procedure accidental-interface::glyph-name (grob)>)
   (glyph-name-alist
     (0 . accidentals.natural)
     (-1/2 . accidentals.flat)
     (1/2 . accidentals.sharp)
     (1 . accidentals.doublesharp)
     (-1 . accidentals.flatflat)
     (3/4 . accidentals.sharp.slashslash.stemstemstem)
     (1/4 . accidentals.sharp.slashslash.stem)
     (-1/4 . accidentals.mirroredflat)
     (-3/4 . accidentals.mirroredflat.flat))
  (stencil . #<primitive-procedure ly:accidental-interface::print>)
  (horizontal-skylines . #<unpure-pure-container #<primitive-procedure ly:accidental-
interface::horizontal-skylines> >)
  (vertical-skylines . #<unpure-pure-container #<primitive-procedure
ly:grob::vertical-skylines-from-stencil> #<primitive-procedure ly:grob::pure-simple-
vertical-skylines-from-extents> >)
  (X-offset . #<primitive-procedure ly:grob::x-parent-positioning>)
  (Y-extent . #<unpure-pure-container #<primitive-procedure ly:accidental-
interface::height> >)
  (meta (name . Accidental)
  (class . Item)
  (interfaces grob-interface accidental-interface font-interface inline-accidental-
interface item-interface)))
```

In order to extract a plain listing of all grob *names* we have to retrieve the first element from each sub-list, which can conveniently be achieved using `car`.

Entering

```
(map car all-grob-descriptions)
```

in the sandbox will show you a long list with (only) all grob names, which is very manageable but not that printable here on this page …

3.6.2. Multiple Lists

Interestingly (and different from most other languages) `map` supports mapping of *multiple* lists. In fact `map` does *not* accept exactly one list argument, instead the given procedure must accept as many

arguments as there are additional list arguments. Then it passes the corresponding elements of all lists to the procedure.

```
guile> (map cons '(1 2 3 4) '(2 3 4 5))
((1 . 2) (2 . 3) (3 . 4) (4 . 5))
```

Each corresponding element of both lists is passed to `cons` as one of its arguments, so `cons` can combine the corresponding elements of both lists to a pair.

You should take care of having lists of equal length because extra elements of the longer list are discarded without any further warning.

```
guile> (map cons '(1 2 3) '(2 3 4 5))
((1 . 2) (2 . 3) (3 . 4))
```

`map` supports procedures with arbitrary numbers of arguments, as long as they match the number of passed lists:

```
guile> (map list '(1 2 3) '(4 5 6) '(7 8 9))
((1 4 7) (2 5 8) (3 6 9))
```

3.6.3. Using `map` to Dissect Nested Lists

Let's complete this section with a slightly complex example making use of a `lambda` expression to transform a list to a different structure.

Let's define a "context mod" variable:

```
contextMod = \with {
  instrumentName = "Violin"
  shortInstrumentName = "Vl."
}

mods = #(ly:get-context-mods contextMod)
```

A `\with {}` clause is generally used for modifying contexts, for passing additional parameters to the creation of voices or staff contexts etc. But it can also be (ab?)used as a function argument to specify a list of key-value pairs. In order to make it usable we have to extract the actual content through `ly:get-context-mods`, which is assigned to the `mods` variable.

```
#(write mods)
% => ((assign instrumentName "Violin") (assign shortInstrumentName "Vl."))
```

`mods` is now a list with two elements, each of which is a three-element list of the symbol `assign`, the key name and the value. In order to make use of this as a configuration store we want to convert it into an association list, i.e. a list consisting of key-value pairs. We want each pair to have the second element as its key and the third element as its value part, so we could - manually - construct it like this:

```
options =
#(list
  (cons 'instrumentName "Violin")
  (cons 'shortInstrumentName "Vl."))

#(write options)
% => ((instrumentName . Violin) (shortInstrumentName . Vl.))
```

But of course this only works because we know the actual input list, in any real-world cases we have to resort to a more generic approach: *mapping*.

We need to create a list of pairs where each pair matches an element of our input list. So we can `map` the list over a procedure that converts each sub-list of the original input list to the desired pair. As such a procedure does not exist we will have to write it on our own.

The following `lambda` expression creates a procedure that takes exactly one argument and evaluates to a pair created from the argument's second and third element (obviously expecting a list of at least three elements):

```
(lambda (mod)
  (cons (second mod) (third mod)))
```

So this is an expression that *evaluates* to a procedure. That means when using it with `map` we *do* have to *invoke* the lambda expression, i.e. surround it with parens. Let's write this in LilyPond again:

