c-Logo-setup: ‘Contribution Manifesto’ (version 0.0.1)

Jochen ‘Lispl-Wicht’ Heller

2nd August 2024

Contents

Chapter 1

Introduction

First draft, about to be completely revised, but already contains informations about the direction:

March 2024

This document is supposed to be as well an introduction to programming in Common Lisp as a documentation of the project. The goal is to motivate people who are interested in maintenance and progression of the *cLogoSuite* and the underlying *common Lisp Logo setup (c-logo-s)* to participate in the project in several ways.

The introduction to Common Lisp follows a practical approach: Any chapter contains the description of one specific part of the whole suite. We start with building up the Logo programming language as an extension of Common Lisp.

Any knowledge of Common Lisp which is needed for understanding the respective chapter will be explained within the chapter. Therefore the language introduction follows the respective task. Where needed, we begin on a general level for a systematic overview. But soon, we will move forward to those concrete means needed for the concrete purpose of a section. Thus the readers learn Common Lisp *and setting up a useful development environment for this project* 'on the job'.

This is not meant as a new didactical approach. I hope, it just helps possible contributors to keep their motivation: Less working through the pure fundamentals more 'medias res': learning by real use-case examples. Most of the historical hints and ovations of the language in use typical for programming textbooks will be missing.

However, I make use of these very good textbooks which I most warmly entrust to the readers as backup for the understanding:

- Peter Seibel: *Practical Common Lisp*,
- Conrad Barski: *Land of Lisp*,
- Michał'phoe' Herda: *The Common Lisp Condition System*,
- Vsevolod Domkin: *Programming Algorithms in Lisp*.

Additionally: The following chapter is anyway reserved for setting up the system we will work with. Since one of the most useful *Integrated Development Environments*

(IDE) for programming in Common Lisp is *Emacs* (besides *Vim*) and I am used to it, the first chapter starts with installing the IDE, opening a Common Lisp file and starting the Common Lisp environment for the first steps in the language and doing some necessary extra steps.

The third chapter will introduce to some basic concepts of Common Lisp first and foremost for getting used to its syntax.

In the fourth chapter we work out the basics for the *common Lisp Logo setup*. For this it is also necessary to learn about the tools for managing a Common Lisp project. Additionally, for the content of this chapter it is essential to learn about Lisp macros already, parsing a Logo instruction and creating a Common Lisp form out of it.

In the fifth chapter we consecutively continue the topic by creating the core of the Logo primitives - fundamental operations and commands. For this purpose it will already be necessary to face up to the *Common Lisp Object System (CLOS)*.

In this way, also programming graphical user interfaces (GUI programming) will be introduced in several chapters as we go along – and several further useful skills.

If this approach is successfull, the readers will have gained enough know-how for searching the to-do lists for a starting point of their own participation after finishing this introduction.

I hope, they will have enjoyed the process of achieving.

One special note about the quality of my programming approach: I am not a professional programmer. All of this happens in my spare time. As a consequence, the solutions I propose will not be ideal of course. They emerge while I try to deal with the problems I encounter with the aid of the informations I explore *how* to deal with them – in the way I seem to interprete each problem and the corresponding informations. As well, I cannot guarantee that I use the technical terms of the art of programming always completely correct. My 'hands-on approach' is a constant learning process. Here I make all the flaws in my reasoning visible. This might become helpful since in this way, it could be relatively easy to find out initial and consequential errors of this implementation. Anything presented here should be treated as preliminary. But, this does not appear as a drawback to me. As long as the solutions work, even though maybe laborious, inefficient, frumpy, error-prone and so on, they offer learning opportunities. Defective preconceptions are the fundament of conceptual changes for better understanding. They are still more helpful than no basic idea of the subject in question at all. Thus, this document might serve as well as an introduction, as a documentation of the state of the development process and as a starting point of a discourse for finding the most appropriate solutions and interpretation of the necessary terms for this implementation of Logo as an extension of Common Lisp. – I am oriented towards the intersection of the style recommendations of *Peter Norvig and Kent Pitman*, *lisp-lang.org*, *Google*, *Ariel Labs*, *Carnegie Mellon University Lisp Group*, *Common Lisp Hyper Spec* and *Guy L. Steele Jr.* as suggested by *Alex Combas*[1] with an emphasis on the style guide of lisp-lang.org concerning object-oriented programming.[2] Thus the project has a style guide it conforms to.

[1] `https://github.com/foxsae/The-One-True-Lisp-Style-Guide`.

[2] `https://lisp-lang.org/style-guide/`.

Chapter 2

First steps of setting up the development environment

Started in March 2024

Common Lisp is the programming language we will use. We will use ANSI Common Lisp which is the US American national standard from 1994.[1] This standard is 'only' the description of the language. Anyone can *implement* this language description and in doing so realise a Common Lisp that conforms to the standard. Two popular proprietary standard implementations and IDEs for industrial use are *Allegro Common Lisp* from the US American company *Franz Inc.* and *LispWorks* from the British company *LispWorks Ltd.*. Both implementations extent the functionalities of the language standard for modern needs, such as contemporary AI programming features, own frameworks for the programming of graphical user interfaces (GUI), mobile runtime etc. They are professional products which means pricy though undoubtedly reasonably priced, aimed at technology companies, universities and other institutions.[2]

Anyway, we use the open source alternatives. They also extent the standard with additional functionalities. Those we do need we will use in a way, that we can bridge the differences between the implementations by something called 'read-time conditionals'. Like this, our source code keeps compatible with any Common Lisp implementation that conforms to the standard – and it doesn't matter, which implementation *you* use.

If you buy a licence of a commercial solution you get a lot of useful tools perfectly geared to each other out of the box. You pay for enviable convenience.

Using open source products on the other hand means, we have *to work* for reaching a good level of convenience. The trouble of open source implementations of Common Lisp is not that Common Lisp in general is a niche language. The trouble is, *because* it is a niche language mostly used by professionals, there seem to be hardly no motivation to put together a package of necessary and useful knowledge and tools geared to each other for lowering the *initial* obstacles. I try to start this job in this chapter.

In the year 2024, the most popular open source implementation of Common Lisp

[1] The name of the technical committee was *X3J13*, the name of the ANSI standard is *X3.226/1994*. Under these names, informations about the standard can be acquired. Under these names you can obtain further informations about the standard.

[2] They also offer good (limited) test versions. AllegroCL here: `https://franz.com/downloads/clp/survey`. LispWorks here: `https://www.lispworks.com/downloads/index.html`.

seems to be *Steel Bank Common Lisp (SBCL)*. Even if it may appear less suitable for beginners, it is very well suited for production. The debugging process with SBCL is not harder than with C++ for instance and I'm here to share my experiences for capacity building. Thus possible objections to SBCL do not convince me (maybe they are outdated anyway). As a matter of fact, for bridging implementations I only would mind SBCL, Allegro CL and LispWorks. Since I only use SBCL, I simply provide the skeletons for the bridges which can be completed by developers who use other implementations like the mentioned two proprietary ones, or *Clozure CL*, or *CMUCL*. (*Clozure CL* is the actively maintained implementation you should find in the Apple Store.) Although interesting, I ignore *Armed Bear CL* – although actively maintained – which is a Common Lisp implementation for the Java environment. If I would use the Java Virtual Machine (JVM) I would prefer to reformulate this whole project in *Clojure* for several reasons – which are not of interest here. I would not use *CLISP* since this implementation seems to be abandoned. Very interesting is the probably youngest implementation *Clasp*, certainly, because of its focus on interoperation with *C++*. Sooner or later I probably will try it out and then maybe throw the bridge to it as well. I really want to use Common Lisp also as an interface for several C++ libraries.

As already announced, the most popular open source solutions for configuring an IDE of Common Lisp are *Emacs* and *VIM*. Even if Emacs, this highly customisable editor system, is more complex than VIM, I think the first steps in Emacs are more easily made (and I do not use VIM). Therefore I will explain how to use Emacs.

Lisp in general provides an interface to the programming environment which is called 'Read-Eval(uation)-Print-Loop' (abbreviated: *REPL*). Another suitable name is 'Listener'. The REPL or Listener is integrated in Emacs via *SLIME*, the 'Superior Lisp Interaction Mode for Emacs' for Common Lisp.[3]

Common Lisp provides 'batteries' of programming solutions just like any other programming language's 'ecosystem'. We will create our project using the same standard build facility *ASDF* which was used to make these programming solutions. Contemporarily, the most convenient way to load them is using the tool *Quicklisp*. With this tool we are going to load *CL-Project* at first, for creating the project skeleton, and the test framework *parachute*.

Since there is a very well pre-configured software package called *Portacle*, which already integrates several of the tools mentioned and a few others for Windows, MacOS and Linux, I will use this one. It eases the first steps.[4]

[3] Even if Lisp is a family of programming languages, a lot of people use 'Lisp' in the narrower sense of 'Common Lisp'.

[4] My original approach of explaining how to install and configure the major tools one after another and combine them is moved to the appendix. Since I still prefer it over Portacle without really asking myself why I assume this might indicate, that my mind still expects some advantages over the combined solution. (It might be related to the fact, that I use Emacs for more than programming in Common Lisp.) Thus, if you are interested to learn about putting it together on your own do feel encouraged to do so. Yet maybe, sooner or later I will decide to simply stay with Portacle and remove the former part, since the IDE solution optimised for working with Common Lisp is absolutely well done.

2.1 Installing *Portacle* i.e. Emacs preconfigured as an IDE exclusively for SBCL with Slime, Quicklisp and further tools.

The Github webpage of Portacle explains the first steps so you can simply follow its instructions: `https://portacle.github.io/`.

However, I also describe these steps so you can already measure up the effort. (It is not much, I promise.) Moreover, I can connect the description with the rest of this chapter.

You will download *one* file which contains everything you need. You should download it from the Github webpage. At the top you have three buttons for the packages for each of the three operating systems Windows, MacOS and Linux. They lead you to the link on the webpage to download the *latest release* for your platform.

For Windows, a self executing file is provided which will assist you. You click the downloaded .exe-file, you will be asked where to extract the file, you probably will chose your home directory, and a subdirectory named `portacle` will be created with all the necessary components inside. You *should not* chose a different name than `portacle` if you like convenience. The components are geared to this name. Within this directory you start `portacle.exe`.

For MacOS, the package is bundled in a disk image. You download the .dmg-file and extract it, probably in the `Application` folder. There you have your `portacle` directory. On any platform this name should stay unchanged unless you like to resign from convenience. *But* you have an intermediate step to do, before you can start hacking. You have to tell the operating system that it is ok for you as well to *use* the software you intended to install past the Apple Store. First, you have to start the app *Terminal*. If you have never done this: You have to click the *Launchpad icon* in the *Dock*, type 'Terminal' in the search field and then click *Terminal*. A window will open with a *prompt* which somehow looks like this:

```
username@MacBook-Pro   %
```

If you extracted the disk image in the `Application` folder you type the command

```
sudo xattr -dr com.apple.quarantine /Applications/portacle
```

at the prompt and press the `Return` key. This will release that folder from the quarantine and you can use the `Portacle.app`. (If you chose another directory than `Applications` you have to adjust that name.) As the description on the github page says: 'You may see several `xattr: No such file errors`, which can be safely ignored.' Even if you have to keep the app in its folder 'you can however drag the app into your dock to create a shortcut.'

For Linux, you download the latest release as a *tar archive*. Here, I want to add a bit to the description on the webpage. First of all, I suggest that you as well open the terminal. If you never have done this before: Press the key with the windows logo to the right from the left `Control` key. From now on, we will call this key '`Super`'

and I will abbreviate it with a lower case `s`. I assume that no matter what desktop environment you use, you now will have the opportunity to enter 'terminal' and at any place you will see a menu entry or an icon which is called 'Terminal'. Click that entry or icon and a terminal window will appear. The terminal prompt will somehow look like this:

```
username@computername: $
```

Here you will first make a directory called `common-lisp`. Type at the prompt:

```
mkdir common-lisp
```

and press the `Return` key. I assume again, that you downloaded the tar archive into the `Download` folder. Now we will `move` the file into the `common-lisp` directory. ('Folder' and 'directory' are synonyms, its just a bit for diversion.) I assume, that the filename will still start with '`lin-`' and end in '`xz`'. Enter:

```
mv  /Downloads/lin-*xz  /common-lisp
```

If I say: 'Enter something.' I always mean: 'Type it and then press return'. (Of course, if you downloaded the archive into another folder than '`Downloads`' you have to adjust the name.) Now you have to `c`hange the `d`irectory *and* `e`xtract the `tar` archive:

```
cd common-lisp && tar xzf lin-*xz
```

After that you can open your filebrowser, navigate to the `portacle` directory and click or double-click the file `portacle.desktop`. Unmodified, this file is also bound to the `portacle` directory. But, if you are using a current version of Gnome as your desktop environment, like many people do, we can make it a bit more convenient. (Maybe the other desktops use the same ressource, I did not try.) In the terminal we `c`opy the file into a hidden directory. It is not *really* hidden. It is just not displayed as long as you don't want to see it. The names of all hidden folders and files start with a dot. So enter:

```
cp portacle/portacle.desktop  /.local/share/applications
```

Again I assume that you use a Gnome desktop and therefore you can use the editor *gedit*. So enter:

```
gedit  /.local/share/applications
```

A new window appears with a content like this:

```
   [Desktop Entry]
Type=Application
Version=1.0
Name=Portacle
Comment=Portable Common Lisp IDE
Exec=bash -c 'cd $(dirname %k) && ./portacle.run'
Icon=/home/<username>/common-lisp/portacle/portacle.svg
Terminal=false
Categories=Development;Common Lisp;Lisp;Text Editor
```

You can edit these lines but only have to modify the sixth that starts with `Exec=`. Delete the rest of that line *after* the equality sign. Then copy and paste `/home/.../portacle.svg` of the next line after `Icon=` in the sixth line. If oddly the line you just copied ends like this, `...//portacle.svg` simply delete one of the two slashes. Now, delete `svg` and replace it with `run`. The hole content should now look similar to this:

```
   [Desktop Entry]
Type=Application
Version=1.0
Name=Portacle
Comment=Portable Common Lisp IDE
Exec=/home/<username>/common-lisp/portacle/portacle.run
Icon=/home/<username>/common-lisp/portacle/portacle.svg
Terminal=false
Categories=Development;Common Lisp;Lisp;Text Editor
```

Now save the file and quit the editor. You may also close the terminal. If you press `super` now (you remember, the button with the Windows logo to the right of the left control key, which I later will abbreviate with `s`) and start to write `portacle` you should see the Portacle icon. – You may click it, little butterfly.

Some words on the different descriptions: It may seem, that the 'Windows way' is the most convenient, followed by the 'Mac Way', and Linux is just too laborious. Actually, if you do it a lot – configuring things with the terminal, you become used to it very fast. The description might be longer, but the empowerment is also stronger. Windows and MacOS are designed for people who do not want to be too much concerned with the internals. Therefore these systems hide a lot behind standard solutions which are useful in many situations but become obstacles if you want to make custom changes. The little extra step in the description for MacOS shows, that it has something in common with Linux. The latter behaves like a Unix, MacOS *is* a Unix. Thus you can also get your hands on the system up to a certain extent, I assume. I think, the reason of being forced to explicitly release something from quarantine, as seen above, is to discharge the Apple support team. If you install software from the Apple store ordinarily, the company can guarantee that this software is fine. If you use something from an unofficial source they can say: 'Well, this is out of our responsibility. Now you're standing on your own feet, young Jedi.' In Linux you are responsible for everything you do on your own, all the time. Yet you also *can* do everything on your own. And if you acquire the knowledge to do so, it very often is *much* more convenient to use the terminal for directly managing your system instead of clicking around. You do not have to care for even more intuitive new GUI control concepts, you do not have to search for menu entries and submenu entries – you simply control everything from the terminal which stays the same and conflate several clicks into one line of instructions in the terminal. The same applies for modifying configuration files manually. You also gain a better understanding of the structure and the 'adjusting screws' of the system and where, as well as how, to squeeze them – if you need it. The description is only longer in the beginning but becomes much shorter as soon as a higher level of experience might be assumed. That is the state in which you may be annoyed if you are forced to use a system where you are limited to the GUIs – or read tutorials about it. You will always want to have a choice: 'For this I want a graphical user guidance with its standard solutions. I am too lazy and don't want to read manual pages about it. But for

that, using a graphical user interface is much too tedious and annoying for me. I want to know how I can control and customise it on the command line. Let's see if anyone had the same problem and posted a useful line that I can adapt to my situation.'

However, this document is not about user perspective's on operating systems. So let us proceed.

2.1.1 First steps in Emacs and Slime.

No matter what operating system we use, we are all now sitting in front of a preconfigured Emacs that is optimised as an IDE for Common Lisp.

At this point we need to establish some conventions. The `Alt` key or the `Command` key of the Mac keyboard will be called `Meta` and abbreviated with a capital `M`. The `Control` key will be abbreviated with a capital `C` and `Shift` with a capital `S`. As mentioned above, the abbreviation of `Super` is a *small* `s`. This button is usually not needed.[5] You can change `M`, but by default it is bound to `Alt` or `Command`.[6]

In Emacs, everything is controlled by keyboard shortcuts. These are combinations of the already mentioned keys mostly with the letter and number keys. If keys are pressed together they are connected with a hyphen, e.g. `M-x` or `C-g`. Keep the first key pressed and tap the second one, then release both. Just try it out now. Type `M-x` and have a look at the bottom line of Portacle (Emacs). 'This area is called [...] **minibuffer window** where prompts appear and where you can enter responses [a special **echo area**].'[7] You see '`M-x`' appear and probably some further entries. Now type `C-g`. You see '`Quit`' for a while until the mini buffer is cleared again automatically. `C-g` is very helpful in a situation, in which you mistyped something and want to clear the minibuffer or to stop something. We abort a partially typed command *or* an executing command with `C-g`. A shortcut by the way for typing `M-x` is the `Menu` key right from `Alt Gr`.[8] Just press it without any other key and look at the minibuffer.

[5] The keynames 'Meta' (symbol: ◆) and 'Super' (symbol: ❖) were already in use in the days of the initial release of Emacs in the year 1985, also 'Hyper' (✦). The 'helm symbol' ⎈ (a ship's wheel) by the way indicates 'Control'. (Besides, Windows 1.0 was released in the same year, as one of several graphical extensions which already existed before Microsoft's attempt – also the concept of 'windows' with this denotation.) The old keynames are a beautiful historico-cultural fact preserved in the terminology of Emacs that in turn is actively maintained to this day. – Besides, the mentioned keys were physically present on keyboards for computers that were optimized for Lisp: 'Lisp machines'. They were arranged in a row with `Control` to be used to play different 'chords' with one hand in combination with the other holding a character key and if need be also `Shift`. Like this, it was possible to bind several additional characters to one key, especially math symbols – a concept which I am very attracted to. (All of this informations can be found on English Wikipedia.) Today the functions of `Meta` and `Super` are mapped to the available keysets without the handy possibility of 'chording' in the original way.

[6] Actually, you can adjust nearly everything in Emacs to your needs. But here, we do not want to. – 'Alt' by the way abbreviates 'alternate' thus it is equal to 'Option' on a Mac keyboard. But there `Command` is bound to `Meta` by default, as mentioned.

[7] GNU Emacs Manual – The Organization of the Screen: `https://ftp.gnu.org/old-gnu/Manuals/emacs-20.7/html_chapter/emacs_5.html`.

[8] 'Alt Gr' stands for 'alternate graphic *character*.' – Maybe I try one day to rename and reconfigure `Alt Gr`, `Menu` and `right Shift` to start chording additional symbols – which then need to be bound to character keys as well. Maybe inspired by the keyset of the *Space-cadet keyboard*:  . Much work but long overdue in times of UTF-8.

Take notes from the shortcuts you learn here. This is good for internalising them.[9]

The screen of the Portacle window is splitted in four areas. The first area is the menu bar. Below the menu bar there is an area called *buffer*. Here we write our code. If you look at the line in the middle of the screen you see a nice logo of Lisp, and `*scratch*` and further informations. That is the bottom line of that buffer. The area with `; SLIME 2.24` in its first line is a second buffer. It's bottom line contains another pretty Lisp logo, `*slime-repl sbcl*` and further informations. The words surrounded by asterisks are the buffer names. You know the minibuffer at the bottom of the Portacle window already.

Just to be sure that you didn't click elsewhere on your desktop meanwhile just click on the title of the Portacle window. Now the cursor in the *scratch* buffer should be filled and should blink. If you start typing, characters will appear.

Now we go on with our conventions. A hyphen symbolises that keys are pressed together thus no hypen means the key is tapped alone. While in the `*scratch*` buffer type `C-x o`. Before you tap `o` you have to release `C-x`. The cursor is now blinking in the `*slime-repl sbcl*` buffer. If you type `C-x o` once more, `*scratch*` is active again. We *switch* buffers like this. (The `o` represents `other` because we switch to another buffer.

Of course, you can also use the mouse to click in the buffers. If you guide the mouse cursor to the bottom line of the `*scratch*` buffer, you will see that its shape is modified. An information appears for a short while and you can read how you might resize the buffer with the mouse, how you can 'make the current buffer occupy the whole frame' and how you can remove the `*scratch*` buffer from display which consequently will make the remaining `*slime-repl sbcl*` buffer occupy the whole frame.

If you have tried (feel encouraged to do so) you might ask yourself how to bring each buffer back. For this we use shortcuts again. Just press `C-x 2`.

Strange, right? Now you have two buffers again but they are the same. You have *three* ways to get the SLIME buffer back.

First: You move the mouse cursor on the buffer name in the bottom line of a buffer. You will see a message in a little box near the cursor. Just follow its suggestions: Click with the left and the right mouse button and look what happens.

Second: You click the `Buffer` entry in the menu bar with the left mous button. Then you see a list with all open buffers. If you click the buffer you want to see again, you will success.