```
options =
#(map
  (lambda (mod)
    (cons (second mod) (third mod)))
  mods)
  #(write options)
  % => ((instrumentName . "Violin") (shortInstrumentName . "Vl."))
```

As this process is something very useful we can now write a function that directly converts a `\with {}` expression into such a property alist. And in fact I have done exactly this in openLilyLib, more concretely in oll-core:

```
#(define (context-mod->props mod)
   (map
    (lambda (prop)
      (cons (cadr prop) (caddr prop)))
    (ly:get-context-mods mod)))
```

The naming of elements indicates their use here: the input is about *mods* (the context-mod) while internally the sublists are considered *properties*.

3.7. Iterating Over Lists: `for-each`

`for-each` is very much like `map` with one single difference: the results of the procedure are not used for anything. Concretely, `for-each` iterates over the elements of a list (or multiple lists) and applies a procedure to each, without creating a new list from the results.

```
guile> (for-each display '(1 2 3 4 5 6 7 8 9))
123456789
```

In this short example the procedure `display` is applied to all the numbers in the list. `display` simply prints the value to the console but doesn't evaluate to anything. So one can also say `for-each` applies the procedure only for its *side-effects*, not for its *value*.

It is also possible to apply procedures that *do* evaluate to something, but that value will simply not be used, so the following example is actually useless:

```
guile> (for-each symbol->string '(a b c d e))
```

This will convert all the symbols in the list to strings but not *do anything* with them.

Everything that is said about processing multiple lists with `map` applies to `for-each` as well.

But actually `for-each` is most useful in combination with custom procedures, even more so than `map`.

4. Scheme in LilyPond

- Music, Scheme and Void Functions
  - Interface of the functions
  - Switching Between Scheme and LilyPond
- Markup Functions

5. Advanced Interaction With Scheme

- Built-in Scheme Functions
- Overriding Stencils
- Scheme Engravers
- Scheme Representation of Music

5.1. Built-in Scheme Functions

LilyPond's documentation has a page "Scheme Function", which is here for the current stable version 2.18 and here for the current development version 2.19.

This is a reference of an enormous number of extremely useful functions, namely all those Scheme function starting with `ly:` that provide an interface with the inner state and workings of LilyPond and a score document. Unfortunately this page is extremely difficult to digest - to a point to being barely usable as more than a *reminder* of what's available. This is mostly due to the fact that this whole page is generated from "docstrings", concise explanations stored directly in the source code. So if you are feeling dumb when reading the docs be assured that you're not alone. Usually it is only possible to get some value out of it when someone on the `lilypond-user` mailing list gives you some snippets to digest or to simply insert into your current project.

The purpose of this section of the book is to provide usable explanations of the different Scheme functions, giving you the knowledge that is necessary to successfully make use of that "communication channel" into LilyPond's heart.

{% credits %}{% endcredits %}

6. Old Stuff

6.1. Music Functions 1: Getting to Grips with Scheme in LilyPond

With this post I intend to start a series of posts introducing writing Scheme music functions in LilyPond input files. Please note that this is not a guru's wisdom poured into blog posts but rather a documentation of my own thorny learning process.

6.1.1. The First Basic Scheme Function

Now we're going to write our first *music function* as described in the Extending manual. Actually it will be a quite useless function but at least it will work - and hopefully get you an understanding how to do it. But let's start with doing it the normal LilyPond way:

```
\version "2.18.0"

myFunction = { c2 }

\relative c' {
  c4 \myFunction c
}
```

first-music-function.png

Here we define a variable with the (unusual) name of `myFunction`, which we can later insert in the main music expression. As you can see it will insert a half note `c` into the music. But as you can also see it does so without affecting the parser - the next note will be a quarter note again, referencing the previous duration *encountered in the source code*.

Now we'll do the same with a Scheme music function:

```
mySchemeFunction =
#(define-music-function (parser location)()
   #{
     c2
   #})

\relative c' {
  c4 \mySchemeFunction c
}
```

First we define a LilyPond variable `mySchemeFunction` and assign it - a Scheme expression (as you can see from the hash with consecutive parentheses). This expression consists of four parts:

- The keyword `define-music-function` This isn't a Scheme keyword but a function defined by LilyPond. As the name suggests it defines a "music function", which is a function that *returns some music*. And as such it can be *used* like a music expression as you see it in the last line of the example. The following three items are the arguments `define-music-function` expects.
- The list of arguments the two arguments `parser` and `location` are mandatory, and as long as you won't need to make use of the information in them you can just forget about them and type it out for each music function you define. If you want your function to process individual arguments (which we'll see in a later post) you will add telling names to this list, for example `(parser location slope)` for one additional argument.
- The *types* of your individual arguments. In the example given for `slope` you would add a type (or "predicate") here, e.g. `(number?)`, but in the example we used you have to provide an empty pair of parens.
- The actual music expression to be returned As we've said a music function returns a music expression. Scheme functions generally return what the evaluation of the last expression produces, so this is where we have to do it. Instead of having to write some Scheme code producing a LilyPond music expression (for which we don't have any means so far) we fortunately can use the `#{ ... #}` construct which allows us to switch to LilyPond mode within Scheme code. Concretely this means the section enclosed in the hash+curly braces is treated as one Scheme expression but can be written with LilyPond code. In our case this returns a music expression with the content of `c2`. So our whole music function returns `{ c2 }`. And so our second example produces exactly the same result as the first one: `{ c4 c2 c4 }`.

Of course it doesn't make much sense to write a function that returns a hard-coded value. But I think it was a good example to understand the first steps of integrating LilyPond and Scheme code. And I promise that in the next post we'll do something useful.

{% credits %}{% endcredits %}

6.2. Music Functions 2: Start Doing something Useful

In the previous post I demonstrated how to include Scheme code in LilyPond input files, and we saw our first *music function*, although it still was a static function only producing a hard-coded middle c. Today we'll do some more by writing functions that actually do something useful and can handle *arguments*.

6.2.1. Recall: Another static function

[Please note: I use "static function" in the colloquial sense and not in the sense of object oriented programming.]

```
makeRedNote =
#(define-music-function (parser location)()
   #{
     \once \override NoteHead.color = #red
     \once \override Stem.color = #red
   #})

\relative c' {
  c4 \makeRedNote c c c
}
```

`second-music-function.png`

As in the example of the previous post this music function returns a hard-coded music expression, only that it doesn't return actual music but rather two overrides that will color the next note red. If you'd want to cover arbitrary music you'd have to override other grobs too, e.g. Flag, Beam, Dots etc.

6.2.2. Making the Function More Flexible: Arguments

Of course it would be dull to copy the function for all colors we might want to apply. A better approach is to *parametrize* the function by accepting an *argument* for the color. In the last post we already saw how arguments can be added to the function definition:

```
colorNote =
#(define-music-function (parser location my-color)
   (color?)
   #{
     \once \override NoteHead.color = #my-color
     \once \override Stem.color = #my-color
   #})
```

I added `my-color` to the list of arguments and provided the `color?` predicate as the single element in the list of argument types. (By convention one writes this type list on its own line. Only when it's empty it's often written at the end of the first line because the empty parens on a single line looks somewhat strange.)

This now means that the function expects to be called with an argument of type `color?`. Inside Scheme code this argument can simply be referenced as `my-color`, but in our LilyPond section enclosed by `#{ ... #}` we have once more to change mode back to Scheme by using the `#` in `#my-color` to reference the Scheme variable.

So our function still does produce the two property overrides, but it doesn't use the hardcoded `red` anymore but will apply the color provided when *calling* the function.

```
\relative c' {
  c4 \colorNote #blue c \colorNote #'(0.5 0.5 0) c c
}
```

`third-music-function.png`

Please have a look at how we call the function now. The function expects an argument of the Scheme type `color?` which is defined by LilyPond. As you will recall we tell LilyPond to switch to Scheme mode with the `#` character, so we can pass the *Scheme* value `blue` to the function. Alternatively we can create an arbitrary color on-the-fly by passing a list of three values

(representing **R** ed, **G** reen, **B** lue) and pass this as a Scheme value. Finding out how exactly to write Scheme values in LilyPond can be confusing for beginners. Sometimes you write them literally (as with the colors), sometimes you have to prepend a `'` or enclose it in `'()`. Getting used to this issue is one major part of getting used to Scheme in LilyPond, and I can't help you too much with that. Basically everything is explained or at least documented in the Scheme Tutorial of LilyPond's documentation - maybe after having read up to this point you may want to (re-)read that now.

6.2.3. Making It Even More Flexible: Processing a Music Argument

As a next step we can actually pass a *music expression* to the function. This is done with the type predicate `ly:music?` (functions and predicates with the `ly:` prefix are not pure Scheme but defined by LilyPond).