Third: As you can imagine already, *of course* you can control everything with the keyboard as well. Just type `C-x b` (`b` for *buffer*) and you see the names of the opened buffers in the minibuffer at the bottom of Emacs. You can either enter one of the buffer names or you can use the left and right arrow keys to change the activ (bold) entry near 'buffer'. If you press `Ret` you successfully change the buffer.

It is obvious, that we still can do many things in this configuration. If you type `C-x 3` you see the difference at once. Of course you can stretch the window in which Emacs is running. So you can arrange buffers side by side, and also further buffers below. There are not more than three possible combinations of buffers.

[9] But also recommendable is this cheatsheet: `https://www.quora.com/What-is-the-best-Emacs-cheat-sheet` and of course the GNU Emacs Reference Card: `https://www.gnu.org/software/emacs/refcards/pdf/refcard.pdf`.

Now you should easily get the SLIME Repl back. It doesn't matter whether you prefer to open it in the upper buffer or the buffer at the bottom [10]

Now, let us pretend we would have a problem in the REPL and it is lost in an infinite loop for instance. Enter the combination `, q` while your acive buffer is the Slime REPL. As soon as you type the comma, you see `Command:` in the minibuffer. If you type `quit` and confirm with `Ret`, we get rid of the REPL. In the minibuffer you see now `Connection closed.`

... But – actually, we want SLIME *back*. Easy! Just enter `M-x slime`. If Emacs asks you, whether you want to start another inferior process, answer no. Than the REPL wasn't closed for some reasons and will appear again. But usually, you connect to a fresh REPL, which you only might want to rearrange (above, below, beneath whatever).

Now – we dealt *a lot* with the REPL already. But the other buffer might have changed from time to time, but we couldn't use it on our own. Just change to it with `C-x o` if you're not already there. The shortcut for opening a file is: `C-x f`. In the minibuffer you see `Find file: ....` There you see the actually directory you're in now. Again with the arrow keys you can switch. You chose different files. With the `delete` key you can *move up* to the respective superdirectory. If you chose a subdiretcory with and press `Ret` you walk deeper into the directory structure.

After you typed `C-f` and you agreed on a location with the others, you might either chose a file in the directory of your choice, or you change to the desired directory and type in the name of the file you want to use.

If the file is open and you wrote some good stuff – perhaps you would be happy to make it reproducable: Of course, you can not only open but also close a file and write its content to disk: Therefore you type `C-x C-s`.

2.1.2 Postconfiguring Portacle.

2.1.3 First steps with Quicklisp.

Before we talk about Quicklisp, there is one important thing to do: If Portacle is not active click into the Slime buffer. If your focus is in the upper buffer switch to the REPL with `C-x o`. Now press `M-x` and write '`paredit-mode`' in the minibuffer and press `Enter` or `Return` . Maybe it appears already automatically after pressing `M-x`. Later, it probably will.

Like this, you switch off the `paredit-mode`. The mode is fine. You switch it on again if you repeat the steps to switch it off. The `paredit-mode` assists you with the parentheses in Lisp which is great and welcome. Only sometimes, if you correct or form or want to insert something, the automatism is a little bit annoying. If the automatically set brackets do not fit your actual needs and or you cannot delete them properly, simply switch off the mode, correct the code line and switch it on again. If you cannot delete an opening or closing paranthese simply switch the mode off. You always can switch it on again.

A second important hint. The REPL maintains a history of all the entered – so far we say – the entered 'things'. To walk through this history *backward* its `C-↑` and

[10] If not: Just try one or all of the three ways to get back on track.

forward its `C-↓`. The 'things' that appear can be modified and entered again with `Return`.

You enter things at the REPL input *prompt* you see in the buffer `*slime-repl-buffer sbcl*`. It looks like this:

```
CL-USER>
```

Just to fill the REPL buffer with a lot of lines, please enter:

```
CL-USER> (ql:system-apropos :sql
```

What you just entered is the Common Lisp *function* `system-apropos` of the Quicklload `package`, abbreviated with `ql`. This function shows you the `system`s related to the `keyword` at the end of the expression: `:sql`. Here you already get a first impression of some Common Lisp terms. We will deal with them in more detail. So far: A Common Lisp `package` is pretty much the same like a *namespace* in C++ – a way to prevent name conflicts of functions and variables defined at different places but maybe used together. A Common Lisp system is pretty much like a library in C/C++, Java, Python etc. It's a pre-built solution for a class of problems – just like programming a GUI, or programming search algorithms, or whatever. This common Logo setup is supposed to become a `system` which also contains packages of so-called `symbol`s so that we can handle potential name conflicts with other systems we need at the same time.

But my first intention was to show you another shortcut of Emacs, so you can clear the SLIME buffer: `C-c M-o`. This is the combination for `M-x slime-repl-clear-buffer` and can be quite handy sometimes if you ... yes, wish to have a clear buffer.[11]

[11] If you someday want to call this Emacs function from Common Lisp: It *is* possible although not recommended. You can set this configuration variable in your Emacs configuration file like that: `(setq slime-enable-evaluate-in-emacs t)`. Then you can define such a function: `(defun clear-emacs-buffer () (swank:eval-in-emacs '(progn (run-at-time 0.2 nil 'slime-repl-clear-buffer) nil)))`. After that, always when you call the function `(clear-emacs-buffer)` that very SLIME buffer will be cleared. But enabling slime to control Emacs functions could be a security risk. Therefore it is recommended to clear buffers with the shortcut I just showed you.

Chapter 3

Introduction to Common Lisp along with some terminology.

Started and provisionally completed in April 2024

In this section I want you to use the REPL to let you experience various basic concepts and to let me introduce some technical terms which are very useful for understanding and talking about Lisp. And since we want to implement a Lisp dialect (*Logo*) in another Lisp Dialect (*Common Lisp*) as an extension it will not be unlikely that we will talk about Lisp in this document while we are using it. I will go on adding some additional informations in the footnotes. These shall be little hints to answers which might appear in the readers mind incidentally. And perhaps they are enough to focus on this document again or to speed up the distracting own research a little.

This section is also meant as an introduction as it is meant as a set of subsections you may return to, if later in the document you stumble over something which is unclear and you want to read about the concept again. It also is meant as a starting point for your own further research. If this is your first time reading about the things I'm discussing here, then you've already read about them when you pick up a professional introduction. I will suggest a few in the course of this section. If it is the other way around, you might compare my understanding with yours, criticise mine or add something to yours and again you deal with the concepts. I suggest that you pretend to be a rookie and therefore try to suppress your knowledge of different languages for this introduction. I have found that it helps me a lot to read several books on a topic at different times and to exchange views on it. Thus: I lay out the things here as I think to have understood them. But I by myself aim at those interested people who learn a programming language for the first time. Again: This is not the document of a professional computer scientist or a software engineer. I am a trained bookseller, was once a works council member, organised a labour dispute, completed my law studies, worked part-time in trade union legal protection, switched my focus from labour law to social law, was a clerk and occasional lecturer on legal topics, I thought a lot about legal didactics such as practical and theoretical philosophy before I had my Eureka moment about what to do next in my professional life and at the moment, as I write this chapter, I am still on the path to becoming a primary school teacher – pretending my project is at a a more advanced level than it actually is using it to organise the other things I need to figure out before I can introduce them here. In fact, I use the

writing process to clarify for myself what I need, to reflect on my understanding process so far, and to document my next steps. I expect that I will have been a teacher for quite some time by the time this is finished. So you see: Everything I did until today had nothing to do with gaining professional knowledge of programming languages. My programming experience is just that of a hobbyist who grew up with computers since 1985 and has always dabbled around programming with Turbo Pascal, C and later a bit of Python, but immediately fell in love with Lisp when he first met Logo in 2014, through which concepts appeared concrete to him that previously remained abstract.

3.1 The regular case of Lisp *forms*

The basic structure of a Lisp expression is of this *form*:

```
(operator argument-1 argument-2 ... argument-n)
```

Thus, the expression is enclosed by parentheses, the first element is the operator, the following elements are the arguments passed to the operator to be somehow modified. We will call the first element the *head* because the head comes first.

At the REPL prompt, enter the following expression:

```
CL-USER> (+ 1 2 3)
```

This will yield the expected result. Thus `1`, `2` and `3` were passed to `+` that summed them up.

The form we just entered is the basic syntactic structure of any Lisp dialect. It is an expression – called a *Symbolic expression* or *S-expression* – which is enclosed by parentheses. S-expressions with this basic structure are *function calls*.[1]

3.2 Code Mode and Data Mode

Now enter the following at the REPL:

```
CL-USER> (quote (+ 1 2 3))
```

Instead of `6` the REPL echoes `(+ 1 2 3)`. What did happen? Well, we modified the form, yes. And we did not change the basic structure we added a new head, `quote`, and passed to it the following element `(+ 1 2 3)` which then was quoted as intended. – There is a shorthand for `quote`. Enter:

```
CL-USER> '(+ 1 2 3)
```

and it has the same effect.

Let us consider the long version first and compare it with our first function call. We can realise:

[1] This basic structure – parenthesised expression with a head and its following elements – is also called *Cambridge Polish Notation* because Łukasiewiecz invented the prefix notation with the operator as the head but without brackets, and *Quine* used this notation *with* brackets in Harvard, Cambridge, MA, USA. It happened to be useful in Lisp independently without Łukasiewiecz or Quine in mind, but it was recognized retrospectively.

1. A function call always has a head which tells us what to do with the following elements.

2. The following elements can also be parenthesised expressions *nested* in the function call.

3. (I assert:) We can only `quote` *one* thing at a time but we can add many items with `+`. Hence, some function calls go along with just the head plus one following item – say *argument*. Other function calls have more than one arguments.

4. As a consequence we might expect that there might also be function calls that consist of their head alone. (And we won't be disappointed.)

Now for the shorthanded version, we can realise:

1. If we precede a function call with an apostrophe, the head does not operate on its operands. Instead the form is just quoted.

2. The apostrophe in front of a form appears like a switch:

   - 'Apostrophe set' = 'just quote'
   - 'Apostrophe unset' = 'apply the arguments to the head of the form'.

From now on, we will distinguish *data mode* from *code mode*. 'Apostrophe set' is *data mode* and consequently 'Apostrophe unset' is *code mode*. As we could see, a form in both modes appears nearly identical. Actually, it is. This feature is called *homoiconicity*, if structures that contain source code can be treated as data without changing them, simply by 'flipping a switch'. This is an important feature, but we will return to it later.

3.3 Lists in Lisp

Here, I want to point out that the structure of every parenthesised expression is that of a *list*. In any Lisp, a list is a special data structure with the main purpose to contain the code. Any executable piece of a Lisp program is a nested list which is processed by the Lisp environment. That's why *Lisp* is the acronym of '*Lis*t *p*rocessing'.[2] It is also the reason why using lists as a data structure for everything is very tempting in Lisp. But in most cases other data structures are more appropriate.[3] Anyway, for complex structures just like source code they are a very good match.

For short again: `(+ 1 2 3)` is a list in *code mode* and `'(+ 1 2 3)` is a list in *data mode*. If we enter the list in code mode, its a function call of `+` with the two arguments `1`, `2` and `3`. We say: 'The form *evaluates* to `6`.' If we enter the list in data mode and it is echoed we say: 'The form evaluates *to itself*.' – Well, we also can say that the form *is* or *will be* evaluated to `6` or *to itself* since of course no form does anything on

[2] The idea is borrowed from an Assembly language that preceded Lisp, the *Information Processing Language (IPL)*.

[3] The concrete structure is that of a linked list, a lineary data structure that is ineffective for handling huge amounts of data which is the regular case of software applications. We'll come to that later.

its own. Either we evaluate the form virtually in our heads, or our Lisp environment evaluates it and lets the computer carry out the appropriate actions. Therefore, if we talk of a form that *is* 'self-evaluating' this term is an adjective, we are talking of a *self-evaluative form*.[4]

To experience the difference of a list that can be both data and code and a list that is just data enter this:

```
CL-USER> (a 1 2 3)
```

Now, I have your attention again, huh? In the REPL there appeared a lot of lines like these:

```
   ; in:  A 1
; (A 1 2 3)
;
; caught STYLE-WARNING:
; undefined function:  COMMON-LISP-USER::A
;
; compilation unit finished
; Undefined function:
; A
; caught 1 STYLE-WARNING condition
```

And in the upper buffer we probably see this:

```
The function COMMON-LISP-USER::A is undefined.
Condition of type UNDEFINED-FUNCTION


Restarts:
0:  [CONTINUE] Retry calling A.
1:  [USE-VALUE] Call specified function.
2:  [RETURN-VALUE] Return specified values.
3:  [RETURN-NOTHING] Return zero values.
4:  [RETRY] Retry SLIME REPL evaluation request.
5:  [*ABORT] Return to SLIME's top level.
-more-

Backtrace:
0:  ("undefined function" 1 2 3)
1:  (SB-INT:SIMPLE-EVAL-IN-LEXENV (A 1 2 3) #<NULL-LEXENV>)
2:  (EVAL (A 1 2 3))
-more-
```

[4] Even if this clarification might appear unnecessary to native English speakers, I think it is helpful. If a form is imagined as 'self-evaluating' in the present progressive tense this might somehow be a useful picture but could also become a distracting mind game of forms as solipsistic entities that are performing acts on themselves for yielding themself again. This is funny but leads on a meta level ('meta' as 'beside it') which might also appear obscure. And there is no need to support a sense of obscurity because Lisp is utterly down-to-earth.

For now, we are only interested in getting rid of it. But anyway, let me introduce you to your very best friend in the Lisp environment: Your *debugger*.

Never treat it as an enemy or an unwelcome stranger. If it makes you angry, you are angry at yourself for confronting your own mistakes. Be kind to yourself. The debugger is straightforward and demands knowledge. Anyway it also supports the learning processes because it gives a lot of hints. Since learning to program is a try and error process like any learning process, the debugger is one of our major learning tools. Here we learn at first hand: 'Undefined function: A' in three versions.

The reason for feeling annoyed by the debugger is because it blocks the flow. The REPL refuses your input. Actually you should focus the upper buffer now. If it is not already active, type `C-x o` to switch or click it with your mouse. As I just wrote, for now we only want to go on without minding the details of the debugger. So just mind the list with the title '`Restarts:`' and type `5` for `[*Abort]`.

So, you know how to '`Return to SLIME's top level`' quickly and you realised, that: A list entered in code mode with a head that does not represent any kind of command will get the debugger on the scene with a useful error message. It's true, up to now we didn't define a function with the name '`a`' and there is no such command 'built into' the Lisp environment. So Lisp cannot evaluate this form. In code mode it doesn't mean a thing. Of course, in data mode, it means itself as you expect, try:

```
CL-USER> '(a 1 2 3)
```

So this is a list which is only useful as pure data. Still, we can call the first element the *head*. But we will also call it `first` as we call the `second`, the `third` and the `fourth` element. The `last` is a different thing, so is the `butlast` as well as the `rest`.

Try it out. Call the data list with any of these functions, just replace the head and explore the results:

```
CL-USER> (first '(a 1 2 3))
```

3.4 Lisp Forms in General

Now, since you know, how to leave the debugger and how to extract the first element of a list in data mode you may do something naughty. Enter:

```
CL-USER> (eval (first '(a 1 2 3)))
```

The naughty thing is not the fact that we caused a helpful error message: '`The variable A is unbound.`' (You might use the actual number of `[*Abort]` or `[Abort]`. Of course they have a different effect as you can read in the list. But for now we do not care, you only see already that the list of possible restarts may change.)

We were naughty because we used `eval`. In most cases, we will *never* use this function, *directly*. (I like norm contradictions.) Besides, we did not need to use `eval` here. We could have typed `a` alone without parentheses as well. We will deal with `eval` later so don't mind for now. Just keep the feeling for the effect and remember that using it is basically naughty. As you already pointed out, the name of the function is the abbreviation for `eval`*uate* and it is one of Lisp's three core functions `read`, `eval` and `print` which are applied in an eternal *loop* in the REPL. Like using lists as the data type for anything is tempting so is using `read` and `eval` much more often then necessary. Be that as it may, not every convenience is preferable.

3.4.1 Forms in Data Mode

However, now you know: If you enter a letter at the REPL or more generally, if you let Lisp evaluate a letter on its own, this letter is *not* self-evaluating – the REPL will not echo it. You already saw, that LISP expects it to be a *variable* and had to realize that the `a` did not represent anything up to now. The same thing would happen with any alphanumeric combination, also mixed with hyphens and further characters.

But – you guessed it: If you switch to data mode, if you *quote* it is perfectly evaluable to itself.

```
CL-USER> 'a
```

The lower-case letter is echoed in upper-case and we will call it a *symbol*. We already heard of S-expressions above so there is a special connection with symbols in Lisp. But for now, we just keep in mind:

- A symbol evaluates to itself if it is quoted i.e. if it is in data mode.
- Every combination of alphanumericals also mixed with hyphens and further characters are symbols.
- Symbols may represent something else to which they are evaluated to in code mode.
- Every head of a form which is evaluable in code mode is a symbol.

Numbers are no symbols in Lisp.[5] Hence it makes no difference whether you enter them quoted or unquoted, they will always be evaluated to themself. You can try on your own by entering single numbers in either way. You also can check it in a function call:

```
CL-USER> (+ '1 2 '3)
```

Since you know that the apostrophe is the shorthand for `quote` of course this is equivalent:

```
CL-USER> (+ (quote 1) 2 (quote 3))
```

So far we can say in general:
Numbers are are always self-evaluating, symbols in data mode are self-evaluating and lists in data mode are self-evaluating. These are *self-evaluating forms*.

3.4.2 Forms in Code Mode.

The first thing we learned about forms was that the basic structure is that of a function call in code mode. If I just write of a 'form' I mostly refer to that: `(`***head*** $argument_1$ `...)`.

There also are *special forms*. Mind this:

[5] In our mind they are, but since a computer is so very close to numbers these are the most concrete things to it. Therefore technically it is much more efficient to treat numbers directly as numbers.

```
CL-USER> (if (= 3 4) 'really?  'no-way!)
```

What happens? Our first element of this list in code mode is the head. The second element might be seen as its first argument and the following elements as its second and the third. But how are they modified together to yield a new result? They are not. In fact: `if` asks whether the next expression is `True` or not. If it is not – like above – then the *last* expression will be evaluated. If it is `True` the penultimate expression will be evaluated instead.

In special forms, another evaluation order is applied than in regular forms.

Let's take a step back for the moment: Which terms have we learned so far?

- There are Lisp expressions which are forms.
- There are self-evaluative forms which are numbers and symbols as well as lists in data mode. All of them are Lisp forms. – All of them are Lisp expressions.
- There are lists in code mode which are regular function calls. Since this is the basic structure of Lisp code, I mostly refer to them if I only speak of a 'form'. These forms are Lisp expressions and they evaluate to a value not to themself.
- There are also lists in code mode that deviate from the regular syntax. They look the same at first glance but they behave differently than a function call because of a different internal evaluation order. These Lisp expressions are called *special forms*.

We easily see that the terms 'expression' and 'form' overlap each other. Yet 'form' is the narrower concept.

Special forms generally are 'built-in'. Any Lisp environment is equipped with them. Together with the *primitive functions* they represent the 'key vocabulary', the core of the Lisp language which is part of the Lisp environment.[6]

Now, let us consider the last *form* of Lisp expressions in code mode and learn another important aspect of Slime/Emacs by the way. Type this line, which unprepared looks awkward. Don't mind, at the right time it will become perfectly lucid:

```
CL-USER> (do ((i 1 (1+ i))) ((= i 0)))
```

Such inappropriate editing is the reason for the joke that Lisp would stand for 'Lots of Irritating Superfluos Parentheses'. It is a consequence of the strict and therefore reliable syntax and of course it can be handled very well. But unprepared and without an introduction it might feel less helpful as it actually is.

Anyway, the first thing you recognize is, that the REPL seems to halt. I let you formulate a silly infinite loop. Thus the poor REPL does a lot. It adds one on one eternally because the condition for stopping the loop is that we reach 0. Impossible.

But we can interrupt infinity with a snap. Just type `C-c C-b`. In the upper buffer you see our good friend debugger again that says: '`Interrupt from Emacs.`' Let's chose `[*Abort.]`, and our vicious circle is broken. We can go on hacking in the REPL

[6] Lisp implementations are mostly written in C, I guess. Thus this 'key vocabulary' is implemented in the C language. Yet after compilation, the object code which is executed by the computer is neither C nor Lisp but the combinations of ones and zeros the computer can deal with.

and the computer can cool down. **`C-c C-b` is a very important shortcut. There will always be an occasion to lose ourself in an infinite loop. So note it!**

If we don't mind all the parantheses but only the outer ones of each expression we realise that `do` is the head followed by two nested lists. There *can* actually be three such forms after the head '`do`' and a lot of things will happen in the background.

Like special forms these forms may follow the basic syntactic structure but have different evaluation orders than a regular Lisp function call. They differ from special forms in that they are not 'built-in'. They are defined in Lisp code and their inner workings can be examined within the Lisp environment. We can call them 'custom special forms' although a lot of them are supplied with the Lisp environment. But we can define our very own macros as often as we want. The `do` macro is one of the core macros. A very old one. Many '(Common) Lispers' prefer the `loop` macro over `do`. It is more optimised but/and it provides a completely different syntax. It is a specialised 'loop language', generally speaking a domain specific language (DSL) as a standard extension of Common Lisp.

Thus, whether a macro modifies the syntax of Lisp expressions just a bit, or it turns it upside down, its call is a regular Lisp form.

> **Now we can say in general:** Evaluable lists in code mode are *forms* which are either
>
> - function calls,
> - special forms or
> - macro calls.

Not the function *operator*, nor the special *operator* or the macro *operator* are forms on their own because in code mode they do not evaluate to themself and in data mode they are only symbols. But their complete evaluable call, the executable expression is the function form, the special form and the macro form.

3.5 Defining in Common Lisp

We can define a lot in Common Lisp. We can define parameters, variables, constants, functions, methods, classes, packages, systems, macros and so on.