```
noOp =
#(define-music-function (parser location my-music)
   (ly:music?)
   #{
     #my-music
   #})
```

This music function takes a music expression as its argument and returns exactly this music expression, so actually it doesn't do anything. But there's one interesting thing I can demonstrate with this. The `ly:music?` argument is a Scheme music expression, and what we want to return is exactly that. Therefore we don't need to switch to LilyPond mode in the function body and then reference the Scheme variable from there. Instead we can simply write `music` as the expression inside the function body:

```
noOpTwo =
#(define-music-function (parser location my-music)
   (ly:music?)
   my-music)
```

This will return the "result of the evaluation of `music`" - which is exactly the same as in the previous example, and you can see that calling `\noOp` or `\noOpTwo` doesn't make a difference:

```
\relative c' {
  c4 \noOp c \noOpTwo c c
}
```

`fourth-music-function.png`

Please note: As the second argument - the `ly:music` we provided the simple `c` here, as a single note is already a music expression for LilyPond. But often you will want to pass compound music expressions as we'll see in the next example.

6.2.4. Coloring Arbitrary Music with Arbitrary Colors

As a conclusion for today's post we'll write a function that colors an arbitrary music expression with an arbitrary color. For this it will take two arguments, the color and the music. First we will override the color property of a number of grobs with the given color, then we produce the music given as the argument and finally we'll revert the color property of the grobs. Please note the use of `\temporary`, a rather new feature that isn't available in stable versions before LilyPond 2.18. This will ensure that reverting the property will *not* revert to the default value (i.e. the color black) but rather to the value that was effective immediately before. If you have to use LilyPond 2.16 you can simply leave that out.

```
colorMusic =
#(define-music-function (parser location my-color my-music)
```

```
  (color? ly:music?)
  #{
    \temporary \override NoteHead.color = $my-color
    \temporary \override Stem.color = $my-color
    \temporary \override Flag.color = $my-color
    \temporary \override Beam.color = $my-color
    \temporary \override Rest.color = $my-color
    \temporary \override Slur.color = $my-color
    \temporary \override PhrasingSlur.color = $my-color
    \temporary \override Tie.color = $my-color
    \temporary \override Script.color = $my-color
    \temporary \override Dots.color = $my-color

    $my-music

    \revert NoteHead.color
    \revert Stem.color
    \revert Flag.color
    \revert Beam.color
    \revert Rest.color
    \revert Slur.color
    \revert PhrasingSlur.color
    \revert Tie.color
    \revert Script.color
    \revert Dots.color
  #})
```

To test the function I write some more complex music in a variable and call the function in different ways.

```
myMusic = \relative c' {
  c4. d8 e16 d r cis( d4) ~ | d1 \fermata
}

\relative c' {
  \colorMusic #blue \myMusic
  \colorMusic #red { c4 c } d \colorMusic #green e\f
  \colorMusic #magenta \myMusic
```

`fifth-music-function.png`

You can see that the function works regardless of getting the music as a single note, a compound expression with curly braces or by passing it the `\music` variable. But you'll also notice that the accidentals and the dynamic letters aren't colored. But of course there's an easy fix: just add the respective overrides to the function.

Everything's fine now. But looking at the last function definition it seems there's a *lot* of redundancy here, and redundancy is something we usually want to get rid of. Probably it would be a good idea to factor all this out into a separate function - and that's what we'll do in the next post.

{% credits %}{% endcredits %}

6.3. Music Functions 3: Reusing Code

In the last post We wrote a music function that took an arbitrary music expression and applied an arbitrary color to it. The problem we immediately saw is that we manually had to override all properties - and that we still didn't manage to catch them all. In general terms, a function that redundantly calls `\override` for each new property isn't exactly elegantly programmed. So today we

will do better and at the same time learn something about reusing code by *refactoring* and by processing *lists*.

This was the function as we wrote it originally:

```
\version "2.18.0"

colorMusic =
#(define-music-function (parser location my-color music)
   (color? ly:music?)
   #{
     \temporary \override NoteHead.color = #my-color
     \temporary \override Stem.color = #my-color
     \temporary \override Flag.color = #my-color
     \temporary \override Beam.color = #my-color
     \temporary \override Rest.color = #my-color
     \temporary \override Slur.color = #my-color
     \temporary \override PhrasingSlur.color = #my-color
     \temporary \override Tie.color = #my-color
     \temporary \override Script.color = #my-color
     \temporary \override Dots.color = #my-color

     #music

     \revert NoteHead.color
     \revert Stem.color
     \revert Flag.color
     \revert Beam.color
     \revert Rest.color
     \revert Slur.color
     \revert PhrasingSlur.color
     \revert Tie.color
     \revert Script.color
     \revert Dots.color
   #})
```

6.3.1. Start factoring out common functionality

For each notation element in question we called `\temporary override NNNN.color = #my-color`, so the first thing we'll do is *factoring out* this common functionality in a dedicated function. This will be a helper function usually not called directly. Each use of `\override` could be enclosed in a music expression so we'll write a new *music function* that returns such a music expression.

```
colorGrob =
#(define-music-function (parser location my-grob my-color)
   (symbol? color?)
   #{
     \temporary \override #my-grob #'color = #my-color
   #})
```

As you can see this function takes two arguments: a *grob* name (GRaphicalOBject) and a color. The color has the type we already know for it (*color?*) while the grobname has to be given as a *symbol?*. In the function body the `\temporary \override` is applied to the given grob with the given color. So now we can call the function with `\colorGrob NoteHead #red`, and as we already have the color as the argument to our entry function `\colorMusic` we can use `\colorGrob NoteHead #my-color` in our main function.

For uncoloring the music we write a corresponding function `\uncolorMusic` that uses `\revert` instead of `\override`. Our modified main function and its two new helper functions now look like this:

```
\version "2.18.0"

colorGrob =
#(define-music-function (parser location my-grob my-color)
   (symbol? color?)
   #{
     \temporary \override #my-grob #'color = #my-color
   #})

uncolorGrob =
#(define-music-function (parser location my-grob)
   (symbol?)
   #{
     \revert #my-grob #'color
   #})

colorMusic =
#(define-music-function (parser location my-color music)
   (color? ly:music?)
   #{
     \colorGrob NoteHead #my-color
     \colorGrob Stem #my-color
     \colorGrob Flag #my-color
     \colorGrob Beam #my-color
     \colorGrob Rest #my-color
     \colorGrob Slur #my-color
     \colorGrob PhrasingSlur #my-color
     \colorGrob Tie #my-color
     \colorGrob Script #my-color
     \colorGrob Dots #my-color

     #music

     \uncolorGrob NoteHead
     \uncolorGrob Stem
     \uncolorGrob Flag
     \uncolorGrob Beam
     \uncolorGrob Rest
     \uncolorGrob Slur
     \uncolorGrob PhrasingSlur
     \uncolorGrob Tie
     \uncolorGrob Script
     \uncolorGrob Dots
   #})

music = \relative c' {
  c4. d8 e16 d r cis( d4) ~ | d1 \fermata
}

\relative c' {
  \colorMusic #blue \music
  \colorMusic #red { c4 c } d \colorMusic #green e\f
```

```
  \colorMusic #magenta \music
}
```

which gives the same result as in the last post: `fifth-music-function.png`

OK, this doesn't make our entry function that much shorter, but we have already achieved a significant first step: avoiding the repetition of the same code. This makes it possible to maintain this code in *one* place. For example, I told you that `\temporary` is a rather new command and that you may just skip that command if you want to use LilyPond 2.16 for some reason. Now that we have factored out the command into a function you can simply update this function *once* and have the modified code for all instances of the function. Or imagine that you want to temporarily disable the function completely, then you can simply comment out the line in the function, and everything will remain black. Nevertheless this has only been the first step …

6.3.2. Using a List as Argument

We haven't yet arrived at the goal because we still need to call our helper functions for each grob type individually. The goal would be to use only a single function call, and that can be achieved by passing the grob names as a *list*. So we want to write a function with the following *signature*:

```
colorGrobs =
#(define-music-function (parser location my-grob-list my-color)
   (symbol-list? color?)
   #{

   #})
```

This function takes a list of grob names and the color and will then *iterate* over this list to apply the overrides (using the helper functions we have already written). We'd call that function like this from our main function:

```
\colorGrobs #'(NoteHead
               Stem
               Flag
               Beam
               Rest
               Slur
               PhrasingSlur
               Tie
               Script
               Dots
               DynamicText
               Accidental) #my-color
```

This will make our main function much more concise and maintainable because adding a newly noticed grob simply requires adding its name to the `my-grob-list` list.

6.3.3. Iterating Over the List

While writing the *signature* of the `\colorGrobs` function was really easy iterating over that list of grob names is something that really can drive you crazy if you're not yet really familiar with Scheme. And I have to admit it drove *me* crazy while writing this post - which you can take as an indication that I won't be able to continue this series too much further ;-)

A straightforward way to iterate over a list that is comprehensible for people coming from other programming languages is the `map` operation. It takes a function and a list of values and applies the function to each of the values one by one. It returns a new list of modified values.

```
(map colorGrob my-grob-list)
```

will walk over our list of grob names and pass each one to the `colorGrob` function. While this looks quite concise it doesn't take the color argument into account, so we have to look further.

Seemingly the solution to this issue would be the `lambda` construct which creates an ad-hoc procedure (basically an unnamed function) that can use any number of arguments.

```
((lambda (arg) (colorGrob arg my-color)) NoteHead)
```

will create a function with the body:

```
(colorGrob arg my-color)
```

`arg` is defined to be the argument passed into the function, and `NoteHead` is passed into it, so `colorGrob` will actually be called:

```
(colorGrob NoteHead my-color)
```

This example is quite useless of course and only there to show how `lambda` can work with hard-coded and variable arguments, but can be made fruitful together with `map`.