3.5.1 Defining variables

Variables in the global environment

Let us start with defining a *dynamic variable* or *special variable*. Superficially the main aspect of them seems to be that of being a 'global' variable. A dynamic variable is defined at the top level that means, it can be accessed from anywhere in a program. This can be a good thing it also can be not so good. If we *need* a dynamic variable we use it.

Actually, there are two ways of defining a dynamic variable. The first is this, enter:

```
CL-USER> (defparameter *special-1* 'look-at-me "A silly assignment.")
```

The first thing we discover in this *form* is that we define this dynamic variable as a parameter? Well, in mathematics 'parameter' indeed is a name for a variable which is treated a bit different than other variables.

The second peculiarity are the two asterisks enclosing the name `*special-1*`. We can have very good reasons for using dynamic variables but whether we do or we don't it is always wise to follow the naming convention for dynamic variables in Common Lisp. That means: The names of dynamic variables have these asterisks as 'earmuffs'. Whenever you use such a variable anywhere in your source code you always know that it is a dynamic variable because of its *earmuff notation*. This is always good.

The third element of this form is a symbol. It can be any Lisp form that evaluates to itself or to a value. This value is assigned to the variable name. Thus `*special-1*` in this case is also a symbol. Just enter

```
CL-USER> *special-1*
```

and you see, it evaluates to ... well, you see it right there.

Again we can discover that there is a last element in the `defparameter` form enclosed by quotation marks. This is a *documentation string*, for short *docstring*. It is optional but also helpful to be used. You can request it like this:

```
CL-USER> (documentation '*special-1* 'variable)
```

Do you notice the tiny little difference? In `(defparameter *special-1* 'variable)` the variable name is unquoted. When called with `documentation` we must switch to data mode. And we must also tell `documentation` that we request the docstring of a `'variable`. You easily can see the difference, just call the same form with `*special-1*` unquoted:

```
CL-USER> (documentation *special-1* 'variable)
```

We can also reassure ourselves that `*special-1*` is a symbol:

```
CL-USER> (symbolp *special-1*)
```

This simply yields `T` which stands for 'True'. The function `symbolp` takes one argument and return `T` if the argument is a symbol. The `p` at the end stands for 'predicate'. Predicates are test functions which return True or false whereby in Common Lisp false is something special. We will deal with it later.

If you want a more detailed description of the symbol `*special-1*` try this:

```
CL-USER> (describe '*special-1*)
```

Also mind the difference if you keep `*special-1*` unquoted.

Now, you learned some additional ways to examine a variable and a symbol. Let us go back to the second possibility of defining a dynamic variable, it is nearly the same way:

```
CL-USER> (defvar *special-2* 'why-not-me "Another silly assignment.")
```

You can examine this new dynamic variable in the same way and both variables can be modified in the same way:

```
CL-USER> (setf *special-1* (+ 2 3))
```

and

```
CL-USER> (setf *special-2* '(aw - some))
```

We also can alter both variables in one line:

```
CL-USER> (setf *special-1* (/ 7 8) *special-2* "heigh ho")
```

The operator `setf` is a macro which originally was a `f`unction while the `f` could also abbreviate 'field' or 'form'. In the SBCL description it is sensibly called `setting form`. However, `setf` is used as an 'assignment operator'. It is generally used to store values, say *data* (the second and the fourth argument) at *places* such as variable names (the first and third argument).[7]

Now, what is the difference between `defparameter` and `defvar`? You always have to initialise `defparameter` i.e. you have to name a place and specify the initial datum (`(defparameter *special-1* 'hello)`. If you call this form with a new initial datum this has the same effect than assigning it in a form with `setf`. On the contrary, you can call `defvar` with a name alone so that the defined dynamic variable is initially *unbound*. But later you cannot use `defvar` to reinitialise the variable like you can use `defparameter`. You only assign new data to a `defvar`ed variable with `setf`. As a consequence, a dynamic variable defined with `defparameter` can be redefined with the same form more easily. Insofar a dynamic variable defined with `defvar` appears a bit less flexible. But usually, `defvar` is preferred.

And this can make sense: Think of `defparameter` as a parameter of a mathematical function like $f(x) = b \cdot x^2$. This b shall be a `defparameter` and the x shall be the `defvar`iable argument to x. Therefore, we change b only once for each use case, while x changes several times as we study the behavior of the function.

If we treat `defparameter`s as a constant in this sense which can be easily modified for the next case, we might use this form for that kind of variables, which are used statically in our program in the sense of traditional global variables without their feature of being special.

Those `defvar`iables we want to use dynamically with their very feature of being special, we define with `defvar`.[8]

Finally, we use such global variables without the property of being special for actual constants that are intended to be immutable in any case. Enter:

```
CL-USER> (defconstant +custom-pi+ 3.1414 "A rough pi.")
```

A different naming convention catches the eye. The enclosing asterisks are replaced with plus signs. They also serve the purpose of highlighting a variable on top level although for a different reason. This global variable is 'write-protected', it cannot be modified with `setf`:

```
CL-USER> (setf +custom-pi+ 3.)
```

[7] Of course, the real *place* is addressed in the physical memory of the computer. But in the source code this place is represented by the symbol we chose as the variable name.

[8] We consider the feature of being special later.

(You already know how to leave the debugger.)

As you can realise by asking for its description, `+CUSTOM-PI+` names a constant *variable* but we just call it a constant:

```
CL-USER> (describe '+custom-pi+)
```

It is a variable that is intended to be initialized once and nevermore.

Lexical variables

If we can define top level variables that can be accessed everywhere of course we also have variables which have a much more limited validity. We call them *lexical variables* because they only exist within a *lexical scope*, are generally invisible outside that scope and are bound to the duration of its existence. In many cases, these are the preferred variables.

A typical way of defining them is within a `let` special form:

```
CL-USER> (let ((a 2) (b 3)) (+ a b))
```

The first form after the `let` contains two further forms. (First, always pay attention to the outmost parentheses of each of the nested subforms of a form.) We might understand this first nested list as 'the variable dictionary' of the *body* of the `let` form. Thus the 'dictionary' is `( (a 2) (b 3) )` and the body is `(+ a b)`.

Every name-value-pair in the first nested list follows the known scheme: the left expression is the variable name, the *place* to which the right expression will be assigned. Their lexical scope affects the body.

After this special form is evaluated the lexical variables `a` and `b` cease to exist.[9]

3.5.2 Defining functions.

For now, we only consider two ways of defining functions in Common Lisp. Together with the several described way of defining variables, we have developed a good starting point for understanding the different forms we will encounter later. The principle keeps the same.

The person who drafted the first theoretical version of LISP as an intermediate step to another programming language in mind back in the 1950s (*John McCarthy*) did not know much about the *Lambda Calculus* of *Alonzo Church* out of the 1930s, he said. It was just a loose inspiration for the notation of the theoretical draft. McCarthy was deeper inspired by *Stephen Kleene*'s theory of first order recursive functions. Anyway, the draft was applied mathematics. – No, we won't pore on that. I just want to introduce the *anonymous function* which we define with a *Lambda expression*. – From McCarthy's theoretical design to his young colleague's (*Steve 'Slug' Russell*) surprising implementation of LISP (thereby proving that a mathematical concept turns out to be a full-fledged actual programming language by itself), Lisp has been developed more

[9] For those who miss type definitions: Later, we also will tell the compiler explicitly which types it may expect and so the readers of our source code will have the desired hints to the types *they* may expect, too. But in the Common Lisp workflow that is a matter of refinement for the sake of efficiency (switching of the automatic type and security checking) when we feel confident to do so. Thus at this stage, we won't mind.

and more intensively. But still, Lambda is not *the* core of the language. However, the lambda expression is a typical and also *a* core feature of Lisp, and this is where the inspiration to adopt it in other programming languages originally comes from.

This could have been a footnote, but somehow I think it is not bad to bump your nose into it, at this point.

Anonymous Functions:

Back to exploration. Enter this:

```
CL-USER> (lambda (x) (+ x x))
```

Not very impressive, huh? The REPL doesn't do much except telling you something like this:

#<FUNCTION (LAMBDA (X)) 52BA2C7B>

At first, you see the structure of a Lambda expression. There is the *Lambda operator* followed by two lists. The first list is the *parameter list* and the second list is the *function body* in which the parameters are applied. The parameter list of the Lambda expression is also called the *Lambda-list*.

Even if you think: 'Why?', enter:

```
CL-USER> (lambda (x y) (+ x y))
```

just to let you experience, that of course the lambda list can have more than one element and how this affects the body.

This function really is unnamed. In the original notation of the Lambda Calculus the first lambda expression looks like this, where λ is an auxiliary symbol: $\lambda x.x + x$. The symbol more or less tells us: 'The parameter x is bound to the expression $x + x$.' If we want to apply that anonymous function we can note it like this: $(\lambda x.x + x)(2)$ and we already can evaluate it because we know, that 2 is the argument we insert as x and right to the left of it we see the instruction what to do with 2. We do not need to explicitly state: $(\lambda x.x + x)(2) = 2 + 2$.

In Lisp this works as well (even if it is not applied like this usually), try:

```
CL-USER> ((lambda (x) (+ x x)) 2)
```

Obvious: We nested the Lambda expression in a new list where the Lambda expression is the head and the 2 is its argument. Equally apparent becomes the use of the terms *parameter* and *argument* in this context: The arguments are the concrete values passed to the function and the parameters are their placeholders in the function definition.

Now, let us name an anonymous function. Enter:

```
CL-USER> (defparameter twice (lambda (x) (+ x x)))
```

Yes, this works. But if you want to call the function by its name, you have to take a detour.

What happens if you just call `twice`?

```
CL-USER> twice
```

Right. Again we see something like:

```
#<FUNCTION (LAMBDA (X)) 52BA693B>
```

Since we have heard of *binding* now we can appreciate, that the alphanumeric thing in curly brackets represents the concrete function and `(LAMBDA (X))` tells us, that the Lambda operator binds the parameter `x` to it. – But we cannot use it.

Maybe, we remember, that a function call is a form in form of a list. A list is parenthesised. Then let us try this!

```
CL-USER> (twice 2)
```

What? Lisp doesn't know that function? (Again, you know how to leave the debugger.) Yes, it does not. We defined it as a variable thus Common Lisp treats it as such. The symbol `twice` does represent the anonymous function but the function is stored at a different place, the place for variable values.[10]

But actually we *can* call the function by the name we gave it:

```
CL-USER> (funcall twice 2)
```

The function `funcall` takes a *function* as an argument. In our case, we even defined a function as data: We stored it in a variable and passed the symbol of the variable to `funcall`. So, roughly `funcall` applies the further arguments to the function it takes as its first argument and cares for the function's execution. For more details consider:

```
CL-USER> (describe 'funcall)
```

Functions that take functions as *input* are called *higher-order functions*. They are not entities on a higher level of insight. They are just functions that take functions as input, which can be very useful.

Yes, we will consider the usual case of a function definition soon, but it is very helpful to go through all of this, trust me.

This:

```
#<FUNCTION (LAMBDA (X)) 52BA693B>
```

as a whole denotes the *function object*, the thing in memory that does the work it is meant for. There is another function you can use to show you the function object of any function.

Again I would like to lead you astray for a moment, enter this:

```
CL-USER> (function twice)
```

You already expected, that you will have to get rid of the debugger, right? This cannot work because even if *twice* represents the anonymous function, it represents its function object already as a variable value. Thus we do not pass a function symbol to `function` but a variable symbol.

If you call a primitive, a built-in function by the way, the function object looks different, try:

[10] Later, we will learn more about symbols and their places. Just for those who heard a bit of *Scheme*, another Lisp dialect: Yes, in Scheme and dialects inspired by it, function values and variable values are stored at the same place. So, in Scheme-like Lisps you can define an anonymous function as a variable and call it as a function without a detour. Both approaches have their justification.

```
CL-USER> (function funcall)
```

Let us now call a Lambda expression with `function` and `funcall`:

```
CL-USER> (funcall (function (lambda (x y) (+ x y))) 2 3)
```

The function `function` *returns* the function object of the anonymous function and that is passed to `funcall` which applies the function to the arguments `2` and `3`.

To reduce the number of parentheses in such higher-order function calls, a shorthand for `(function ...)` exists. The following is the equivalent expression:

```
CL-USER> (funcall #'(lambda (x y) (+ x y)) 2 3)
```

Actually, *with a Lambda expression* you even do not need the shorthand `#'`:[11]

```
CL-USER> (funcall (lambda (x y) (+ x y)) 2 3)
```

However, the explicit use of the shorthand `#'` highlights this context which is helpful for me when I read the source code: I can realise where functions are passed as data to other functions even out of the corner of my eye, so to speak. To me, this is relieving. **That is why it is part of this project's style guide to *always* use `#'` even if it is not necessary.**

Thus, we also write this, for instance:

```
CL-USER> (apply #'+ 1 '(2 3))
```

even if we could write this as well:

```
CL-USER> (apply '+ 1 '(2 3))
```

So, why anonymous functions are useful? We can pass short ad-hoc functions to other functions if this is appropriate and if we only need that special anonymous function in this very special case. And there we can see whats happening *in* the function at the point where it is applied.

So, why the knowledge about anonymous functions is useful, even if I never want to use them? It is very unlikely, that you never will use them. Anyway, others will and therefore it is necessary to know the concept for understanding the source code. *And*, the debugger will very often if not always mention Lambda expressions. This structure is fundamental in Lisp. We can actually understand the programming language we use as the frontend for the backend, that creates the concrete Lisp source code from our more abstract Lisp source code, which is ultimately compiled to object code. This is the reason, why the structure of Lambda expressions are shining through a lot of other Lisp structures we use more frequently.

[11] In ANSI Common Lisp `lambda` is not a unique anonymous function operator. It is defined as a macro which combines the Lambda expression with `function` implicitly if that is not explicitly preceded. It also can combine it with `funcall` implicitly as seen in this context: `((lambda (x) (+ x x)) 2)`.

Named Functions:

The impatient shuffling of feet is over. Usually, we define functions with `defun`:

```
CL-USER> (defun twice (x) (+ x x))
```

Now, we can call `twice` immediately:

```
CL-USER> (twice 2)
```

And if we want a description:

```
CL-USER> (describe 'twice)
```

we might see something like this:

```
COMMON-LISP-USER::TWICE
```

symbol

```
TWICE names a special variable:
Value:  #<FUNCTION (LAMBDA (X)) 52BA693B>

TWICE names a compiled function:
Lambda-list:  (X)
Derived type:  (FUNCTION (T) (VALUES NUMBER &OPTIONAL))
Source form:
(LAMBDA (X) (BLOCK TWICE (+ X X)))
; No value
```

Every information should be easy to grasp now. Especially the double occupancy of the symbol `twice` as a special (= dynamic) variable (we defined at first) and as a compiled function (we defined now). Likewise the Source form isn't obscure anymore.

Apparently, 'defun' is a macro that connects its first input – a symbol name – with the Lambda expression which is the Lambda-list (= the parameter list) and the function body, so its second and third input or the third and fourth subform of this whole nested list.

Mostly, we define named functions thus we only use `defun` and will not occupy the variable value *slot* of the same symbol with another function object that does the same or even something different than the function object in the symbol's function value slot.

But for the purpose of this subsection, I guess this could have been helpful.

3.6 A Deeper Look into Lists.

Lists contain data. Everything that can hold data is a data structure. The name 'list' can represent different data structures in different programming languages, because a lot can *look* like a list from outside.

However, a list in Lisp clearly represents the data structure of a *linked list* as mentioned before. The pure name 'linked list' doesn't tell us to much about the

concrete data structure. But it represents a specific concept. In German, the structure is called 'linear *verkettete* Liste' ('lineary *chained* list' and like that the basic building block of a linked list is a chain link.

3.6.1 The basic building block

This single chain link is a pair of *pointers*. That is a technical term we do not have to care about in Lisp. But in this context it is helpful to refer to it in a superficial way.[12] In Lisp, we call this pair of pointers *cons cell* which might abbreviate 'construction cell'. The first part of a cons cell points to the place of a datum in memory. The second part of a cons cell points *either* to the place of another cons cell in memory *or* it points to a technical constant which points to no place in memory. In the programming language C this is called the 'NULL pointer'. In Common Lisp, this constant has the name `nil` which is an acronym for 'not in list'.

At the top of the *Lisp Machine* of the 1970s you could see a stylisation of this concept:

The arrow out of the left part of the cons cell points to the datum. The arrow out of the right part points to another cons cell or to `nil`.

So, let us do it, let us `cons`truct a cons cell:

```
CL-USER> (cons 'datum nil)
```

The function `cons` returns a form, a list in data mode with one element, a representation we are now used to. But we also recognise: This list with one datum is actually a cons cell with its two elements, a pointer to the symbol `datum` and a pointer to `nil`. This pointer to `nil` tells Lisp to set the closing parenthese in the representation returned on the screen, so to speak.

If we do not want to care about the details, we can also build the same `list` with:

```
CL-USER> (list 'datum)
```

There are use cases for `cons` as well as for `list` as well as for further functions for list manipulation.

3.6.2 'Consing' lists.

But we stick to `cons` for a while, because in this subsection we take a deeper look into lists. Thus, while you enter the following line you can imagine what it will return:

```
CL-USER> (cons 'datum1 'datum2 'datum3 nil)
```

[12] Pointers are concrete objects in C, the programming language that is used to implement Lisp. Since in Lisp, we use the convenience of the ready implementation which makes no use of pointers, we do not need to care about its technical details.

– And again, you know how to leave the debugger because, no! This is not the way how it works. With `cons` we construct *one* cons cell and quasi adjust its both pointers. Thus? – We have to do this:

```
CL-USER> (cons 'datum1 (cons 'datum2 (cons 'datum3 nil)))
```

To track this form start in its most inner subform with `'datum3` where you also find the `nil`. That is the second argument to the next cons cell that points to `'datum2`. We can confidently call the most inner subform the cons cell. As well we can call the subform with `'datum2` a cons cell. And you guessed it, also the outermost form with `'datum1` as its first argument is a cons cell. Although this representation of a *linked list* looks pretty nested, we receive a flat list out of this:

```
(datum1 datum2 datum3)
```

This relieves the eyes. And if it is just for this result, the more convenient equally flat function call would of course be:

```
CL-USER> (list 'datum1 'datum2 'datum3)
```

Since we know now that using the function `list` is a convenience for deeply nesting `cons`es (say: *consing*, we can imagine that a list that evaluates to itself in data mode is not that trivial as we might have thought until now, consider:

```
CL-USER> '(datum1 datum2 datum3)
```

The REPL returns the same looking value. It is not the same, these are two lists as two different entities. *But* we can perform the same list operations on them – the do have the same structure. The same applies to `(cons 'datum1 (cons 'datum2 (cons 'datum3 nil)))`. That shows: It happens a lot in the background – very fast and efficient, but a lot. And the source code which finally is compiled to object code looks much more like the chain of `cons`es than like the `list` function.

Before we proceed to some list further operations I want to introduce a special case to you:

```
CL-USER> (cons 'datum1 'datum2)
```

Yes, this is also possible, why not. Of course, the second part of the cons cell can as well point to a datum instead of another cons cell or `nil`. That is a *dotted pair*. You know what? Try this:

```
CL-USER> '(datum1 .  nil)
```

You see? In the regular case (= the second part of a linked list's last cons cell points to `nil`) this is not made explicit. But if it points to a datum this deviation is highlighted. So, this is a *dotted list*:[13]

```
CL-USER> (cons 'datum1 (cons 'datum2 'datum3))
```

A dotted list is the opposite of a *proper list*, that is the chain of cons cells ending in `nil`.

[13] Another very special case is left out here. If you switch off the restraint, you also can point from the last cons cell to the first cell of the list to make it circular. But I have no use for it in this project even if it is interesting to play with.

'Deconsing' lists.

While dealing with linked lists, the basic distinction is that of the `first` *datum* and the `rest` of the list. Try `first`:

```
CL-USER> (first '(datum))
```

Please mind, that actually the `datum` was returned and *not* the first cons cell which 'is' the list of one element. Now try `rest`:

```
CL-USER> (rest '(datum))
```

These are the intended names of these basic list functions. And we will mostly use them. Let us do it again with a list of more cons cells:

```
CL-USER> (first '(datum1 datum2 datum3))
```

and:

```
CL-USER> (rest '(datum1 datum2 datum3))
```

Now the `rest` of the list is *a new* list starting with that cons cell to which the first cons cell of the *old* list points.

The `last` *cons cell* of a list will be returned like this:

```
CL-USER> (last '(datum1 datum2 datum3))
```

From the `first` to the `tenth` cons cell of a list, we can extract *its datum*:

```
CL-USER> (sixth '(datum1 datum2 datum3 datum4 datum5 datum6))
```

This is what Steve Russell intended at the time when he implemented LISP.

Actually at that time he by himself bewailed a lack of inspiration. Because of the architecture of the computer he programmed, he decided for the first part of the cons cell that it would then be known as `car` that is called 'Contents of the Address part of the Register'. The second part of the cons cell would then be known as `cdr` (pronounced 'could-er') that is called 'Contents of the Decrement part of the Register'. *You may safely forget the long names.*

These abbreviations developed their own magic and became more popular than `first` and `rest`. But they mean exactly the same as `car` and `cdr`. Thus, they are the names of the two closely connected pointers *and* they are the names of the functions which return the datum `first` points to as well as the chain of cons cell is returned to which `rest` points to.

The old names may appear more catchy because they can be conveniently combined. If you want to extract the second datum of a list, you can call this chain of functions:

```
CL-USER> (first (rest '(datum1 datum2 datum3)))
```

Again, the *head* is `first` and its argument is the following subform which is evaluated first before its value – the `rest` of the list – can become the input of `first`.

Alternatively you can write this:

```
CL-USER> (car (cdr '(datum1 datum2 datum3)))
```

Of course, the first version is much more lucid. *But* as a third equivalent version try this:

```
CL-USER> (cadr '(datum1 datum2 datum3)))
```

This straight-forward combination '`cadr`' is pronunced 'kae-der' like `cdar` is pronunced 'could-ar' or `caddr` is pronunced 'kae-dee-der' in Lisp slang. These can be handy in certain situations. **But their use is to be justified in this project.** As long as an even cumbersome use of `first`, `rest`, `second` and so on is more appropriate for comprehensibility **it is to prefer!** (This applies, even if I also like `car` and `cdr` on their own very much because they appear somehow symmetric.)