```
(map (lambda (arg) (colorGrob arg my-color)) my-grob-list)
```

will walk over `my-grob-list` and pass the items one by one to the ad-hoc (lambda) function. This way we iterate over the list of grob names passing them each to `colorGrob`, accompanied by the static color argument (my-color).

Unfortunately, our journey isn't at its end. As said `map` applies the elements of a list to a given function. This function evaluates to a single value (which is what each Scheme expression or function does), and `map` creates a new list of all these values. This is very useful for manipulating lists in general, but in our case this is a problem: we have to return a *music expression*, but `map` creates a *list of music expressions*. So unfortunately we can't go that way and have to apply real *recursion*. (The Guile manual documents map and lambda, and is a helpful resource for Scheme in general.)

6.3.4. Preliminary Conclusion

Today we seemingly didn't achieve too much: we factored out some code and then defined the goal where we want to go next. But on the way to that goal we arrived at a dead end. But of course I described this on purpose because I hope discussing these things at such a slow pace will give you an opportunity to get familiar with some of the peculiarities of how Scheme works. While `map` and `lambda` don't help us with the current issue they are indispensable tools for your career as a LilyPond-Scheme programmer, and it's good to get to grips with them as early as possible.

We will complete this exercise in the next post and learn about a very fundamental concept of Scheme: *recursion*.

{% credits %}{% endcredits %}

6.4. Music Functions 4: Recursion

Today we'll come to an end with a series of posts demonstrating how to write a comparably simple music function in LilyPond. I deliberately chose to describe the stuff with such verbosity and at that gentle pace because I think that's what is missing in LilyPond's (otherwise excellent) documentation. At least I think *I* would have greatly benefitted from that kind of explanation …

In the first part we started considering how to insert a Scheme music function in a LilyPond input file, and in the second part we started writing a function that colors an arbitrary function with an arbitrary color. While this was more or less hard-coded and exposed lots of redundant code we

wanted to improve it in the third part through *refactoring* and *list processing*. Unfortunately this didn't work out as easily as expected, and finally we'll be solving the task through *recursion* today.

In a comment to the previous post Jay Anderson pointed out that there is another solution that doesn't require using recursion. However I decided not to change the post that I had already written. So just keep in mind that the following post does *not* demonstrate the only approach but rather one of several different options. Apart from that I think that while Jay's solution is actually more straightforward as a programming construct it would require considerably more previous knowledge and understanding. Using local variables with Scheme is completely different from other languages' behaviour, so it would warrant a dedicated post or even series of posts.

6.4.1. Recursion

Scheme is a dialect of the LISP family of programming languages. "LISP" is derived from "LISt Processing", and therefore lists are an ubiquituos concept in Scheme as in all other LISP languages. By now I can see some of the benefits of that approach, but I do insist on saying that it can be quite difficult to get into the spirit of things with Scheme. And until you have achieved this nearly everything about it can be extremely confusing. List handling is one of these things.

To solve our task of calling the coloring function for each grob type you would write something like this in Python:

```
for grob in my_grob_list:
    color_grob(grob, my_color)
```

which can be directly translated to English as "For each `grob` in the list `my_grob_list` please call the function `color_grob`, passing it the current `grob` and the general color `my_color` as arguments". In Scheme you have to take a completely different approach by calling a function recursively.

Recursive functions are functions that call themselves with a modified argument list. A very important thing is to insert a check for a state to end the recursion because otherwise the program would get stuck in an infinite loop. The general outline for such a function is:

- check for the exit state and leave the function if this is true
- do something with the arguments and recursively call the function itself with modified arguments.

When using recursion to iterate over lists this usually means you will do something with the first element of the list and call the function with the remaining part of the list. The exit check can then determine if there are still list elements present to be processed.

Since our current case of a LilyPond *music function* complicates matters we'll have a look at a generic Scheme function first:

```
#(define (display-names my-names-list)
   (if
    ;; check if there still is a list element left
    (null? my-names-list)
      ;; no: we're at the end of the recursion
      ;; simply issue a new line and exit
      (newline)
      ;; yes: we still have elements to process
      ;; begin a compound expression
      (begin
       ;; display the first element and insert a new line
       (display (car my-names-list))
       (newline)
       ;; recursively call the function
```

```
      ;; with the rest of the list
      (display-names (cdr my-names-list))) ;; end of begin
     ) ;; end of the if expression
  )
```

This defines a Scheme function with the name `display-names` and an argument `my-names-list`). Other than LilyPond's music functions this doesn't check the type of the argument, but the function body expexts it to be a list. Actually the code comments say it all, and it's the implementation of what has been said above. The only thing to notice is that in LilyPond's Scheme you have to give a "then" *and* an "else" expression to the `if` clause. In other Scheme dialects this may be omitted. And you should be very familiar with the `car` and `cdr` operators and their relatives. `car` returns the first element of a pair or list. `cdr` returns the second element of a pair, or with a list it returns a new list with all the elements of the list except for the first one (the "rest" of the list). (Or with Scheme lists you can consider a list to be a pair with one first element (the `car`) and the second element (the `cdr`) being a list on its own.)

Now you can call that function with

```
#(display-names '(NoteHead Slur Flag Beam))
```

and see your list of grob types printed nicely in the console output.

6.4.2. Applying It To Our Task

You will recall that we wanted to write a function `\colorGrobs` that takes a list of grob names and a color as its arguments. In the last post we already wrote the function signature and now we can apply our recursion experience to write the body of that function. Finally it looks quite concise too:

```
colorGrobs =
#(define-music-function (parser location my-grob-list my-color)
   (symbol-list? color?)
   (if (null? my-grob-list)
       ;; issue an empty music expression
       #{ #}
       #{
         % color the first grob type of the current list
         \colorGrob #(car my-grob-list) #my-color
         % recursively call itself with the remainder
         % of the current list.
         \colorGrobs #(cdr my-grob-list) #my-color
       #}))
```

We define a music function that takes a list of grob names and a color as its arguments. When the list is empty the recursion stops by issuing an empty music expression. This is very handy as this way you can write an empty part for the if clause, something you can't do in pure Scheme code in LilyPond. If there *is* an element left we call `\colorGrob` passing it the first list element as the grob name argument and then recursively call `\colorGrobs`itself with the grob name list stripped of its first element. That's what we do with the `car` and `cdr` operators.

Now we have to write a corresponding `\uncolorGrobs` function and put everything togehter that we have done so far. I won't spend all the screen estate to quote it all once more, but you can download the complete LilyPond file to run and inspect.

6.4.3. More Cleaning Up

OK, we have achieved our goal, a function that colors an arbitrary music expression in an arbitrary color. But I won't let you go home now because there are still some things that can be improved

with regard to “code hygiene”. The first issue will be some more factoring out to get rid of redundant code, the second will make the function still more generic.

6.4.4. Factoring Out Even More

One thing I don’t like about our solution so far is that we have to write several things twice, for *coloring* and *uncoloring*. It would make sense to merge the respective functions into *one* by providing yet another argument telling which direction we are going to do.

First we update our `\colorGrob` function to take an additional boolean `color-on` argument. If that is set to `#t` (true) we apply the `\override`, otherwise we `\revert` it:

```
colorGrob =
#(define-music-function (parser location my-grob my-color color-on)
   (symbol? color? boolean?)
   (if color-on
       #{
         \temporary \override #my-grob #'color = #my-color
       #}
       #{
         \revert #my-grob #'color
       #}))
```

The necessary change to our outer function `\colorGrobs` is even smaller. We don’t need to add a new conditional clause here but can simply extend the argument list by the boolean argument `color-on` and pass this on to `\colorGrob` unchanged.

```
colorGrobs =
#(define-music-function (parser location my-grob-list my-color color-on)
   (symbol-list? color? boolean?)
   (if (null? my-grob-list)
       ;; issue an empty music expression
       #{ #}
       #{
         % color the first grob type of the current list
         \colorGrob #(car my-grob-list) #my-color #color-on
         % recursively call itself with the remainder
         % of the current list.
         \colorGrobs #(cdr my-grob-list) #my-color #color-on
       #}))
```

Now we have to update our entry function `\colorMusic` to call the new internal function accordingly. Here we get a little inconsistency: Instead of calling `\uncolorGrobs` we now call `\colorGrobs` and have to pass a color as an argument although uncoloring doesn’t need a color information at all. I will accept this for today because it doesn’t add too much redundancy and because it is in code that the *user* won’t ever have to enter. The clean solution would be to work with optional arguments but we’ll defer this topic to a future post. As before I won’t copy all the stuff but provide the complete file for download.

6.4.5. Use a Generic Grob List

One last thing we will improve in our function is the list of grob names that has to be passed into the function. This requires some typing and especially it makes it somewhat random whether all relevant grobs have been taken care of. Fortunately LilyPond provides a means that can give us all we need to make this really generic: the `all-grob-descriptions` function. This produces a nested list of pairs, where each pair consists of a grob name and a list of its properties. Now we can make use of several things we learnt during this tutorial series: `map`, `lambda`, `car` and `cdr`.

What we need is a list of all grob names, that is the first element of all pairs that make up the `all-grob-descriptions` list. In other words we need to `map` the `car`s of all elements of this list to a new list. You should remember that `map` takes the elements of a list and passes it to a function and creates a new list of the results of that function, and we can use `lambda` to create this inner function ad-hoc.