But just for practice not really for practical use, you now have all the necessary informations to define your own custom `eleventh`, `twelfth` and so on functions, if you want to. Also the 'cadeedadeedeldumdeedoo' combinations are finite so you could see how far you would come.

A further important list operation is `reverse`:

```
CL-USER> (reverse '(datum1 datum2 datum3)))
```

This can become helpful for constructing and deconstructing lists when the respective functions are combined and repeated several times in automatic processes under certain conditions.

And now we can appreciate why in Lisp `nil` is known as `nil` that is called `not in list`. We also can appreciate why a proper list ends in `nil`: Because we easily can check the end of a list while automatic processes walk blindly through the chain of cons cells. We do not have to care for the state of the datum to which the last cons cells `rest` points to, like in a dotted list. Additionally, we can appreciate why a circular list might become problematic.

There is even more to say about `nil`. Enter this:

```
CL-USER> '()
```

You see? *The empty list* evaluates to `nil`. Actually, in Common Lisp `nil` *is* the empty list, so `'()` evaluates to itself, which is `nil`. Think about it. It is absolutely justifiable.[14]

Now try this:

```
CL-USER> (if nil 'true 'false)
```

In Common Lisp, everything that is *not* `nil` is true.

```
CL-USER> (if t 'true 'false)
```

You see: `T` and `t` are symbols for just 'true' in Common Lisp. But also:

```
CL-USER> (if 'datum 'true 'false)
```

To avoid misunderstanding: This returns `TRUE` but because the 'true' *branch* contains the symbol `'true`:

[14] The Lisp dialects more strongly inspired by Scheme use more distinctions, also with good reasons.

```
CL-USER> (if (+ 5 6) 19 'false)
```

If anything which is *not* `nil` is `true` than `nil` is not only 'the end of the list' *and* 'the empty list' it also is the *false value*. That does make perfectly sense. It is different in Scheme-like Lisps, these *do* have the boolean values `true` and `false`, also for good reasons. But in Common Lisp we are in a different context.

Only one distinction is important to me in this project related to `nil`: If `nil` shall represent the empty list in our source code, we write `'()`. Thus, we explicitly assign `'()` to a variable and if we walk through lists we test for `'()`. If we assign the false value to a variable or if we test for it, we write `nil`. I do not care if you alternatively write `()` or `'nil` which is equivalent. But I want explicit hints to list operations and to logic operations.

By the way: It can be helpful from time to time to be able to *signal* `nil` from the keyboard to SLIME, i.e. the REPL. The hotkey is `C-u Ret` (`Ret` stands for `Return`). Just try it, even if it appears quite boring, and add it to your notes of the Emacs shortcuts. It is an equivalent to `C-d` that signals `EOF` ('end of file') in a Unix/Linux shell outside of Emacs – `nil` can serve the purpose of EOF, too.

3.6.3 Nesting lists.

Like I mentioned above, lists are not the general data structure for everything. A list is a rigid linkage of pointer pairs we call cons cells of `first` and `rest`, so in a *flat* list we have to follow the whole path from the first cons cell along its `rest` over the next 998 cons cells to finally reach the 1000th to extract the datum its `first` points to:

```
CL-USER> (first (rest (rest (rest (...(rest '(datum1 ...datum2475))
...)))))
```

Its like a numerical order we describe recursively: We cannot state `datum1000` before we stated `datum999` before we stated `datum998` etc. We have no good reasons to do this because there are much better suited data structures for such operations.

But this rigid linkage of lists actually *is* their key feature. They are very well suited for nested data (up to a certain abundance). Linked lists *are* representations of trees and walking through their cons cells is nothing else as walking along the branches of a tree no matter how highly branched it is. The linkage of a nested list becomes a railway with a lot of sets of points that lead us to any turnoff we need to take. And here the different combinations of `car` and `cdr` definitely can be appropriate 'switchpersons'.

Or less metaphorical: The linked list *is* a recursive structure and therefore the way to get from one element to another can most conveniently be described in a recursive way which plays its advantage if you cannot reach your target straightforwardly.[15]

Since Lisp code is also data stored in complex nested lists, the Lisp environment is optimised for parsing and manipulating it efficiently, entire forests. That is the reason why working with linked lists is so tempting in Lisp. It has all the means for doing so. But if we don't deal with data which are organised in a complex tree structure we will not use them. As seen above, working with linked lists can also be very inefficient with huge amounts of data.

[15] The inventor of Lisp, McCarthy, titled his paper on LISP 'Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I' that was published after Russell implemented the idea.

We naturally *will* use them, when we have to deal with Lisp code, of course. But for now, we won't go any deeper. Just to indicate a nested dummy data list:

```
   CL-USER> (list (list 'c (list 'c# (list 'db)) (list 'd (list
'd# (list 'eb))) (list 'e) (list 'f) ...))
```

But you saw already `do` macro forms, `let` special forms, Lambda expressions, a `defun` macro form. You could get an impression there.

3.7 The difference between the values and the effects of forms

I told you that functions return something or more general, if a form is evaluated it returns something, either itself or a different value. Look here:

```
CL-USER> (+ 3 4)
```

This form returns `7`. Two integral numbers `3` and `4` were passed as arguments to the function `+` and in some way these two numbers were reduced to the one resulting integral `7` which was returned by the REPL. That is the *return value*.

Now, try this:

```
CL-USER> (princ "hello")
```

We nearly see double:

```
hello "hello"
```

Why is that? – The upper `hello` is the *effect* of `princ` the lower is its return value. Let us consider it in more detail:

```
CL-USER> (describe 'princ)
```

We are interested in the *documentation string* or *docstring*:

```
   Documentation:
Output an aesthetic but not necessarily READable printed representation
of OBJECT on the specified STREAM.
```

What matters here is that the effect of princ is to make the computer do something somewhere. We will have to deal with the stream concept later for now it is sufficient to know: The function `princ` displays on the screen by default. This is what we wanted. We passed `princ` a string and it caused the effect that this string appeared before our eyes. So why does it return the string additionally? We did not want that? Let us say, this is a consequent behaviour. Often forms have an effect and a return value. Often they only return a value. Sometimes they only have effects. Sometimes we can make use of both, often we don't care about the return value. Sometimes effects bother us and we are only or mainly interested in the return value.

You always recognize the return value of a form in the REPL. It is the value which is highlighted like `NIL` if you enter it at the REPL prompt. As you will realise, if a form outputs something to the screen and it returns one or more value(s), the return

value(s) always appear after the output in the REPL. Like this you get a feeling for the difference. Imagine you program a graphical user interface where you only are interested in the output in a window, still values are returned but the do not appear on screen.

A lot of effects are representations on the screen or read and write access to the harddisk or making the speakers sound or reading inputs from the keyboard or something like that. Much more effects happen at the places in memory where we can store, read and modify data. These effects definitely take place but mostly unnoticable in the moment when they happen.

Let us define an unbound dynamic variable:

```
CL-USER> (defvar *place*)
```

Now let us modify it thus *initialize* it.

```
CL-USER> (setf *place* (list 'I 'am 'data))
```

What you can see at the REPL are the values returned by `defvar` and `setf`. Neither could you recognise how *place* was managed to become a symbol as a name for a variable to be the place in memory to store a value nor anything gave us a hint that the list `'(I am data)` was actually stored at that place.

After the effect occured, we can check the place by calling it:

```
CL-USER> *place*
```

Now the REPL shows us the value to which the symbol *place* is evaluated as a variable name *because* `'(i am data)` is stored at some place in memory which can be accessed in this way.

Now, let us write a little dummy function:

```
CL-USER> (defun dummy (x y) (progn (setf *place* 28) (/ x y)))
```

The REPL again returns the value of `defun`. The effects of the form are combining the symbol `dummy` with the Lambda expression we just entered and assigning the function object to that symbol's function slot. We really want that, but we only realise whether the effect really occured with a function call:

```
CL-USER> (dummy (7 21))
```

The function returns the correct value of `7` divided by `21`. Thus the function modified its arguments properly. But:

```
CL-USER> *place*
```

If somebody else would have written `dummy` and documented it in the way that this function divides its first input by its second we would have had no reason to expect, that also a special variable would be modified by it as well – a place somewhere else in memory which has nothing to do with the function's explicit purpose. And even since we know it, we didn't feel this effect somehow – although the effect was dramatic: We assigned a data list to that place and the function replaced it with a number.

This is a *side-effect* and would be undesired, yes, adverse in this case. But in a more rationally designed function such implicit modifications of places would be perfectly

fine. By the way, `progn` is a special operator. Any subform of that special form is evaluated subsequently but only the values of the last subform are returned (consider its details with `(describe 'progn)`).

So:

- Forms can return values and can have effects, also desired or undesired side effects.
- Additional information: If we are interested in the return value, we use it to pass it on as an argument. The concept of *function composition* is very closely associated with Lisp.

A Lisp form is always evaluated thus in principle it always returns a value. But if the return value is not useful one might decide to suppress it. Here is another dummy example:

```
CL-USER> (defun dummy-2 (str) (princ str) (values))
```

The first thing to notice here is that we do *not* have to nest several subforms in a `progn` special form like we did above. The defun macro provides an implicit `progn`. This is a further effect of it. It surrounds all the subforms after the Lambda list with a `progn` special form to make the proper body of the Lambda expression.

The second thing I want to show is the function `values` as the last function in the body. Called with no arguments it returns no values, so:

```
CL-USER> (dummy-2 "hello")
```

returns nothing but has an effect. The REPL only comments `; No value`. That is no return value.

This distinction between effect and side-effect is not so common in the context of Common Lisp, I think. Frequently programmers seem to say something like this: 'I use this function for its side-effect(s) and that function for its value.' Thus they call everything a side-effect that is different from outputting a return value. But to me the distinction is useful. Therefore I use it in this document.

3.8 Blocks

'Blocks' are control elements of programs. In Common Lisp we can use them explicitly but mostly they are implicitly provided by the macro or special forms we use.

Explicitly we create a `block` special form with the special operator of the same name. The block might be called `nil` or by any other symbol. Take a very simplified look:

```
CL-USER> (block nil (princ "welcome ") (princ "to nil") (return-from
nil))
```

First thing to notice: The subforms after the block name are evaluated in order and the value of last subform is returned just like in the `progn` special form. Second thing to leave the block we generally use `return-from` followed by the block name. In a block `nil` there is a convenience:

```
   CL-USER> (block nil (princ "welcome ") (princ "to nil") (return)
(+ 3 4))
```

We can just use `return`. As you can see here, subforms after `return-from` or `return` are not evaluated. Any macro from wich we can `return` implicitly provides a `block nil`. When we see them in action, we will seize on blocks again.

So far we can remember: Lisp forms return values but we `return-from` blocks.

3.9 Repetition

Here I have to tell a little bit more in advance before we consider some examples:

Programming is a lot about repetition. A machine that performs a task repeats certain steps as long as the machine is running. It is a continuous loop even if not an infinite one because at one point in time it will always be finished, switched off, shut down, be somehow disconnected from its energy source, the source runs out of energy and so on. Put in this way, living and thinking are also continuous loops.

Two terms are associated with repetition in programming *recursion* and *iteration*. Mathematically, *iteration* is a special case of *recursion*. Technically, in terms of programming, the computer does different things if it iterates or if it recurs. That's why it is meaningful to be aware of the differences but also of the things in common.

The latin word 'iteratio' means 'rerun' while 'recursus' means 'recurrence'. If I return to the starting point, I recur. If I do the same thing again I just finished, I rerun. If I return to the starting point for doing the same thing again, I repeat it once. If I do this more than once, I repeat it several times. From this perspective, iteration and recursion conceptually coincide.

Iteration as a special case of recursion means: 'Do something, recur to the starting point and rerun the same thing for x times.' – and I am aware of the number of x. In other words: If I iterate, I know from the start how often I have to repeat what I am doing.

Recursion in general simply means: 'Do something, recur to the starting point and rerun.' But this would be equivalent to: 'Sisyphus, roll this stone up the hill until the end of time and be happy.' So to narrow it down a bit, recursion means: 'Do something, recur to the starting point, check whether your job is finished, if not: rerun.'

This is why iteration is a special case of recursion: We know in advance how many steps we need until our job is finished. That's why we can allocate our resources well and only have to count our reruns while we repeat our work. But while we do so, we recur and rerun.

In general, we do not need to know how long we will have to do the same thing over and over again, as long as we know exactly what tells us to stop. But so, except for iteration, in all other cases of recursion we cannot predict what resources we need for our work because we do not know, how long the job will take. This would be particularly annoying if we actually *could* determine how many repetitions we need, but the necessary information was withheld from us. Then we recur and rerun in general terms while in particular we could iterate much more efficient.

It turns out, that most repeating tasks in programming are iterations. That is not that surprising. Programming is mainly dealing with data. (Most of the informations

on algorithms dealing with data in Common Lisp come from *Vsevolod Domkin*, 'Programming Algorithms in Lisp'.) Very frequently data are stored in structures that are quite straightforward. Iteration is straightforward. There seems to be a correlation.[16]

When we talked about nested lists I already indicated, that recursion and Lisp are something like a dream team. It is very tempting to describe any repetitive task in a recursive way. Lisp supports recursive thinking. If you become used to it, it feels very natural. And in the Scheme-inspired Lisp dialects this is a core feature of the language: If iteration is a special case of recursion we always can describe a solution in a recursive way. Therefore, we need to practice carefully describing the special case in recursive terms. What we do *not* need are different source code constructs for different variants of iterations. These are just *syntactic sugar* – a lot of unnecessary details that obfuscate the learners of a programming language. So, to learn Scheme, you need to practice recognising that a recursive problem belongs to the special case of iteration and therefore describing it as *tail-call recursion*. By that you also practice recursive thinking in general, so it becomes natural to develop the appropriate recursive answer to a problem no matter whether its tail-call recursion or of any other kind.[17]

It also turned out, that a lot of people prefer syntactic sugar. And again, there are good reasons for purity, just as there are good reasons for more detailed approaches that, with the greater detail, also offer more robust, efficient 'example solutions'. Instead of becoming a proficient recursive thinker who can easily create the most efficient description on her or his own with just *one* universal tool, we have a toolbox of means for different use cases.

Common Lisp is sweeter than Scheme. The iterative means are called loops. And these are preferred over tail-call recursion. The standard encourages us to use the loop macros, which also seems to me to represent a sort of consensus among experienced Common Lispers. Anyway, many ANSI Common Lisp implementations provide tail-call optimization.[18] Especially SBCL (the implementation we use with Portacle) is properly tail recursive.

***But*, we will use recursion only when we need it for problems that *indicate* a *general* recursive solution, if an equivalent set of loops would become unnecessarily ugly.**

If we use the loop macros for the much more frequent iterative problems we enjoy the experience of the professionals. They designed the loop constructs in the most efficient way for us. In general, recursion can consume a lot of resorces of the CPU (slows down processes) because a lot of intermediate steps have to be stacked up until

[16] In this very moment I just ask myself for fun: Is an artificial neural network somehow a complex data structure and therefore the 'thought process' in a neural net also somehow potentially recursively traceable? Probably bunk. I will ask my good friend ChatGPT what it has in store about that in its data base.

[17] Well, now I have also to refer to *Harold Abelson, Gerald Jay Sussman and Julie Sussman*, 'Structure and Interpretation of Computer Programs', even if I didn't work through it except the first about 100 pages. But I certainly will have to return to it again. Since I started my journey into Lisp with *Brian Harvey*, 'Computer Science Logo Style' there was no chance for me to avoid the SICP just as there was no chance not to try out Scheme. After taking Emacs Lisp into account for a while I ended up with Common Lisp but also appreciate *Clojure* very much. I always fall between the cracks.

[18] That is the background work for the syntactic purity. Roughly, as far as I think to have understood it: Tail-call optimization cares for the transposition of a tail-call recursion into a loop and this will then be translated to machine code. I guess, the loop is closer to the cumbersome actual processes that happen inside the machine, when it repeats a task several times.

the *stop condition* occurs and may still have to be modified one after another until the result can be returned. If the computer knows in advance how many steps it will make, it can manage the ressources properly before any process starts. Otherwise it just takes it as it comes even if it has to consume all of the resources the operating system places at the disposal. Thus, we do not want to practice tail-call recursion until we can flawslessly describe any iterative solution in recursive terms to become 'Common Lisp Schemers' or 'Common Lisp Clojurians'. We want to use what Common Lisp offers us on its own in the Common Lisp mindset. And we still can distinguish the general recursive problems from the iterative ones if we use loops for the latter. If loops become ugly, that might indicate a recursive problem.

Common Lisp offers us two ways of describing loops. One way is traditional Lisp, the `do` macro you saw above in the subsection about forms in code mode. The form with a lot of parentheses. This is the general, very 'lispy' macro for iteration. You can describe any possible loop with it. We will not deal with it *now*.

There are two simplifications for two narrow use cases of the 'do' macro. Consider this:

```
CL-USER> (dotimes (i 5) (format t " d %" i))
```

This time, I used a different subform for the output to the screen. We will deal with that later. You see, this is the most simple use of an iteration, where the number of steps are known. The range is defined in the list after `dotimes`. The counter variable (here: `i`) is automatically initialized with 0 so we define the number of the steps to take with the *count form* `5`. The macro also provides an implicit `progn` special form around the body, thus any subform after the second subform will be evaluated. We only want to have the actual state of the counter variable written to the screen in each step.

The second simplification is for flat lists:

```
CL-USER> (dolist (i '(0 1 2 3 4)) (format t " d %" i))
```

Here, more is happening in the background. The number of the steps are obvious: It is the number of the elements. But that has to be determined. The output to the screen is the same we saw before. Thus here, in every step, a list element *was* assigned to the variable `i`.

Both simplifications of `do` provide an implicit `progn` and also an implicit `block`:

```
CL-USER> (dolist (i '(1 2 3)) (print i) (when (= i 2) (return)))
```

So: We can exit a general `do` loop and its specialised variants with `(return)`. The macro form `(when (= i 2) (return))` is a simplified conditional form with only the `t`rue branch. It also provides an implicit `progn`, by the way.

The unique Common Lisp way of describing loops is the already announced loop domain specific language (DSL) provided by the `loop` macro. To enter a continuous loop until you signal `nil` from the keyboard:

```
CL-USER> (loop when (read t nil :eof) return 'WHYYYYY???)
```

Do you remember? Did you take a note when I suggested it? You signal `nil` from the keyboard in the REPL with `C-u Ret`.

In this `loop` macro form there are two Lisp forms: `(read t nil :eof)` and `'WHYYYYY???`. The keywords `when` and `return` used in the `loop` macro form are *no* Lisp forms. They are part of the loop DSL. The first Lisp form is responsible for reading `nil` from the keyboard. Using `read` like this is quite low-level.[19] We will deal with it later.

The continuous loop is a special use case of `loop`. (Likewise we can use `do`.) This is no iteration. The `dotimes` loop from above could be rewritten like this:

```
CL-USER> (loop for i from 0 to 4 do (print i))
```

Like I said before: `loop` is a complete language optimised for describing iterations. You can already grasp the style. It can be called a *pseudo code*, an artificial way of speaking that is appropriate for describing algorithms easily understandable for humans.[20]

Everything you can do with `do` you can do with `loop`. So there are a lot of possible loop descriptions. A brilliant overview is the 'Periodic Table of the Loop Macro' on pp. 200–201 in *Conrad Barskis* 'Land of Lisp'. But even that is 'just' an (outstanding) introduction. Also the chapter 'Loop for Black Belts' in *Peter Seibels* 'Practical Common Lisp' does not treat the topic exhaustively. But both Barski and Seibel equip their readers with solid knoeledge.

The `loop` macro uses the implementation dependant internals for efficient iteration thus it arguably is a bit more potent than `do`. Thus: **The `loop` macro *is* to prefer.**

Anyway,..., for some reason the general `do` macro attracts me, so I can't resist using it too.[21]

Now, at the end of this subsection we also take a short glimpse on a possible recursive version of the `dolist` loop from above:

---

[19] Actually, I came across this possibility while I tried to translate some of the very low-level examples in *Brian W. Kernighan and Dennis M. Ritchie*, 'The C Programming Language' to Common Lisp. Even if it is quite rough I think, it also is a good experience. If you want to play around a little ... At least, you learn how to grow deeper into the documentations of Common Lisp and the informations you may request in internet forums of programmers like 'Stack Overflow'. Unfortunately, I never have the time, to really finish most of the books. They become reference books afterwards.

[20] The keyword *for* of the traditional *for loop* was introduced in a precursor language of *ALGOL 58*, originally as the German word '*für*' in the programming language *Superplan* of the German-speaking Swiss mathematician *Heinz Rutishauer* (in 1951!). When I skimmed through a set of teaching hints of some German people who taught Logo to children with the English instruction set, I experienced a surprising view on teaching programming. The instructors realised that the *for loop* was kind of difficult to comprehend. So ad hoc, they had some success when they replaced the keyword *for*. Imagine the *for loop* becomes 'the Gunther loop': `(loop Gunther i from 0 to 4 do (print i))`. – One of Logos positive characteristics is that it's instruction set easily translatable. These instructors didn't think of that – probably because they thought: 'The reference language of any programming language is English, so programming must be learned in English.' Like this for children the keywords of any programming language become arkane 'magic words' if they are not easily translatable. So in certain cases, their sound is associated with the behavior of the computer, which in turn could be replaced by any fantasy word. In fact, I think this shows that the same effect can be seen in adults. If you propose such a stupid alternative, you have not understood the concept of the *for loop* itself. So, the instructors had no sense for an appropriate pseudo code example in mind and could not rephrase a Logo instruction like this: `for [i 0 4 1] [print :i]`. (The fourth element of the first list is the optional step form.)

[21] I suppose it attracts me for the same reason, tail-call recursion attracts Schemers: It is consistently lispy although the *keyword* `do` for a loop was not introduced in Lisp (published in 1960) but in *Fortran* (published in 1957). By the way: I do like the Scheme approach as well. But `loop` is a unique beauty on its own. The language is worth to learn, no question.

```
(defun dolist-recursive (lst)
  (cond ((null lst) nil)
        (t (print (first lst))
           (dolist-recursive (rest lst)))))
```

If you enter this in the REPL then you can call that function just with a list:

```
CL-USER> (dolist-recursive '(0 1 2 3 4))
```

There is another `con`ditional special form that you will learn more about later. But again you can acknowledge: That form also implies `progn` special forms. You see the two subforms of `print` and `dolist-recursive` in subsequent order?

The reason I'm showing you this is not to promote or rail against recursion. Recursion is a tool that is useful in its problem domain. No, I want to show you, that recursion is no magic. Sometimes it is introduced with a certain amount of obscurity, because implementing a recursive function means: *The function calls itself.* This might appear spooky, but it is not. The first function *call* is clearly different from the second function *call* and so on. Because – look at the first *call* of the function (you did it in the REPL after you entered the function). And then look at the last line of the function you entered. You called the function with the list `'(0 1 2 3 4)` as its argument. The function then *tests* the *stop condition*. As long as the list is not empty the 'first' element of the input list is 'print'ed on the screen *and* the function `dolist-recursive` is called again *but* with the `rest` of the list: The second call happens with a reduced copy of the original list – that is another input. This scheme procedes until the stop condition takes effect. And then the 'last call' of the function returns `nil`.

Consequently, recursion means: 'Do not recur *quite exactly* to the starting point' or 'modify your input with any repetition' *and* 'mind the stop condition'. Like this, it is perfectly rational. Anyway, recursive processes are fascinating and stimulate the imagination – especially when dealing with children, of course! But here we can and should deal with recursion in a rather plain way.

By the way: The term 'tail-recursive' alludes to the fact, that the result is reached with the last recursive call and then is only passed through until it returns.

Enter this:

```
CL-USER> (trace dolist-recursive)
```

Now you will experience the evaluation in a new way. Again call:

```
CL-USER> (dolist-recursive '(0 1 2 3 4))
```

Think a little about what you see. To `untrace` the function simply call:

```
CL-USER> (untrace dolist-recursive)
```

I think one reason for the attractivity of recursion that catches the eye is: The call to repetition is in itself part of the problem solving. Thus it is not a distinct entitiy specialised for looping constructions like `do` or `loop` in Lisp, or more specialised in other language 'for' and 'while', maybe also 'repeat ... until' which focus on the stop condition. It is a way of rationally write a function with a focus on the problem to solve, that deals with just one step like in a loop form *and* its recursive call *prepares* the input for the next step. This is a more ... holistic approach.

One reason for loop constructs is, like already mentioned: I only have to focus on one step and on the stop condition and don't need to care about the most appropriate order of the recursive call.

The general `do` macro appears to me somehow as a combination of both, maybe this is why I like it. So: After all, I *do* demonstrate the `do` macro, not as a one-liner but properly edited instead, as you would use it in a source code file. But of course you can also enter it at the REPL prompt. As long the closing parenthese of the whole form isn't typed, pressing `Return` just results in a linebreak:

```
(do ((lst '(0 1 2 3 4) (rest lst))
     (stop-cond nil)
     (useless))
    (stop-cond (princ 'bye)
               (terpri))
  (if (null lst)
      (setf stop-cond t)
      (progn (princ (first lst))
             (terpri)))
  (setf useless 'hidden-wisdom))
```

This example is a bit overloaded, just because I wanted to fill every position of the skeleton and demonstrate the three variants of the variable definition. A proper `do` macro form for this case could be this:

```
(do ((lst '(0 1 2 3 4) (rest lst)))
    ((null lst))
  (progn (princ (first lst))
         (terpri)))
```

The first form after the `do` is the *variable definition* form or simply *varlist*. Like in the `let` special form it contains subforms with a symbol as a variable name as its mandatory first element. That is demonstrated in the third subform of our first version of the `do` loop above with the symbol `useless`. That variable is unbound so far.

The second subform of the varlist contains a variable `stop-cond` initialised with the *init-form* `nil`. Finally, the *first* subform initialises the variable `lst` with `'(0 1 2 3 4)` and defines the *step-form* `(rest lst)` that assigns the new value for the next step.

The second form after the `do`, the *endlist* contains two subforms: The mandatory *end-test-form* and the facultative *result-form(s)*. When the end-test-form evaluates to `T`rue the loop ends. Then any final work *can* be instructed. Like you see in the first version, you can enter more than one result-form. Once again, there is a `progn` special form implied. Here, I use it for `(terpri)` which forces a linebreak.

After these two forms, another `progn` special form is implied that can contain *facultative* statements as the `do` loop *body*.

You *can* comprehend both examples. If at first glance it appears weird, just type it in, stare at it for a while and especially, compare the second version with the recursive version above.

I *thought* a `do` counterpart to the continuous `loop` from above could be this:

```
(do ((eof nil (when (read t nil :eof)
```

```
                    t)))
    (eof))
```

But it exits the loop already by pressing `Ret` not only or at all with `C-u Ret`. ... Well, but as you can see anyway, the step-form just like the init form can actually be of 'any' regular Lisp form. Maybe you want to experiment or have already a clue of the bug and want to correct this `do` form to behave like the said `loop`.[22]

The 'for' loop of other programming languages is:

```
(do ((i 0 (1+ i)))
    ((= i 5) (terpri))
  (print i))
```

(The form `(1+ i)` is a shorthand for `(+ i 1)`.[23])

The 'while' and also 'repeat ... until' loop of other programming languages is:

```
(do ((i 0))
    ((= i 5) (terpri))
  (print i)
  (incf i))
```

(The `incf` form is a shorthand for `inc`reasing `i` with `(setf i (1+ i))`.)

It also could look like this:

```
(let ((i 0))
  (do ()
    ((= i 5) (terpri))
  (print i)
  (incf i)))
```

We can see: A `do` macro form can merely consist of a lot of variable-definition subforms full of init- and stepforms in which all the work is done, and after passing the end-test the result is returned by the result-form. As well, additional steps, especially (side-)effects can be defined as statements. If you only initialise variables or keep them unbound in the variable-definition form, you will leave the 'bookkeeping' to the statements in the body. If you do not fill the varlist at all then the endtest must be dependent of something that is already defined outside the `do` loop as in the last example.

Whether `do`, `loop`, tail-call recursion or any other general recursion *pattern*: You always can push the computer into an 'infinite' loop. Then you made a mistake with the stop condition or the modifications of the data for the next step. **Therefore remember the shortcut for forcing Slime to leave that vicious circle: `C-c C-b`.**

[22] One source of information about the behaviour of a macro is `macroexpand` or `macroexpand-1`. We will come to that later. For now, you just can watch the difference between the `do` and the `loop` in question if you nest each form into this form *in data mode*: `(macroexpand-1 '(do ((eof ...)) (eof)))`.

[23] Its counterpart is `(1- i)` for `(- i 1)`.

3.10 About Scope

Up there, we were talking of defining variables. We also talked about effecting them by manipulating their state, that is changing the data stored in the memory cells which we call the *place* that is referenced by the variable name.

I also told you of the *lexical scope* using the example of the special form of `let`.

Now I want to guide you a bit deeper. For this purpose, I will introduce a few more terms.

3.10.1 Preparatory terms

First: As I told you, the REPL is the Read-Eval-Print-Loop. The process of reading happens first. The process is quite complex and takes time, although significantly less than a second. The part of the Lisp environment that is responsible for this job is the *reader*. And it takes its *read-time*. The process of evaluation is done, you sense it, by the *evaluator*. This parts computes the forms in the version that is passed to it by the reader. SBCL and most of the other Common Lisp implementations *compile* the Lisp code to machine code in this process, so there is a *compiler* behind the evaluator. Thus the time period that is consumed by it is called *compile-time*. The *printer* is responsible for displaying the output that is the result of the *executed* code. The time in which the machine runs the code therefore is called *execution-time* or *run-time*. Even though I may be wrong about the technical details at this point, the idea given for the terms *read-time*, *compile-time* and *run-time* should still suffice and be accurate enough. So we know now that things happen on read-time, other things happen on compile-time and further things happen on run-time.

At read-time, variables and their assignments are made known. Thus, the reader focuses on them in their *textual* context within the source code arranged by the programmers. At run-time, the variables 'come to life' and they are used and manipulated in ways, that are 'natural' to the machine.

Variables used in a textual context are like words of a language that are assigned their *lexical* meaning (their reference of a certain place in memory). Therefore, I call the variable assignment subform of the `let` special form the 'dictionary' that is used in the special form's body.

3.10.2 Lexical Scope

Lexical scope is textual scope of *lexical variables*, and *a lexical variable cannot be seen outside its textual scope*[24]. That means, all the lexical variables in the 'dictionary' of a `let` special form are only visible in the body.

The main purpose of *blocks*, which we have already discussed a little, is to delimit such a lexical scope from the context surrounding it.

If we define a variable with `defparameter`, `defvar` or `defconstant`, we define them in the outmost *top level* lexical scope. So, they are visible in *all* the blocks that we nest in the toplevel for deeper lexical scopes. Since these toplevel variables affect all deeper nested blocks *globally*, this is the aspect of *global* variables.

[24] *Herda*: 'The Common Lisp Condition System p. 5.'

The origin of any lexical scope is the Lambda expression again. Maybe, minding the traditional notation of $\lambda(x).(x + x)$ can be helpful alongside the Lisp form: `(lambda (x) (+ x x))`. The special operator λ *binds* the variable x to the anonymous *function* $(x + x)$. If we call the function with a proper argument, This argument is captured by the λ and applied to replace the variable, inserting it into the named blank space (respectively place):

```
CL-USER> ((lambda (x) (+ x x)) 2)
```

Again remind two things: 1. The subform with the anonymous function is the *head* of the main form. Thus, if you would have defined a *toplevel* function with the same rule and the name `twice` this form would do the same: `(twice 2)`. 2. In Common Lisp, `lambda` is technically a macro that contextually prepares the lambda expression for the reader to be evaluated as the head of a form, as an argument to a higher-order function, or just as a function object.

Now, the Lambda expression in Lisp does not only capture *values* that follow it as arguments. It also *overwrites* the variable *names* that precede it. Thus it opens its own lexical scope within the lexical scope in which it is nested. But even more: If it does not overwrite the variable names from the surrounding environment, it can read the original variables. That means, the same name can be used in the way it is used outside and it can be used as a completely different place for values even of a different type. And since this unique usage of the variable *name* happens within the limits of the anonymous function, the different variable usage outside keeps uneffected. An example:

```
(let ((x 'some-value)
      (y 3))
  (print x)
  (print ((lambda (x) (+ x x y)) 2))
  (print x)
  (values))
```

The `values` form is unnecessary here, I just wanted to suppress the return value of the last call to `print` to be sure, that you see `SOME-VALUE` *two* times.

So far, the lexical scope is observable as it is described. This behaviour of the Lambda expression is called *closing over a variable*, so a Lambda expression is a *closure*. But it is necessary to be aware of the necessary further aspect: Such a closure also can use and *modify* the variable outside of its lexical scope, because through the name it also has write-access to the data:

```
(let ((x 0))
  ((lambda () (incf x)))
  (print x))
```

This pattern – a Lambda expression in a `let` special form – is frequently used and therefore has its own name: *let over lambda*.

It is not very surprising. We could also have increased `x` outside the anonymous function. Here we passed it through to the Lambda expression – we actually *can* pass it through, the read-access. The variable name is *not* in the lambda list of the expression and is therefore effected *outside the body of the anonymous function* – the write-access.

The `print` function takes `x` as an argument *outside* the Lambda expression and therefore prints its state in the surrounding lexical scope.

Now remind yourself that a named function in Lisp is 'just' an anonymous function that is assigned to a symbol as its name (as the compiled function object). And all the necessary implications for conveniently defining:

```
(defun twice (x)
  (+ x x))
```

are managed by the `defun` macro. You also can define an implicit let over lambda like this:

```
(let ((x 2))
  (defun twice ()
    (+ x x)))
```

You must call the function `(twice)` without any argument to simply have `4` returned. So using it like this makes less sense or is quite static. But for example: If you want to have several functions for instance that are effected by some common data stored in a *lexical* variable because you prefer it over a global one, then you might open a lexical scope with `let` around this combination of functions.

Thus: The lexical scope hides variables from access from outside the scope. One function for instance cannot 'see' the variables used in another function. But *within* the lexical scope, all variables defined in the environment of lexical scopes in which the said scope is nested can be accessed.

Also a loop macro implies a block and therefore opens a lexical scope. Remember this:

```
(let ((i 0))
  (do ()
    ((= i 5) (terpri))
  (print i)
  (incf i)))
```

All the variables we can define in the varlist here are only visible within the `do` macro form but `i` is captured from outside.

The value of a lexically scoped structure is that we actually have blocks with lexical variables that have a local effect and they disappear with the block in which they are used. The value of closures is that a connection is possible from the inside of a block to its environment. Thus the data exchange and manipulation can be organised in a structured way. To me this appears to be a continuous inward pull of data from the top level to the inner nestings, through which the subsequent orderly output of the necessary results occurs from the depths. Some kind of organism.

Since in Lisp we can switch between data und code mode so Lisp is homoiconic – code and data share the same structure – this leads to certain consequences which becomes handy for several programming solutions.

3.10.3 Run-time scope

After the reader has read and rephrased the code for the evaluator, the code becomes computed and finally executed and this is, when the variables are used to carry out actions with the data associated with them. The 'variables come to life'.

At this time, it becomes relevant, that variables are not effected by processes which happen in the memory independently from this executed code. When we run *our* program we do not want ... the mail program for instance to access *our* variables for *its own* work. This would mess up a lot. We don't want the opposite way as well.

This example might be a bit oversimplified. It should again at least be sufficient for developing a first notion of the run-time scope: That is the scope opened at run-time in which the variables that are accessible from everywhere in the program are managed.

3.11 'A' Word about Memory Management

As we learned during this crash course: A lot of things are happening in the background which are managed by the Lisp environment. Actually, automatic Memory Management is mainly an invention of the developers of Lisp – simply because they needed it to implement Lisp properly. One tool is called by the beautiful name 'garbage collection'. During the course of a program a lot of side-effects push data here and there in the memory cells and stay where they are even if they are not needed any more. Each function disappears after its work is done, but leaves traces of its work in the places it occupied. Therefore the garbage collector drives through the memory.

We do not have to care about these processes very much. But it is good if you already have heard of the concept of a *stack*. A stack is a data structure which successively is filled element by element. The data on the stack is accessed *last in first out (LIFO)*, that means, the last data `push`ed on the stack is the first to be `pop`ped from it.

Let us have a look. First, let us define a special variable:

```
CL-USER> (defvar *stack* '())
```

A list is a wonderful demonstration of a stack. Now go on:

```
CL-USER> (push 5 *stack*)
```

And continue again:

```
CL-USER> (push 42 *stack*)
```

That is already enough. Now let us `pop` and see, which element we get:

```
CL-USER> (pop *stack*)
```

You can play this game as long as you like. It reminds me a bit of yo-yo.

You traced a recursive function and saw the single steps of manipulating the input list, a function with such a behaviour is also called a *list-eater*. Any immediate step is stored on a stack one after another. And at the 'turning point' of the function, so to speak, the stack is going to be processed in reversed order until it is released again. There exist an error message: '*Stack overflow*'. If this happens, we messed it up. It is no drama. We probably have to restart SLIME. I just mean, that we defined some stuff, we really have to review and revise.

We also can take over control with special means for *unwinding the stack*. Common Lisp has a very matured so-called *Condition System* that lets you modify the control flow according to your needs on a very comprehensive level. We will deal with all of

that. But this also is a constant try and error process. Thus we will see, whether and if how much we will stress the automatic memory management until our application does what we want it to do. Everything on that in this document will be based on *Michał 'phoe' Herda* (2020) 'The Common Lisp Condition System: Beyond Exception Handling with Control Flow Mechanisms'.

3.12 Why dynamic variables are special and not just global?

In the penultimate subsection we thought about the global aspect of top-level variables defined with `defparameter`, `defvar` and `defconstant` again. For this, they can be seen as similar to global variables in the terminology of other programming languages.

The Common Lisp term for `defparameter`s and `defvar`iables is *special variable* because of their aspect of being *also* dynamic variables – in Common Lisp. Those top-level variables are *globally special*.

But *locally* we can also *declare* variables to be special. And that is, we declare them to be *dynamic*.

If we use the above mentioned macros – again – these implicitly declare the global variable to be special. *Explicitly* on a local level, we can declare and use it like this (an example of Herda, p. 11)):

```
(defun bar ()
  (declare (special *y*))
  *y*)

(defun foo ()
  (let ((*y* 42))
    (declare (special *y*))
    (bar)))
```

If you enter both functions `bar` and `foo` one after another, and then call this:

```
CL-USER> (foo)
```

the function `foo` calls `bar`. The function `bar` returns the variable `*y*` which is *locally* declared to by *special*. But it is *not* initialised. It *also* is declared special in `foo` where it also is initialised. Consequently, since `foo` calls `bar` and `bar` outputs `foo`, `42` is returned as the value of the *special* variable both functions can access.

Now, see the difference to a globally special variable. Just enter:

```
CL-USER> *y*
```

You might have already expected it: Our special variable 'died' with its function calls.

So this is another technique than *let over lambda* with a similar effect – well superficially the same: It is different in the background and happens at a different time.

The *symbol* `declare` does not effect the reader and the evaluator. So that supposed list is basically a 'special-special form'. Because of that, it can only be used in specific contexts such as a `defun` or a `let`. And it has its special position. You can see that

above in the two contexts mentioned so far. The symbol `declare` can be used for different purposes. To 'declare' a variable to be special is one of them.

'*Wait a minute!*' you might say now. Didn't you show us, using the example of a side-effect, that we can assign a new value to a special variable and the changed value will remain? Are the global variables not that special? No, they are special in the very same way. I could have demonstrated them at first before the 'special-special form.' But I wanted to make the point with explicitly declaring variables *to be* special.

Consider this. Let us start with an unbound globally special variable:

```
CL-USER> (defvar *var*)
```

Now let us initialize it at the top level:

```
CL-USER> (setf *var* 'toplevel)
```

Now we just use a `let` to make the point:

```
CL-USER> (let ((*var* 'in-let)) *var*)
```

And, you guessed it:

```
CL-USER> *var*
```

It is the same effect. In the example above, when I focussed on side-effects, I did *not* nest the special variable in a `let` binding form. That is the difference.

So what happens? The idea of a stack becomes relevant again. Once a special variable is defined (whether global or local), its name (symbol) is linked to some sort of stack. The stack is empty in the first step of our new example, because the variable is unbound. A first element is pushed on the stack at the top level with the symbol `'toplevel`. In the local `let` environment the special variable *var* is used in *a new context*. The assigned value is pushed on the stack. As long as we stay in this context, this last value applies. When the context disappears, that value is popped from the stack and the first value will be accessed again, as long as it is accessed from toplevel.

Now I can tone down my exaggeration from above: The values of a special variable change dynamically during run-time '"depending on [the environment] a given function was run from"' that used the variable. Insofar special variables are *dynamic* and '*a dynamic variable cannot be seen outside its runtime scope.*' (such as '*a lexical variable cannot be seen outside its textual scope*' which still is relevant on runtime, too)[25]. It is not about the interference of our program and a running mail program for instance, but about the interdependencies within our program, of course. The operating system keeps the processes of different programs seperate from each other.

So this underlying mechanism is what makes variables special in Common Lisp: If they are treated in this way, they are dynamic. 'Global variable' is only the common term for a variable defined at the top level. And in programming language environments that do not deal with dynamic variables that is their only aspect. In Common Lisp, global variables are automatically declared special and this feature is of greater interest since they can be used dynamically.[26]

[25] Herda (2020), p. 5.

[26] Because `defconstant`s are basically write protected global variables, they are not special because they ought to be static during run-time.

3.13 Data types

In Lisp we do not *need* to declare the types of the data we use in our programs. The Lisp environment determines the correct types and does this very well. Anyway, we *can* declare types in Common Lisp and there are two good reasons why we do so:

- If we *can be sure* that the values that are passed to our functions will always be of the data types we want to declare, then this might speed up our program a bit. The automatic needs to do less safety tests.
- *But more importantly:* As soon as we can safely promise that our type declarations are reliable it is very helpful for the readers of the source code to see at once which types of data are assigned to which variables.

The consequence for this project: Even if we cannot give the compiler our promise, that the data passed to our functions will always be of the types we declare, we already declare them *but comment out our declarations.* These declaration comments already inform the readers about our intentions. *The last step of refinement* will always be the activation of the declarations, when enough testing seems to make our promise confidable.

You can already try to comment in the REPL. Actually, you already know how a comment looks like:

```
CL-USER> ; this is a comment.
```

Anything that follows a semicolon in a line goes unnoticed from the reader. Thus you also may comment a form in the same line:

```
CL-USER> (- 2 3) ; this returns -1
```

But for this purpose we use a different form of comment useful to comment out regions which:

```
#|
Everything between these "clamps"
goes also unnoticed from the Reader.
It is only used for commenting out
regions of source code.
Regularly we only use these multiline-comments in
   unpublished source
code for temporarily experimentally remove larger segments
   of code.
If we find out, we don't need it, we erase the commented
   code
segment. If we find out, we do need it, we uncomment it and
    refine it
if necessary.

!!!Commenting out type declarations is the only case !!! in
which the "clamps" are used until the declarations can be
   subscribed
```

```
(erase the comment "clamps").
|#
```

Explicitly, the special form `locally` creates a scope in which declarations for the compiler are possible. The following Lisp forms are surrounded by an implicit `progn`. As a sketch consider:

```
(locally (declare (fixnum x y))
  (\dots))
```

As we already know, the declarations precede all the regular Lisp forms that belong into the body of a form. Implicitly the `locally` environment is part of the `defun` macro form, the `let` special form and many other forms that introduce new variables. As one example, a sketch with `defun`, here between the 'clamps':

```
(defun dummy (str) ; name and Lambda-list
  "A docstring."
  #|
  (declare (string str))
  |#
  (\dots)) ; body
```

Another example with `let`:

```
(let ((foo '()) ; variable list
  #|
  (declare (list foo))
  |#
  (\dots)) ; body
```

We test the type declarations in the REPL. Wrong type declarations might lead to so called *segmentation violations* which might be called like that, or *segmentation fault*, *segfault*, *SIGSEGV*, *signal 11* or *sig11*.