```
(map (lambda (gd) (car gd)) all-grob-descriptions)
```

is all we need to perform this. The inner function will produce the `car` of all elements it is passed, `map` provides the elements of the `all-grob-descriptions` list and will produce the new list containing all grob names that LilyPond knows. You may check the result of this by `display`ing it. Just put

```
#(display (map (lambda (gd) (car gd)) all-grob-descriptions))
```

in an otherwise empty LilyPond file and compile it. This won't produce a PDF but instead lists all available grob names in the console output.

The last step to do is to enclose this line in a newly written Scheme function and call this function when we need the list of grob names:

```
allGrobNames =
#(define-scheme-function (parser location)()
   (map (lambda (gd) (car gd)) all-grob-descriptions))

colorMusic =
#(define-music-function (parser location my-color music)
   (color? ly:music?)
   #{
     \colorGrobs \allGrobNames #my-color ##t

     #music

     \colorGrobs \allGrobNames #my-color ##f
   #})
```

One thing to note with this is that using `\allGrobNames` significantly slows down the compilation of our test music. I didn't check the impact on large scores, but in the end you may have to evaluate the tradeoff between a generically comprehensive solution and processing speed in the context of your own concrete project.

As a final listing I will now print the whole file - which has become comparably short by now - and the resulting score - now the accidentals and dynamics are colored as well.

```
\version "2.18.0"

colorGrob =
#(define-music-function (parser location my-grob my-color color-on)
   (symbol? color? boolean?)
   ;; check for the boolean argument
   (if color-on
       ;; either set the color for the grob type
       #{
         \temporary \override #my-grob #'color = #my-color
       #}
       ;; or revert it
       #{
         \revert #my-grob #'color
```

```
       #}))

colorGrobs =
#(define-music-function (parser location my-grob-list my-color color-on)
   (symbol-list? color? boolean?)
   (if (null? my-grob-list)
       ;; issue an empty music expression
       #{ #}
       #{
         % color the first grob type of the current list
         \colorGrob #(car my-grob-list) #my-color #color-on
         % recursively call itself with the remainder
         % of the current list.
         \colorGrobs #(cdr my-grob-list) #my-color #color-on
       #}))

allGrobNames =
#(define-scheme-function (parser location)()
   ;; create a list with all grob names from LilyPond
   (map (lambda (gd) (car gd)) all-grob-descriptions))

colorMusic =
#(define-music-function (parser location my-color music)
   (color? ly:music?)
   #{
     \colorGrobs \allGrobNames #my-color ##t

     #music

     \colorGrobs \allGrobNames #my-color ##f
   #})

music = \relative c' {
  c4. d8 e16 d r cis( d4) ~ | d1 \fermata
}

\relative c' {
  \colorMusic #blue \music
  \colorMusic #red { c4 c } d \colorMusic #green e\f
  \colorMusic #magenta \music
}
```

colored-grobs2.png

As you can see our project has turned into a number of short functions (they could have been formatted even more concisely, and some of the comments could be left out in real life). It is characteristic when using Scheme to have many functions that are defined somewhat pyramidal: from the most broad functions at the bottom up to ever more specific and small functions at the top of the file. This is due to the nature of Scheme consisting of *expressions* that are *evaluated* rather than of *commands* that are *executed*. With Scheme you don't generally write a chain of commands or loops within one function but rather replace any compound statement with a function call.

Of course - and explicitly pointing this out may be important to a significant share of the readers - I would now put all code above the `music` variable definition in a library file and `\include` this. So *using* the `\colorMusic` function doesn't have any bigger impact on the structure and readability of the actual music input files.

We have come to the conclusion of our little series of tutorial posts. I hope you enjoyed them and have learned something along the way. Maybe writing Scheme functions isn't all that daunting to you anymore - that would be a great outcome of my efforts. At least for me this was the case. I think I've taken the next step in tackling Scheme - of course I'm now waiting for the next occasion where I feel stupid not managing to cope with seemingly simple tasks …

Feel free to improve the tutorials by asking questions or adding examples in the comments. And if you have something to say on the matter or on other Scheme related topics don't hesitate to contribute your own tutorial.

{% credits %}{% endcredits %}

7. License

TODO: The credits block should contain a link to the license page and the `git blame` page for the given page ("released according to CC, but the individual copyright can be retrieved from this page").