3.13.1 Collection Data Types

We already know a lot about lists. Lists are data collections but we can see that they are not very efficient for linear usage. Therefore below, we discuss more appropriate *collections* at first which posess contiguous areas of special memory cells called *registers*. The reason why the single elements of these collections can be managed much more efficient since the elements are arranged like on a pearl string. The elements are the places in which data are stored. In contrast, the elements of a linked list are pointers which point to places scattered all over the register cells. In the end, we will add two special use cases of lists again to complete the picture.

Arrays

The most important collections for us are *arrays* which we can imagine as whole data fields:

```
CL-USER> (make-array '(3 2) :initial-element 'seed)
```

They look like a nested list but they are not.

With 'data field' I really mean the image of an acre with several furrows. And any furrow contains single grain stalks in a row. The array was arranged with plow and harrow. And the elements of any row are arranged with a seed drill. – Of course we can imagine an array as a matrix as well. But at least for children, this analogy might be catchy and just importing the word `array` into another language makes a magic word out of it like it happens in German continuously. Catchy metaphors may help to make the concept concrete and find proper words in the own language.

Students who have already learned linear algebra in school may also resonate with the notion of a multidimensional row vector as a metaphor for an array. Maybe they would also like the idea of a data field and would also compare the structure of a grainfield with a 'multidimensional row vector'. Because that is the object, we made with the above call: a multidimensional array.

The term *vector* is reserved for a one-dimensional row vector or a one-dimensional array in Common Lisp. So, if we stay in the metaphor, a vector is a single row of an acre – or an acre out of a single row:

```
CL-USER> (make-array :initial-element 'seed)
```

All well and good: Of course, we can also state that a multidimensional array is a nested array and a vector is a flat one just like a list simply a different data structure in the background, completely unsentimental.

However, as you can see in the first example: If you want to create a multidimensional array the first argument to the *constructor function* is a list of two numbers. The first number states the number of dimensions i.e. the numbers of arrays whithin the array. The second number states the number of elements within the subarrays. Thus you always have symmetric multidimensional arrays.

For vectors you can use a constructor function that superficially behaves similar to `list`:

```
CL-USER> (vector 1 2 3)
```

A different data type is that of a *character*, entered in this notation:

```
CL-USER> #\c
```

We can combine characters to vectors of characters:

```
CL-USER> (vector #\s #\t #\r #\i #\n #\g)
```

This vector is *immutable*: We cannot modify that vector itself, we only can produce a modified copy. If we use the constructor of arrays we create a more dynamic collection. Let us take the following example from the Lisp Cookbook.[27]

We start with defining a special variable that is bound to a one-dimensional array – a vector – which is `adjustable`, its elements are `characters` and if we put characters in the vector's places we start at the first position (we start counting with 0):

```
(defvar *char-vec* (make-array 0
                               :element-type 'character
                               :fill-pointer 0
                               :adjustable t))
```

[27] `https://lispcookbook.github.io/cl-cookbook/strings.html`

Now we put characters in the places with a simple loop over a *list* full of characters:

```
(dolist (char '(\#\s \#\t \#\r \#\i \#\n \#\g))
  (vector-push-extend char *char-vec*))
```

Look and be amazed:

```
CL-USER> *char-vec*
```

Probably you guessed it already. The collection data type `string` is a string *of characters*. And as such it is a `vector` of characters. Thus `language=Lisp]{(vector #\s #\t #\r #\i #\n #\g)` and `"string"` are different representation of the same context and all the operations which apply to the first `vector` are also applicable to `string`s. In return this is not possible. There exist functions specialised for `string`s in this very representation. Anyway: *Strings are (special) vectors and therefore arrays.*

Hash Tables

A hash table[28] are special arrays. There elements are twofolded as a pair of a *value* and a *key*. The key can be an arbitrary object which are *associated* with the value. The access mechanism of the hash table is based on a *hash function*[29]

In other languages hash tables may be used to implement *associative arrays*, or 'dictionaries', or 'maps'. In Common Lisp hash tables are created like that:

```
CL-USER> (defvar *hash* (make-hash-table))
```

And then, key value pairs are created like this:

```
CL-USER> (setf (gethash 'foo *hash*) 'quux)
```

Thus, with `gethash` we can create a key in the hashtable like you see here – *if* we use the form as a `setf`able place like shown. The whole form associates the key `'foo` with the value `quux`.

The function `gethash` is used in two ways. On its own, it returns *two* values:

```
CL-USER> (gethash 'foo *hash*)
```

As we can see, the first return value is the value associated with the key we used as first argument to `gethash` – now, and in the `setf` form before. What will be returned with another key we did not use yet? Will it show us that the key does not exist and also no value to associate with? Let us try:

```
CL-USER> (gethash 'bar *hash*)
```

Quite transparent.

```
CL-USER> (setf (gethash 'bar *hash*) nil)
```

Will this also be unequivocally?

[28] In German: 'Streuwerttabelle' (= 'scutter value table') – here I think it is helpful at least for German readers to get a hint for the translation.

[29] 'Streuwertfunktion' (= 'scutter value function').

```
CL-USER> (gethash 'bar *hash*)
```

Of course. Thus the first return value represents the associated value and `nil` represents the absence of a value. The second return value states the presence of a key as *True*, if present, and `nil` as *false* if the key in question does not exist.[30]

As we can see, hash tables appear a bit mysterious because they are opaque unlike regular arrays and lists. But with specialized functions and the general loop macro we can iterate over the table efficiently and reliably so that can use them anyway.

Association Lists (alists) and Property Lists (plists)

But we can make the principle of an associative collection visible. Especially the *association list* is useful as a demonstration.

```
CL-USER> (list '(:key1 .  value1) '(:key2 .  value2) '(:key3
. value3))
```

Because we know now the idea of key-value-pairs it is sufficient if I simply point out here that an alist is a list of nested key value pairs, especially combined as dotted pairs. Like that, any entry of the alist fits into *one* cons cell.

It is also possible to nest proper lists with two cons cells:

```
CL-USER> (list '(:key1 value1) '(:key2 value2) '(:key3 value3))
```

Finally an alist is also sensible which uses a further degree of nesting:

```
CL-USER> (list '(:key1 (value1-1 value-2-2)) '(:key2 (value2-1)))
```

Just:

```
CL-USER> (defvar *alist*)
```

and `setf` the first alist of dotted pairs to that place. (Let us also practice kind of a 'lispy' parlance with the things we learned so far.)

Now we can access the second `:key2` with

```
CL-USER> (assoc :key2 *alist*)
```

This function uses the *keyword* `:key2` to find the whole entry i.e. the sublist with the key value combination. It is the most effective combination of the three variants becaus the key value combination is just *one cons cell*. However, `assoc` works in any case.

The counterpart of `assoc` is `rassoc` ('reverse assoc'):

```
CL-USER> (rassoc 'value3 *alist*)
```

This works only with alists of dotted pairs.

Until now, I confronted you with `keyword`s en passent. But now we can appreciate them more. This notation, `:key` is the frequently used *keyword notation*. There is another mechanism behind the colon which makes the `keyword`s to special symbols. Optically, the notation is helpful to recognize these special symbols as keywords in a context where we can make use of them.

Finally, we consider the *plist*:

```
CL-USER> (list :key1 'value1 :key2 'value2 :key3 'value3)
```

Yes, you're right. A plist is simply a flat list of alternating kay-value-pairs. And do you know, where we use them frequently? We will see soon.

[30] More on this in *Seibel* (2011), Practical Common Lisp, pp. 138 ff.

3.14 Custom Data Types

Of course, also in Common Lisp we can create new data types. The more advanced means of Object Orientation are not of the concerns of this introduction. The will be discussed later while we start with the project.

The more low-level form is `defstruct` to create *structures*. Enter this:

```
(defstruct compositum
  "A dummy for a compound data type."
  first-place
  (second-place 'symbolic)
  (third-place "default" :type string))
```

You see, a `defstruct`ure is a symbol assigned to several places which can also be initialised and be of different types that also can be specified. These places are called *slots*.[31]

Along with `defstruct` goes the automatic creation of a *constructor function* `make-` for instances of that structure (here: `make-compositum`). Also accessor functions for each slot are automatically created (here: e.g. `compositum-first-place`. Finally, a predicate function is created (here: compositum-p) to test whether an object is an *instance* of the respective structure.

Just try it out. Enter this at the REPL prompt:

```
(defparameter *comp1* (make-compositum :first-place 5
                                       :second-place 27
                                       :third-place "never"
                                          )
```

As you see again, we refer to the places with their names preceded by a colon. There – we have alternating key value pairs in a column. If we rearrange them, we have them in a row. If we imagine how `make-compositum` might be defined it could be something like this:

```
(defun make-compositum (:first-place num1
                        :second-place num2
                        :third-place "never")
  (\dots))
```

The Lambda list of the function with keywords is a *plist*.

This looks exactly like we used `make-array`, doesn't it? If you call the variable now, we will see a new data type, which again looks a bit like a list. But like an array, it is not:

```
CL-USER> *comp1*
```

Now, if we want to access the second slot we can do so:

```
CL-USER> (compositum-second-place *comp1*)
```

Suppose, we wouldn't know exactly, that *comp1* is a place for a structure of the type `compositum`. We can check it like this:

[31] In other programming languages slots are called *fields*.

```
CL-USER> (compositum-p *comp1*)
```

This is just a first look. So-called *classes* of the Common Lisp Object System (CLOS) look quite similar to structures. They create additional functions and the counterparts differ a bit. But structures already provide a rough insight. Of course there is much more to the CLOS than that. But we will deal with it later. Even if everything is an object in Common Lisp we are not forced to program in an object oriented way. We can also use parts of this system with structuress instead of classes. But we don't have to. So we can deal with it when we really need it.

Common Lisp is a 'multiparadigmatic' programming language. Personally, I don't think too much of the word. If I think about a paradigm in the linguistic sense or Thomas S. Kuhn's scientific paradigm, that can be shifted in a revolutionary way than a programming language cannot be multiparadigmatic. Only different instances of one paradigma can be applied in one paradigmatic programming language. Unless we think of a programming language as a sentence and any paradigm represents a set of possible constituents at a certain position. But then . . . I would think of four members of the class of object-oriented programming. But procedural programming, modular programming, logic programming, functional programming, reflective programming, generic programming, meta programming– how many members have these classes and the others?

I assume that 'programming paradigm' was originally meant in an exclusive way of a shift to a new paradigm. The programming language Pascal was designed in a way, that you not only was forced to program in an imperative, procedural way. You were forced to program in a structured way whereby I think an essential part of structured programming is the habit of planning and constructing software on the drawing board. Do you were didactically forced to structured programming techniques while you learned Pascal. I felt so relieved when I read Harveys critic about *Dijkstras* structured programming approach. I hated the Turbo Pascal textbook, which was the only one available to me back in the 1980s that forced me to draw structural charts, arrow diagrams and flowcharts of a program I would like to design without knowing what to program before I could start to write code – because there is just *one* right way to programm and that is the structured way! I mean, as an adult, I can understand that the structured path was necessary when trying to develop software using programming languages that descended directly from Fortran/Algol. As an adult I also really liked to draw graphs to visualise complex property law and law of obligations cases. But as a child, this was an obstacle that lasted for a long time. Not the drawing, but imagining something that I couldn't observe, and refining skills of an engineer with a lot of technical background that I couldn't have with 10 or 11 years.

Anyway, in Lisp this approach makes much less sense. But in languages like Common Lisp it makes a lot of sense to use all the possibilities provided to use an appropriate style of organising your sourcecode with which you can solve a problem in a comprehensible way. Therefore I think a term like 'multi-style' would be more appropriate. And since Common Lisp is a multi-style programming language and there are a lot of different problems to solve, we have a good chance of using more than one style in our project.

3.15 Two Custom Data Structures

We know symbols and lists already. Now we can look a bit deeper in their structure.

3.15.1 A Custom List

Do you remember the structure of a *cons cell*? – Exactly, it is an individable pair of pointers. *In this project we* call one pointer `first` and the other pointer `rest`. But we know, that the second given name of `first` is `car` and of `rest` is `cdr`.

If we use these alternatives in our source code, then the respective functions become disputable.[32] **Hence the comment must provide reasons that support the deviation. If the reasons are accepted after discussion, the comment will be confirmed and the use case becomes an idiomatic precedence for the whole project. The respective precedence is to name in the comment. So, we become used to such 'shorthand strategies' and if we see `car` and `cdr` along with a *respective precedence comment* it is easier for anybody to read the code with some expectations.** Consequently, in these cases the precedences become binding. Alternative strategies with `first` and `rest` *for these cases* are then unjustified *in this project*.

Back to the structure. Our custom *cons cell* structure (`c-cons`) simply looks like this:

```
(defstruct c-cons
   "A custom cons cell."
   first
   rest)
```

Now, we already can `c-cons`truct one instance of it:

```
CL-USER> (make-c-cons :first 'first :rest nil)
```

Well, this is not *that* convenient as we are used to. Let us *wrap* a function around this to let it behave like the primitve cons for *abstracting away* some details. (Again: the word 'primitive' does not mean 'underdeveloped' but 'built-in'.)

```
(defun c-cons (first rest)
  "A wrapper around MAKE-C-CONS with FIRST and REST as
     arguments."
  (make-c-cons :first first :rest rest))
```

It is clear what we made here, right? The parameters in the Lambda list are the variables in the call to `make-c-cons`.

Give it a try:

```
CL-USER> (c-cons 'first nil)
```

Better, but the output is kind of raw. For now, we just try a quick, moderately considered approach in a second version of `c-cons`:

[32] No matter how I like the 'character switch' of `car` and `cdr`.

```
(defun c-cons (first rest)
  "A wrapper around MAKE-C-CONS with FIRST and REST as
     arguments."
  (let ((cons (make-c-cons :first first
                           :rest rest)))
    (format t "[~a ~a]" (c-cons-first c-cons)
                        (c-cons-rest c-cons)))
    (values))
```

Now we're talking ...a bit more. But let us customise the automatically created accessor functions a bit more by wrapping them, too and let us handle the `nil` more properly. By the way, don't mind it you read:

```
WARNING: redefining COMMON-LISP-USER::C-CONS in DEFUN
```

It is a perfectly reasonable warning, but not an obstacle. Now, back to further development:

```
(defun c-first (c-cons)
  "Getter function  that returns the FIRST of a custom cons
      cell."
  (c-cons-first c-cons))

(defun c-rest (c-cons)
  "Getter function that returns the REST of a custom cons
     cell."
  (c-cons-rest c-cons))
```

We adjust the `c-cons` function again:

```
(defun c-cons (first rest)
  "A wrapper around MAKE-C-CONS with FIRST and REST as
     arguments."
  (let ((c-cons (make-c-cons :first first
                             :rest rest)))
    (if (null (c-rest c-cons))
        (format t "[~a]" (c-first c-cons))
        (format t "[~a ~a]" (c-first c-cons)
                            (c-rest c-cons)))
    (values)))
```

And now the result looks, like it should. Whether it is sensible to suppress the return value `nil` with `values` is questionable. I wanted to come close to the primitive `cons` which returns the `cons` cell it creates.

We already can test, whether an object is a `c-cons-cell` i.e. an instance of the structure `c-cons`. Let us try it with the automatically created functions first:

```
CL-USER> (c-cons-p (make-c-cons :first 'first :rest nil))
```

This works perfectly. Now, let us try our wraps:

```
CL-USER> (c-cons-p (c-cons 'first nil))
```

Not – . . . – quite. Actually, the output and return value is even misleading. It looks like we just had called `(c-cons 'first nil)` and our function definition of 'c-cons' would not suppress the `nil` that `format` returns.

Thus, if we want to continue the basic idea, we need a new *abstraction* that fits into our scheme. It is a pity: The automatically created `c-cons-p` already has the right name. . . . That is a source for peculiarities based on a hasty approach. But that is not bad, because we already have an approach, anyway. Later refinements may change it from the ground up. But for any most appropriate solution we once needed a starting point.

Temporarily, let us call the abstraction . . . `c-consc-p`:

```
(defun c-consc-p (obj)
  "Tests whether its argument OBJ is a CONS cell. Returns T
      or NIL."
  (cond ((and (stringp obj)
              (string= "[" (subseq obj 0 1))) \dots)))
```

Whooo, this is getting weird. – Well, I *will* confront you already with new functions, macros and so on. But while I was asking myself, which approach might be sensible and tested the behaviour of several functions in the REPL, I realised that we need to step back a bit and focus on our last version of `c-cons` again:

```
(defun c-cons (first rest)
  "A wrapper around MAKE-C-CONS with FIRST and REST as
     arguments."
  (let ((c-cons (make-c-cons :first first
                             :rest rest)))
    (if (null (c-rest c-cons))
        (format t "[~a]" (c-first c-cons))
        (format t "[~a ~a]" (c-first c-cons)
                            (c-rest c-cons)))
    (values)))
```

First of all, try the difference of `format`s output in the REPL:

```
CL-USER> (format t "test")
```

and

```
CL-USER> (format nil "test")
```

If you find a `t` *somewhere, there will be always the option of a* `nil`*, I swear.* Thus the first call *prints* a human friendly `test` to the screen and *returns* the value `NIL`, which is also output by the reader. The difference can be recognised by the different color in the REPL, and the last ouput is the return value. – The second call *returns* the string in its raw state.

The function `format` is very helpful to combine the return values of several functions in a string. Actually it also is some sort of a very specialised DSL. But this complex convenience can sometimes be too tempting.

Consider this alternative instead, we already used somewhere:

```
CL-USER> (princ "test")
```

This appears more useful to me as a new starting point for `c-cons`: We have the fancy output on the screen and the same string as the return value which we can process further. But this won't work so easily. I think, I just `concatenate` a bit. So at first, I 'simply' replace the `format`:

```
(defun c-cons (first rest)
  "A wrapper around MAKE-C-CONS with FIRST and REST as
     arguments."
  (let ((c-cons (make-c-cons :first first
                             :rest rest)))
    (if (null (c-rest c-cons))
        (princ (concatenate 'string (string #\[)
                                    (string (c-first c-cons
                                       ))
                                    (string #\])))
        (princ (concatenate 'string (string #\[)
                                    (string (c-first c-cons
                                       ))
                                    (string (c-rest c-cons)
                                       )
                                    (string #\]))))
    (values)))
```

Can you retrace this? – The function `concatenate` is a *generic function* in the sense of 'one fits many'. If you pass the type of a data *collection* as the first argument, the function 'glues' the individual collections that you pass to it as further arguments together and returns them as *one* collection of the given type.

With the `string` function we also can transform an object into a string. So, the *character* `#[` becomes a string of a single character, its counterpart `#]` as well and between the brackets the return value of `c-first` respectively `c-rest` are '`string`'ed. I use `princ` as announced to have a pretty output *and* the regular `string` returned.

There are two calls to `princ` since these are two branches. The `true` branch outputs and returns in the case of a cons cell with its `rest` pointing to `nil`.

Does it work? – No, we have to suppress the suppression (commenting it out because maybe later we want to use it again and want a reminder):

```
(defun c-cons (first rest)
  "A wrapper around MAKE-C-CONS with FIRST and REST as
     arguments."
  (let ((c-cons (make-c-cons :first first
                             :rest rest)))
    (if (null (c-rest c-cons))
        (princ (concatenate 'string (string #\[)
                                    (string (c-first c-cons
                                       ))
                                    (string #\])))
        (princ (concatenate 'string (string #\[)
                                    (string (c-first c-cons
                                       ))
```

```
                                        (string (c-rest c-cons)
                                          )
                                        (string #\])))))
    ;;(values)))
    ))
```

It still has its *bugs*: It works, but not yet as intended. Just call it with two data as `first` and `rest` and you will see. And also try to combine two c-cons cells.

But so far, we can already think about the predicate. As an intermediate step I think of adding slots to the structure for bookkeeping. What do you think? I know, additional informations blow up the whole structure and your right. But at this moment it might become a helpful solution that supports better ideas. Try:

```
(defstruct c-cons
   "A custom cons cell."
   first
   rest
   representation)
```

Oooh, Lisp doesn't like it, because we changed the number of slots:

```
    WARNING: change in instance length of class C-CONS:
  current length:  2
  new length:  3
  ; Evaluation aborted on #<SIMPLE-ERROR " @<attempt to redefine
  the  S class  S incompatibly with the current definition :@>" 10054BC463>.
```

So, we have to leave the structure `c-cons` behind and define a new one: `c-cons2`. In this detail `defstruct`ures are less flexible as their 'big siblings' `defclass`es. Redefining `defstruct`ures is in itself undefined by the ANSI standard, so a ANSI Common Lisp implementation can just deny an attempt to modify a `deftruct`ure. So, while experimenting like we do it now, we can come in a situation like this. But let's not let that upset us. What has to be, has to be.

But before we actually define a new structure, let us still think about our approach of our `c-cons` function. Maybe it is not so bad if `defstruct` is a bit inflexible. Then we can pause from our euphoric experiments.

What do we want `c-cons` to do? At this point, we want it to return a custom `cons` cell that looks like this, `[first]`, and that the predicate for our structure returns `T` if an instance of `c-cons` is passed to it. What we do have is a function that outputs `[first]` to the screen and returns the string of the output as well.

When does the original `c-cons-p` predicate function returns `T`? What value is passed to it?

```
CL-USER> (c-cons-p (make-c-cons :first 'first :rest nil))
```

We know the return value of `make-c-cons` very well. So it is the structure itself we need to pass to the predicate function ... how surprising.

So if our `c-cons` function does not return the structure we will not be happy with it. As a matter of fact, right now it does return the wrong value. We do have the structure assigned to a lexical variable in our function. Why don't we return that instead?

```
(defun c-cons (first rest)
  "A wrapper around MAKE-C-CONS with FIRST and REST as
     arguments."
  (let ((c-cons (make-c-cons :first first
                             :rest rest)))
    (if (null (c-rest c-cons))
        (princ (concatenate 'string (string #\[)
                                    (string (c-first c-cons
                                       ))
                                    (string #\])))
        (princ (concatenate 'string (string #\[)
                                    (string (c-first c-cons
                                       ))
                                    (string (c-rest c-cons)
                                       )
                                    (string #\]))))
    ;;(values)))
    c-cons))
```

Did we get it? It looks perfectly reasonable that the lexical variable `c-cons` is returned.

```
CL-USER> (c-cons 'first nil)
```

This behaves good. Now have a further try:

```
CL-USER> (c-cons-p (c-cons 'first nil))
```

Success! And we can use the delivered predicate function.

Well – I am just wondering if it would be better to also return `[first]` instead of outputting it. Maybe we should leave it to another *handler* function to bring it to the screen. Intuitively I would say, it is better if we can modify the string on special occasions. But how do we manage to return more than one value? We already know the function for that purpose, look:

```
(defun c-cons (first rest)
  "A wrapper around MAKE-C-CONS with FIRST and REST as
     arguments."
  (let ((c-cons (make-c-cons :first first
                             :rest rest)))
    (values
      (if (null (c-rest c-cons))
          (concatenate 'string (string #\[)
                               (string (c-first c-cons))
                               (string #\]))
          (concatenate 'string (string #\[)
                               (string (c-first c-cons))
                               (string (c-rest c-cons))
                               (string #\])))
      c-cons)))
```

You see the modifications? We use `values` – not for suppressing a return value, no. We nest the two forms that produce our desired values in the `values` form as its first and second argument. Try it. Did you realise that we do not need `princ` anymore?

But wait. What about the predicate function?

```
CL-USER> (c-cons-p (c-cons 'first nil))
```

That's slipping away from us now – again. No. We just need to handle the two return values. So, we need to define our own predicate function `c-consc-p`:

```
(defun c-consc-p (obj)
  "Tests whether its argument OBJ is a CONS cell. Returns T
     or NIL."
  (let ((fn    (first obj))  ;; this is a new strategy:
     parsing
        (first (second obj))
        (rest  (third obj))
  (multiple-value-bind (junk ccons) ;; this is new
      (funcall fn first rest)
      (declare (ignore junk))  ;; this is a new detail
    (c-cons-p ccons)))
```

Well – this is sufficient for introducing three new interesting things. But the function itself is not appropriate. Well, it does what it should, at least if we pass something like `(c-cons 'first nil)` to it, see:

```
CL-USER> (c-consc-p '(c-cons first nil))
```

Did you recognise that the input form of `c-cons` must be in data mode? This is not the typical way of calling a predicate function. And now assume that we pass any other function to it which does not return two values: The form of the macro `multiple-value-bind` is specialised for two return values here. We will end up in a debugger, if we would pass another function that returns a structure or any other single return value.

However, see what we have learned here: In the variable list of the `let` special form we used the `list` functions `first` to `third` for *parsing* the function form of `c-cons` that is the argument in data mode. Such a strategy is especially helpful if we write functions that may take an arbitrary amount of arguments. These are usually collected in an implied list and whithin the function, the arguments must be parsed in such a way.

If we deal with multiple return values the demonstrated way is one to handle this. This macro form is a bit unusual at first glance but soon understood. The first list after `multiple-value-bind` *announces* two variables which represent the first and the second return value of `c-cons`. The second line is the function call of the parsed `c-cons` form itself. The function `funcall` needs a symbol of a function and its arguments seperately – that is the reason of the parsing. So the second line that contains the third subform in the macro form is always the function that returns the announced multiple values. After the function call we can use the `declare` special-special form to `ignore` the first argument. It is a further way to use `declare` and immediately after the function call and before the body is the right position of it. We tell the compiler:

'Don't warn me that I only used one of two variables. I want it that way.' And the last subform is the call to that function which receives the second return value as input respectively the intended return value.

So, we didn't finish the development of a custom cons cell and list. But we explored a bit more and played with the language. Hopefully, it was a bit fun and it gave an impression of one playful approach of developing a solution *bottom-up* starting in an naive way.

Perhaps it also gave a hint, that a data structure is just this: A con*struct* to organize data in a useful way that goes along with a whole set of functions adjusted to the structured collected data. If we speak of data *types* in the end we talk about the names of the data structure we use.

In other programming languages data structures might also be called *composite data types* in general or maybe *non-primitive data types*.[33] A primitive data type in *this* sense is *atomic*. It is a name of *one* datum which represents *one* information, like the `int`egral number `4` or the `char #\f`. If we call these *atomic* we also could call the opposite *complex*.

Since we know what a `defstruct`ure is, we simple could state now: A composite data type is a set of at least two slots to which we can assign values that again are either atomic or complex so we assign values of different types to different slots. If we think of an array, its elements are usually not called 'slots'. But apparently they also can contain data of different types:

```
CL-USER> (vector 4 9.7 23/87 "Hi" #\c '(a (b c)) 'symbol)
```

But we also often use arrays with elements of one data type, for example number arrays, or one-dimensional character arrays which we call `strings`, as we know.

In the sense of 'built-in' we call data structures 'primitive' here that are provided by Common Lisp 'ex factory' whether they are atomic or complex. If we speak of our custom composite data structures, we talk about the entities we create with `defstruct` (or later also `defclass`.

Thus, custom data structures look like the single return value of this call:

```
CL-USER> (make-c-cons :first 'first :rest nil)
```

and provide several functions adjusted to the specific custom case to handle them.

By the way: If we need further functions to parse this representation of the custom structure it can be transformed into a regular `string` just like that:

```
CL-USER> (prin1-to-string (make-c-cons :first 'first :rest nil))
```

That string can be manipulated with all the appropriate `string` functions we will discuss later.

3.15.2 A Custom Symbol

Much more brief we will talk about symbols now. A symbol in Common Lisp consists of five *cells*, but for our custom symbol, we take slots so:

[33] They can also be called *struct*, *record*, *union*, *tuple* or something like that.

```
(defstruct c-symbol
  (name :type string)
  (value nil) ;; no place for values of type 'function
  (function nil :type function)
  (plist nil :type list)           ;; bookkeeping
  (package nil :type symbol))      ;; bookkeeping
```

Since a regular Common Lisp symbol has a seperate value cell and function cell we take over this scheme and split this information in two slots. This is the characteristic of a so-called *Lisp-2*.

In contrast, in Scheme there is just *one* cell for the value and the function object. Consequently, Scheme is called al *Lisp-1*. That means, in Scheme you can `define` a function and a variable with the same *form* and you cannot associate the same symbol name with a variable and a function. In Common Lisp, this is possible.*It is very useful to know whether the Lisp you are using is a Lisp-1 or a Lisp-2. If you define a function in Scheme and later define a variable with the same symbol, your function will be overwritten – or the Scheme implementation will warn that this will happen or will refuse to overwrite the former value. In ANSI Common Lisp you just can use one symbol for a variable value and a function object. Both is perfectly justifiable. You just have to know this feature of the Lisp you are using.*

A `symbol-plist` may safely ignored by us, and also the `package` slot (or cell for the built-in `symbol`) is an administrative information. We can leave to the automatisms of the Lisp environment.

We will play no other game of starting to intuitively develop a function *framework* for our second custom data structure. But now you can imagine already, how you would develop the primitive functions `makunbound` and `fmakunbound` for our `c-symbol`.

First, let us define an unbound special variable:

```
CL-USER> (defvar *foo*)
```

Its variable and function cells are empty. That is what *unbound* means. There is no `nil` assigned to them. Look:

```
CL-USER> *foo*
```

And also:

```
CL-USER> (*foo*)
```

Well, maybe you would anyway assign `nil` as the default value of the both slots. The `nil` in these slots can also be the '`Condition of type UNBOUND-VARIABLE`' or the '`Condition of type UNDEFINED-FUNCTION`'.

Now, let as assign a value to it

```
CL-USER> (setf *foo* 'value
```

and let us define a simple function with it.

```
CL-USER> (defun *foo* (x) (- x x))
```

Now the variable `*foo*` is *bound* and the function `*foo*` is defined. You can check this by yourself.

You can already feel where this is going now. Try:

```
CL-USER> (makunbound '*foo*)
```

(Mind that *foo* must be passed in data mode.) What happens if you call the variable again? And now:

```
CL-USER> (fmakunbound '*foo*)
```

This knowledge can be very helpful in intuitive development.

A Note on Symbolic Programming

Since the very first day, when I accidently opened the PDF of Brian Harvey's book 'Computer Science Logo Style. Volume 1: Symbolic Computing' I am trying to get the meaning of 'symbolic computing', 'symbolic programming' and also of 'computational thinking'. I would now like to take this opportunity to present my thoughts and attempt to find a solution based on them, because having a notion of it helps to deal with the concepts made concrete in Lisp.[34]

If I start in the middle with symbolic programming then it is easy to come across 'numerical programming', arguably the opposite of symbolic programming. That is also called 'scientific programming', because it is used in science a lot. If it would be the opposite of symbolic programming this would mean, the latter is 'unscientific programming'. That is a rather stupid conclusion.[35]

On a still very concrete level, we can state that any programming language that consists of instructions that are named with words or phrases of a natural language is a symbolic programming language. In this point of view, any activity of so called 'high-level' programming is symbolic programming. This is true for programming in any programming language in use today. The first programming language designed for beginners was a FORTRAN descendant and called BASIC as an acronym for 'Beginner's All Purpose Symbolic Instruction Code'. This language failed to be an appropriate beginners language and had nothing to do with Logo. Thus, if Harvey subtitled his Logo textbook with 'symbolic computing' he must have meant something different.

A symbol is something that represents something else. Because we know now what a *Lisp* symbol is – a compound data structure that is used to assign names with values and function objects – we might say, programming in Lisp is closely associated with using its symbols: Therefore, symbolic programming is programming in Lisp. However, this would have the same quality as the statement in the previous paragraph, simply more narrowly defined for a language which uses them as a data type.

[34] Two weeks after writing this subsection I stumbled over a forgotten note, a quora answer of *David Bergman* to the question 'What are the similarities between Lisp and Prolog?' I copied into a text file, from which I learned that the term *symbolic language* denotes programming languages that 'in some sense deal with a Herbrand universe of terms as the semantic domain'. I would have to play with Herbrand structures for a while to get a picture of Herbrand's theorem before I can use that information here on a concrete level. It would be great to know a computer scientist who is a Lisper and could explain Herbrand structures and how to deal with them in Common Lisp. Since I do not have the time for it now, I just keep this passage as it is.

[35] Besides, also programming in a scientific environment and in a commercial one could be distinguished in the first decades of electronic computer systems. In the scientific environment, computers were used for more complicated calculations and programmed mainly with FORTRAN while in a commercial environment COBOL was preferred for tasks of commercial calculations.

But symbols actually are used in other programming languages explicitly as well. Just mind *Prolog*, the origin of high-level first order predicate logic programming, I think.[36] And there, a symbol is something different we can reply. It is an *atom*, so it is not a complex data structure. It represents an immutable name or an immutable object.

But Prolog is *also a homoiconic* programming language. Thus, like in Lisp code we somehow can switch code to data and data to code. And this is said to be the essence of symbolic programming. We can change our perspective on code and data – between a symbolic meta-view and more concrete levels *with* the language. The code itself can reflect our reasonings. Our considerations do not need to be rephrased in a more 'computer friendly' way. This allows abstractions and to handle complex data structures with great comfort. The features of Lisp symbols belong to the means to support this.

Thus, on a more general level symbolic programming is programming with abstractions, solving problems by developing concepts and searching for relations and general structures to find simpler equivalent expressions. Like we could guess already, this needs a lot of work in the background.

If we take symbolic programming as the opposite of numerical programming, we should determine the characteristics of the latter: Conclusively, that would be programming close to the technical details, process and result oriented, focused on the arithmetic operations and discrete relations of numbers.

Thus: The characteristics of numerical programming reflect descriptions of arithmetic and arithmetical thinking, and symbolic programming reflects descriptions of algebra and algebraic thinking. This, in turn, corresponds to symbolic computing, which abstracts mathematics from concrete arithmetic-numerical applications and deals with mathematical relationships, patterns and structures that can be discovered, analysed and used in any domain accessible to the human mind. It also corresponds with the inner processes of the computer, that are ... *computing* stuff.

But that certainly does not mean that algebra and arithmetic oppose each other incompatibly. They are simply different perpectives on the same things. And algebraic thinking also precedes concrete arithmetic case processing. Speaking of background tasks: The algebraic thinking of a C programmer happens in the background of the computer, that is the person's mind. I would say, *structural programming* is at first hand a style of organising all these processes in the mind to develop the algebraic view on the problems to solve and then step down along the structures to the explicit levels that can be phrased in terms of the programming language that *helps to also pay attention to the way the computer actually executes the strategies we have discovered.* Thus, languages like C or Fortran which are concrete in this sense,[37] do not support to communicate your algebraic thinking with its code directly, but your considerations to write the code can also be called 'symbolic computing' in its very sense. On the other hand, programmers using languages like Lisp or Prolog can develop their algebraic reflections and do *not* need a lot of effort to phrase them in the language of their choice. Here the background work of rephrasing the solutions in a computer friendly

[36] The descendent of LISP, IPL was a low-level logic programming language or at least used for the purpose of *doing* logic – propositional logic, I guess.

[37] And C was developed for *the task of writing an operating system – Unix* – thus it *was supposed to* make the inner workings of the computer more lucid.

way, is done by the computer. (However, this automation is the fruit of the explicit work of C programmers.)

Insofar, if the term 'computational thinking' makes sense, it does not mean 'thinking in a way that is appropriate for programming a computer' but 'strategical (and playful) thinking of experienced *human* computers through which they developed their arithmetic capabilities' instead.[38] That is algebraic thinking: Problem solving, choosing an appropriate representation for a problem, modeling the relevant aspects of it, recursive thinking, using abstractions and decomposition, heuristic reasoning, planning, learning, and scheduling in the presence of uncertainty, also the process identifying errors in thinking which is debugging – most of the characteristics *Jeannette M. Wing* stated in her definition of 'computational thinking' which is a summary of Seymour Paperts 'powerful ideas' he layed out in his book 'Mindstorms'.

What does this have to do with Lisp? Lisp also supports numerical programming as it supports symbolic programming. It is said to be inefficient for complex numerical calculations. This is quite irritating since 'numbers' are concrete data types in Lisp.

In human thinking everything is a symbol and the theoretical basic concept of S-expressions also treated everything like that, numbers as well. But why encapsulate the most concrete thing of a digital computer within a complex data structure which then evaluates to itself: `"4"` should be associated with `4`? Especially Common Lisp as the ANSI standard for software engineering of course provides all the means for efficient arithmetical numerical computations. It is very pleasantly positioned between mathematical elegance and roughness. In the last decades, high-efficient numerical computational frameworks were mostly written in C++ if not in Fortran. So, today it would be more economic to use Common Lisp as an interface to the C++ libraries for scientific computing, like Python. But in itself, Common Lisp is much better suited for stepping down to more concrete levels than it is possible with Python. It is quite interesting that since its new ANSI standard of 2017 Modern C++ contains means of abstractions that also are familiar to Common Lisp. Thus, Modern C++ supports ascending to more abstract (algebraic) levels like Common Lisp supports descending to more concrete (numerical) levels even if C++ is not homoiconic and Common Lisp cannot breach through a lower level of abstraction.

All of this does not mean, that 'Common Lispers' are always the most brilliant, mathematical elegant programmers[39] and C/C++ programmers are clumsy plumbers obsessed with detail.[40] On the contrary. As I mentioned above it needs a lot of exercise, practice, self-criticism, humour, discussion, and enjoying the search for an even better solution in any programming language.

But to try your hand at the art of programming makes a lot of fun especially in the algebraic, symbolic style *provided* by Common Lisp. It is also interesting, that

[38] Human computers were the mathematicians who could calculate complex computations in times when digital computers did not exist or were still too slow. Because *they* were called 'computers' the digital calculator machines were called in the same way.

[39] Look at me. At least, I can still serve as a bad example.

[40] I would never dare to speak badly about plumbers, I just remembered a quote of *Alan Perils* about the development of Algol. And brilliant programmers undoubtedly can build cathedrals of code no matter in which language. One of the most brilliant software architects or artists wrote the backend of this timeless editing system I am using, the inventor of TEX, *Donald E. Knuth*. Even a bungler like me can appreciate that by believing the excitement of recipients of his code and the reliability of TEXI'm using with its front-ent LATEX.

this pastime increasingly arouses curiosity about more concrete levels of programming. The transition can easily become: Common Lisp → C++ → C, and back – to Scheme like dialects maybe, over Prolog-like up to Haskell-alike languages.

3.16 Predicate functions and Comparing Things

We already saw predicate functions. In the lazy language of lispy technical jargon we just call them 'predicates'. But actually, they are functions that *test* whether a certain *predicate* applies to an object under consideration. We test for a certain property of an object in a general sense, not only an object of the CLOS. Conventionally, in Common Lisp such a predicate function ends in the suffix `-p`.[41] In reality, however, this convention is not consistently applied. It is always true for the predicates created by the `defstruct` constructor functions, we experienced above.

A proper predicate function is a boolean function that returns `T` if the object in question passed the test and that returns `nil` if it failed.[42] **So, we only add the suffix `-p` to a function if it behaves like a boolean function. And we always add `-p` and not just `p`.**

Why do we care for a hyphen? I want the custom predicates of the project be named consistently. Try some primitive predicates:

```
CL-USER> (evenp 7)
```

or

```
CL-USER> (symbolp 'you)
```

or

```
CL-USER> (functionp (function +))
```

But there are also others with no `p` at the end:

```
(= 7 7.0)
```

or

```
(equal "Du" "du")
```

or

```
(char/= #\a #\b)
```

Here in the last example, we can see how to represent single characters by the way: with the notation of `#\a` for the lower-case character 'a'.

Some further detail:

```
CL-USER> (atom (cons 1 nil))
```

which is equivalent to

[41] The convention in Scheme is the questionmark as final character.

[42] The Common Lisp version of the boolean values *true* and *false*.

```
CL-USER> (not (consp (cons 1 nil)))
```

So you can recognize, every data structure that is *not* a `cons` cell is considered an atom.

There are many more predicates. The point might be clear: A '-p' can indicate, that the function is a predicate and it is a custom product of our project.

Why do I point out that a predicate only is a predicate when it returns `T` or `nil`?

I'll show you another control structure related to `if`:

```
(and (= 100 100) (string= "cat" "cat") (+ 8 3))
```

To realise its functionality consider this case:

```
(and (= 100 100) (< 99 8) (+ 8 3))
```

Thus, if any argument in this form is *not* `nil` then it is evaluated. As soon, as there is a `nil` the evaluation stops and `nil` is returned:

```
(and t t nil t t t (+ 8 3))
```

'So this is a logical junction, a conjunction not a control structure!' – If this would be the case, then it also would return just boolean values. But it returns the value of a form:

```
(let ((a 10))
  (and (> a 9) (* 4 6)))
```

The last argument form is evaluated because the first argument form is true. This is equivalent with

```
(let ((a 10))
  (when (> a 9)
    (* 4 6)))
```

Imagine a function that returns `nil` if it was evaluated without problems and anything that is *not* `nil` if anything went wrong. You also can imagine the opposite case. If we intentionally combine such functions with `and` their evaluation order becomes dependent of the successful evaluation of each subform. Either the evaluation of the `and` form stops with the evaluation of the first successful subform or for the opposite case, the evaluation of `and` stops with the first failing function call.

The opposite of `and` is `or`. It is obvious, how it works:

```
(or (< 100 100) (> 99 8) (- 8 3))
```

or generally

```
(or nil nil nil (- 8 3) nil nil nil)
```

The evaluation stops with the first form that is evaluated to a value different from `nil`.

Speaking of the opposite: If we use `not`, we negate the result of its argument, thus we can yield a boolean value even if something else than `T` is returned. Consider this modified expression first:

```
(or (< 100 100) (< 99 8) (- 8 3))
```

Now the last expression is evaluated to a value different from `nil`. Let us negate it:

```
(not (or (< 100 100) (< 99 8) (- 8 3)))
```

If we would negate it again, `5` would finally become explicitly `T`.

Why all that? Since in Common Lisp `nil` represents *false* and anything which is `not nil` is *true* there are a lot of ways to make use of this behaviour. In the appropriate contexts we use a function as a truth test as well as for further processing its return value, which is not `nil` but also not explicitly `T`, or we want to exploit the side-effects.

So again: **We *only* name functions as predicates that return a boolean value and mark them with the suffix `-p`.** Like this we see at once, when we make a test with nothing but `T` or `nil`. So, if we see a custom function in a context of a test this might then indicate, that our test gives additional actions a piggyback.

So, the naming convention is not about restricting tests or the behaviour of functions. It is about making the human reading process easier by creating reliability and providing assistance to the eyes.

3.17 Further project related predefinitions:

3.17.1 Commenting

The most important comments are the *Documentation Strings (Docstrings)*. We use them everywhere where they can be used and we base our content and style on the docstrings of the functions, macros, variables, methods, classes and so on supplied with Common Lisp.

Moreover, we follow a common comment scheme:

```
;;;; Four semicolons at the top of a source file
;;;; and the beginning of a line.
;;;; Here the purpose of the file will be explained

#|
     Exception: Multiline comments for the licence notices
        right after
     the description of the file's purpose.
|#

;;;  Three semicolons at the beginning of a line seperate
   regions
;;;  of code.

;;   Two semicolons describe regions of code within a
   function or
;;   other top-level forms.

;    One semicolon is for a short note on a single line.
```

[language=Lisp]

3.17.2 Terms related with Debugging

Since I'm not sure if it is clearly defined anywhere with general agreement, but I want to be understood unequivocally:

Bugs

In this document I will explicitly call anything a *bug* that does not prevent the execution of a part of a program, but is the cause of an observable malfunction in some or all use cases of that faulty part. Think of the continuous `do` loop above that I wanted to exit only with `C-u Ret` but that turned out to exit already with `Ret`. This unintended behaviour is a bug – out of my perspective, even if the problem would catch the eye of another programmer easily. It is also a bug, if a function or a series of functions leads into an unintended infinite loop, this is a bug. Maybe we could say, that a bug is a semantic inconsistency? At least we can distinguish intended from unintended behaviour and assume at first hand, that an unintended behaviour is a malfunction unless further analysis suggests that it is a consequence of the intended predominant behaviour.

Inefficient but properly working algorithms are no bugs even if they are intended to execute faster. The intention does not match the capabilities of the solution cobbled together so far. Thus, slowdowns indicate improvements of algorithms and one's level of experience in overcoming simplistic solutions at first hand. But these are still important steps because there *is* already something executable that *can* be improved.

Errors

Thus, if I end up in the *debugger* because of a 'compiler error message', that is an *error* for me, not a bug. These are typos or incorrect syntactic uses of the programming language. As a matter of fact, the 'debugger' is the output of Common Lisps *condition system* which is capable of much more than messaging errors in the defined sense. The condition system can be used as a *real* debugger and for more.

Conditions

Once I have a condition defined, be it for exception handling or for some other context, it remains a *condition* as long as it forces the program to enter the intended state. Therefore, a condition also is a bug if it turns out to have been defined incorrectly, even though the program appeared to be running properly until the unintended behaviour was discovered.

Thus *debugging* is the *process* of identifying the causes for un*intended* behaviour of the executable program and of eliminating them.

Testing

Testing is the systematic, formalized and computer-aided process of challenging functions and sets of functions by letting them run through several cases. Thus, simplified: If this is a function: $\lambda x.mx$ than it is tested automatically by replacing m with 1 first and evaluating the function with $x = \{0, 1, 2, \ldots, n\}$. In a second test run m is changed

to 2 for all values of x and so on. If anything goes wrong in a specific variable assignment, an appropriate message is produced but the test run procedes. (If something would go wrong in this example we might suspect that our multiplication function is buggy.[43])

In so far *testing* is different from *exploring* which we do to get a picture of the behaviour of the program part we focus on. We explore to learn, especially about the behaviour of the components we use to create new components. If we do so while we create the program part we focus on and try to figure out, how we can let it do what we intend, this also appears like 'testing' and it also becomes more and more systematic, but it stays informal. We also have far fewer test cases. Exploring is part of the creative process. Testing is part of the refinement process.

Kludges

About 'cobbling' something together: I like the word 'kludge' (IPA: /kluːdʒ/) I learned from Brian Harvey's Logo textbook. A *kludge* is a workaround like keeping a WW II airplane engine together with chewing-gum and some heavy wire – which might make a professional academic engineer cry. Presumably, this project mainly consists of kludges. A kludge must not necessarily run inefficiently but can become difficult to maintain because of obscurity, complexity or confusion – thus a kludge can be *error-prone* in a way that one can mess it up in a way, that the program will *not* execute someday. It also can cause *bugs*, of course. Therefore: **A kludge is used** 1. **if a part of the program is needed to run somehow at once but needs to be refined later, especially** 2. **if the workaround is needed for an unintended but consequent behaviour as a result of a solution that otherwise first has to keep unchanged.** Kludges recognised as such are highlighted for that purpose. Kludges that are not recognised in the first place may turn out to be such in subsequent debates about them. These will be marked additionally. If the disputants agree that a so-called kludge is no kludge at all, this part becomes unhighlighted.

3.17.3 Metasyntactic variables

If temporarily placeholders for proper variable names are used in intermediate steps, they have to be marked with a comment that contains the date of their introduction. They have to be renamed properly as soon as possible. I allow this selection of metasyntactic variable names in the project:

- needle, haystack (the context should be obvious)
- foo, bar, baz
- qux, quux, quuux, ...
- toto, titi, tata, tutu
- hoge, fuga

[43] This would be an interesting analogy for the discussion of the viability of the own *basic mathematical conceptions*, in the classroom. 'If the multiplication function in my head is somehow buggy, I miscalculate sometimes. I have to try to debug it.'

- hede, hodo
- bla, blubb

3.18 Final Comment of this chapter

I assume that this compact introduction to Lisp was quite rough. I hope, it did not deter you and you can keep the information in mind while you go on reading and it will be helpful for your learning process. You can always go back here, if you have a vague idea but need to reconsider the relevant part of the introduction.

Of course, again: Complete introductions are *Barski*, 'Land of Lisp' and *Seibel*, 'Practical Common Lisp'. Of some value is *David S. Touretzky*, 'Common Lisp: A Gentle Introduction to Symbolic Computation' as well, but only for the completion of some details. The available version is a reprint of the book from 1990, so the 1994 ANSI standard is not reliably covered. The book contains some anachronisms. But especially its chapter 8 about recursion through nested lists with *recursion templates* (or rather *recursion patterns*) for the different use cases is very helpful. Barski (and Touretzki) focus on lists while Seibel puts vectors, arrays and hash tables at the heart of his book. Barski's book is a very funny and deep learning experience. Seibel has also a very readable entertaining style but is much more focussed on professional programmers without overwhelming ambitious laypersons. Both books are 'dead sexy' as 'Xach on #lisp' attributed to Seibel's work.

Chapter 4

The Common Lisp Logo Setup: Design Sketch

Started in May 2024

4.1 Backend

4.2 Command Line REPL

4.3 Setting up a Project with Common Lisp

4.4 Setting up a Test Framework

4.5 Installing Berkeley Logo as a Reference System.

4.6 The characteristics of Logo

Started in May, 2024

Logo is a Lisp dialect developed[1] as a general-purpose symbolic programming language easy to grasp for school children starting in third grade. The developers were closely associated with the researchers of the *Project MAC* (Mathematics and Computation) of the M.I.T. in the fields of artificial intelligence, mathematical logic and development psychology. Therefore Logo was originally developed in the Lisp dialect *MACLisp* which was used at that time in artificial intelligence research at M.I.T. and was the direct predecessor of many following Lisp dialects. That is the reason, why the language components of Logo and Common Lisp are closely related. The same applies to Emacs Lisp.

[1] By *Wallace Feurzeig*, *Seymour Papert*, *Cynthia Solomon*, *Dan Bobrow*, *Richard Grant*, *Frank Frazier* and *Paul Wexelblat* from 1966 onwards.

Logo is optimised for language processing. The origin of the language name is the Greek word $\lambda o \gamma o \zeta$ (logos) from its simple meaning 'word' up to 'the thought'. The project group did research with children in different school classes and also in a learning laboratory of the M.I.T.

The famous *Turtle*, intensively supported by Seymour Papert as the core of his 'Mathland' metaphor for redesigning school education, was an extension that was introduced around 1971. This tool expands the possibilities to experience mathematical ideas and the effects of programming approaches through *Turtle Geometry* in space or isomorphic in the plane.[2] Actually, it started as a robot that reminded someone in the project of a turtle. Papert called it the 'floor Turtle'. This was tested also with pre-school children. Later the Turtle became a graphical interface, the 'light Turtle'. However, it is an important and useful part of the educational programming environment, but it is not essential to Logo. As a matter of fact, the Turtle can be implemented with its instruction set, called *Turtle Talk* by Papert, in any programming language.[3] The tool itself, detached from Logo, is in widespread use until today on the basis of *Scratch* (of the M.I.T.) which targets children again as a graphical-symbolic programming language. A modification of Scratch – which extends the instruction set and purifies the Turtle again – is *Snap!* (of the Berkeley University). Any online Java Script or Python online introductory course let children guide tiny figures on short routes with implementations of Turtle Talk. The Matatalab is a beautiful implementation of the floor Turtle and a purely symbolic Turtle Talk. Because of its nature as a feature of the learning environment, we postpone its description and development to later chapters.

4.6.1 Technical Peculiarities

Started and provisionally completed from 2nd to 4th of May, 2024

The language should enable children to intuitively explore and discover important mathematical concepts and to be as close as possible to human language. Consequently, it supports both numerical and non-numerical, so symbolic operations for solving numerical and non-numerical problems.[4] Originally Mathematical ideas should be discovered through developing programs for language games and experiencing their behaviour with checking the output and debugging.[5]

[2] Generally described by *Harold Abelson* and *Andrea diSessa* in 1981.

[3] Abelson and di Sessa already sketched a way to implement it in UCSD Pascal in the Appendix of their book 'Turtle Geometry', a wide spread programming language in the beginning 1980s which was used for didactic purposes as well.

[4] That's why its a descendent of Lisp. Lisp had all the characteristics needed. But its syntax appeared a bit too strange for unprepared children.

[5] It was this approach which catched me together with the lispy structure of the language. I was searching for a programming language for solving tasks of jurisprudential argumentation theory and thought of modifying Emacs accordingly. I recognised Emacs Lisp (in theory) as a potential candidate with its obvious optimization for text pattern matching when I used Emacs to write with LaTeX. (Also no need for also learnig how to program the user interface.) But the material for learning Emacs Lisp was not that motivating for me. It was then when somehow Harvey's books appeared in my browser and I thought: 'Right, of course, the turtle. We were told about that back then. What is it actually like?' And then I became totally consumed by the approach when I suddenly realized that I am doing

In Logo, data typing is dynamic so the children do not have to care about that. But this is not different from Common Lisp. Logo also has *dynamic scope*. In fact, it has no lexical scope.

> **Thus we have to figure out how we manage that variables in Logo expressions behave like their behaviour is described by Harvey in his third chapter of volume 1 of his textbooks. Probably we will implicitly declare them locally special if they are not global.**

The use of paranthesised S-expressions was apparently considered to be distracting. So from nullary to binary functions parentheses are implicitly set but can be made explicit in Logo. Additionally school math notation was taken into account with binary infix operators as alternative for regular lispy prefix notation (function form).

```
?  3 + 4 = sum 4 3
```

In *Berkeley Logo* the `?` is the Logo REPL prompt. Thus the mathematical operator symbol is *infix* and the verbal symbol of the same function is *prefix*.

> **Apparently, in the background we will have to tell Common Lisp how to parse complex Logo expressions and transform the infix operators with their operands in proper Lisp function forms. Probably, we will *not* transform `3 + 4` to `sum 3 4` before we convert that to `+ 3 4`. But maybe this intermediate step might be reasonable as well. Here must finally happen: `3 + 4 = sum 4 3` → `(= (+ 3 4) (+ 4 3)).`**

This test above for numerical equality will be answered in a strange way, at first glance:

```
You don't say what to do with true
```

This has two implications: First of all, Logo is different from other Lisps in the way that values are not just echoed. They must *always* be passed to another functions. As a consequence, we always need one 'function' that finalises the return values passed through a chain of other functions and performs an action, like printing to the screen. Logo follows a very strict pattern of *function composition*.[6] Thus in Logo, the self-evaluating behaviour of atomic and complex forms which is typical for regular Lisps is interrupted.

---

math while playing with language structures: 'I am doing language math and its *fun*!' (Actually, I fell for the language when Harvey invited to explore Logo right at the beginning: 'Oh my God! It's sexy!' And he already got my full attention before with his preface.) A geometrical approach would not have appealed to *me* at that time and also not before. No doubt, it is marvelous to experience recursion visually especially with its connection to primary school general studies. But it does not seem to be *the* general approach. If I am *not* a singularity, then language patterns can be the icebreaker for those, while for others its geometrical patterns or even arithmetical patterns.

[6] I put the last 'function' in quotation marks since the Logo terminology is different as we will see later.

So, we have to modify the Common Lisp read-evaluation process for Logo expressions. The last function which is responsible for doing something else with an input value than modifying it for passing it on to a next function *always* take *one* argument as input. That argument is either of the data type `word` or a linked list of the datatype `sentence`.

Secondly, as an expressly designed educational programming system, Logo also has a supporting debugging system with messages and diagnostic tools that are intended to be especially helpful for children. Hard interruptions of the debugger would be demotivating. Thus the error messages are quite friendly and helpful. They also contain the value of the last function in the whole expression, which I will intuitively call *the loose end*, from now on.

So here we have to recreate the Logo feedback system with the Common Lisp Condition System.

If we want to see a proper output of the value in the REPL we must tell Logo to `print` it:

```
? print 3 + 4 = sum 4 3
```

From three arguments onwards, even logo cannot avoid parentheses:

```
? print sum 3 4
```

but

```
? print (sum 3 4 5)
```

If you type

```
? print sum 3 4 5
```

instead Logo will prompt

```
You don't say what to do with 5
```

Again: The error message always contains the lose end.

Lists are enclosed by square brackets in Logo and serve as data collections in general, as *input lists* and as *instruction lists*.

This is the first Logo line, Harvey used in his book:

```
? repeat 50 [setcursor list random 75 random 20 type "Hi]
```

It should arguably look like this in Common Lisp:

```
(repeat 50 '(setcursor (list (random 75) (random 20))) '(princ
"Hi"))
```

We will have to recreate the whole Logo instruction set as Common Lisp functions and macros and perhaps also *aliases* of Common Lisp standard components. Our first reference is Harveys 'Berkely Logo Manual'. As we can see, this may also contain functions like `setcursor` that manipulate the screen even if in the REPL. These will have to check, whether they are used within the Common Lisp REPL or in another envirnoment.

The list in the Logo expression above is an instruction list which will be transformed to a Common Lisp form. Its head indicates that it contains code. Anyway, it is in data mode. **We would probably define `repeat` as a macro that expands preferably to `loop`.** We will see how this works in the next section.

As a programming language that is optimised for language processing, Logo has an unique data type: `word`.

Words can be combined in a `sentence`:

```
? print [this is a sentence]
```

A `sentence` is a flat list of words in data mode not an own data type:

```
? print listp [this is a sentence]
```

There is no predicate 'sentencep'. So the function `sentence` combines at least two arguments which are either `word`s or `list`s and returns the `word`s or the *elements* of a `list`:

```
? print sentence "this [[is] [a nested] list]
```

The `word`s are unique because they are no symbols. They are either `number`s:

```
? print word 3
```

or any alphanumeric and pure alphabetic combination:

```
? print word C3PO
```

There exist a shorthand for the function `word`:

```
? print "hello
```

So, we form sentences with

```
? print (sentence "Hello "3 "times "C3PO.)
```

In this case, the result of this necessarily is different:

```
? print [sentence "Hello "3 "times "C3PO.]
```

The second line `print`s a data list on the screen that contains all the list elements quoted *verbatim*.

The equivalent Lisp form is apparently not:

```
CL-USER (print (list 'Hello '3 'times 'C3PO.))
```

Do you realize, that the output of Logo is case sensitive? Let us assume this as the equivalent Lisp form:

```
CL-USER (princ (list "Hello" "3" "times" "C3PO."))
```

(Please notice, that I switched from `print` to `princ`.)

Again, this might imply two things. First of all, `"Hello` or `word Hello` returns a string and not a symbol. Otherwise a capitalised version would be returned. Secondly, in the Logo REPL the we only see the human friendly output of `princ` and the return value is suppressed, the proper `string`. The return value is silently passed either to the next function or to the 'Logo error message system' if it is the loose end.

But this is not everything. Consider this:

```
?  print (word 3) + (word 4)
```

Besides, here we see a case where Logo *cannot* evaluate a complex expression with subexpression that take less than 3 arguments without using parantheses. The operands of infix operator `+` must be distinguishable, at least in Berkeley Logo. Maybe we find a brilliant trick with Common Lisp. But I doubt that at this moment. And maybe this also is quite alright.

> **If we use different combinations of parentheses to enclose the elements of complex expressions in Berkeley Logo, we can use the error messages to reconstruct the evaluation order of the Logo 'functions'.**

Back to our actual topic: The explicit `words` `3` and `4` are summed up just like that. And now take a look on this predicate function call:

```
?  print wordp 3
```

And also on this one:

```
?  print wordp "Hello
```

Numbers are of the same data type as any other alphanumeric combination *quoted* like this: `"C3P0`. But numbers can only sum up with numbers:

```
?  print 3 + (word "C3P0)
```

because + **doesn't like C3P0 as input**.

So we might expect, that the data structure `word` might contain slots like these:

```
(defstruct word
  "The Logo core data type."
  (name :type string)
  (arithmetic-value :type number))
```

11. 5. 2024:
I am glad that I stepped back from writing in this document for a while. I thought there are some wrong conclusions and I found good stuff while experimenting with a custom `word` structure. Meanwhile I really wanted to find some papers of Seymour Papert from between 1966 and 1982. I am a bit annoyed of the habit of pretending, that alway the newest is the most actual and original approach. There is no such thing as permanent pioneering. That is just dishonest marketing, lazyness and ignorance. As a matter of fact, everything I encounter as educational tools used with the computer is basically the same stuff from decades ago, just with a different appearance. And I think it is desperately necessary to compile a reader of the research findings of the people around the Logo movement back then and their pedagogical experiences just to define the place from where we already could go further. However: **I found what I was longing for already a while ago and what I didn't expect to find and therefore have apparently overlooked: I found the 'Logo Memo II' from 1975, the description how Logo was implemented with Maclisp.** Until now I only found the 'Logo Manual' from 1974. But it must have been in the same list. Probably I was too focused or maybe my mind wanted to think about it independently. However, I am very happy! I already saw, that a lot of my assumptions are quite precise even if in detail below I considered some nonsense. But still, this makes me a confident reader of the document now, which I type out now with LaTeXand therefore I read it very carefully. I am very curious how much I can just adopt straight away and what I want to be adjusted to ANSI Common Lisp. After that I will rephrase this document according to my new insights. It feels, like I would be part of the discussion back then and am able to participate confidently ... as well as participate in the narrower `Logo` discussion as in the broader `Lisp` discourse. A beautiful feeling.

(Right now I saw, since I opened the wrong PDF for continuing to type it in LaTeX, the one with the Logo Manual from 1974 instead of the Logo Memo II, that we will consider that as well, later. For now I'm still sticking with Harvey's material and of course this will stay the main reference. However, there are a few details described that I would like our Logo setup to inherit – except they are also covered by the Memo II.)

11. 5. 2024: This is the rest I wrote between 2nd and 4th of May which I will rephrase later. Consider also this:

```
?  print (word "C "3 "P0)
```

We can use `word` to concatenate several words.

So: We will have to handle the data structure `word` so that its instances behave in the intended way. Actually there is *no* data type `symbol` in Logo.

A further consequence:

Most likely, the raw form of the Logo expressions in our Logo setup consists of `Strings`, so we need to deal with the appropriate `read` functions their analysis relies heavily on `String` functions.

At this moment, I imagine the raw form of `print wordp "hello` to be like this:

```
"print wordp Ḧello"
```

This is the output of `read-line` in Common Lisp. Just enter it in SLIME, then type the expression and press `Ret`:

```
CL-USER> (read-line)
```

I would like to call Logo expressions as inline code in a `logo` macro:

```
CL-USER> (logo print wordp "Hello)
```

This macro would then manage all the background work to rephrase the Logo expression into:

```
(princ (word-p (make-word :name "Hello"))).
```

and pass it to the evaluator.

In Logo, function definitions look like this (already embedded in the desired `logo` macro (any of the following examples are taken from Harveys textbook):

```
{(logo
   to primer :name
     print (sentence first :name [is for] word :name ".)
     print (sentence "Run, word :name ", "run.)
     print (sentence "See :name "run.)
   end)
```

Again, the pedagogical intentions shine through: Children add new verbs to the Logo language so they *teach the computer* new words. Active verbs are the absolute appropriate choice, because instructions make the computer act. Therefore the keyword `to` introduces the function definition and the keyword `end` closes the bracket. The tabs are not necessary. Traditionally, Logo source code is written without tabs. But there is no need for further justification that this stylistic element is helpful, so of course, it will be a style convention to use tabs.

The constituents should be clear. The Lambda list of the Logo 'function' contains all the variables in the function head that look like Common Lisp keywords. This Logo definition ought to be rephrased and passed to the Common Lisp reader in this form:

```
(defun primer (name)
  (princ (sentence (logo-first name)
                   (sentence (word is) (word for))
                   (word name) (word .)))
  (princ (sentence (word Run,)
                   (word name)
                   (word ,)
                   (word run.)))
  (princ (sentence (word See)
                   name
                   (word rund.))))
```

Of course, the Logo components must be defined.

I do not just want this blank space convention (which will be auomatically supported by the editor as ususal), I also want to introduce documentation strings, so this is also possible:

```
(logo"
  to talk
    description [A procedure that asks for a person's full name and
       parses it.]
    local \"name
    print [Please type your full name.]
    make \"name readlist
    print sentence [Your first name is] first :name
    if (count :name) > 2 ~
       [print sentence [Your middle name is] first bf :name]
    print sentence [Your last name is] last :name
  end
")
```

I don't know, whether the use of the character ‘ ’ is common to all Logo implementations. It indicates that this and the next line belong together. I like it a lot because it makes parsing easier. In this example it connects the first line of the `if` expression with the ‘true branch’ in the second line.

There are two rephrasing strategies conceivable. If we take the Logo setup as an extension of Common Lisp it is natural to add lexical scoping to Logo, it would look something like this:

```
(defun talk ()
  "A procedure that asks for a person's full name and
     parses it."
  (let ((name))
    (princ "Please type your full name.")
    (setf name (readlist))
    (logo-print (sentence "Your first name is "
                          (logo-first name)))
    (when (> (logo-count name) 2)
      (logo-print (sentence "Your middle name is "
                    (logo-first (butfirst name)))))
    (logo-print (sentence "Your last name is: "
                          (logo-last name)))))
```

I am not sure, whether it also was a pedagogical decision to keep variables to be special and intentionally leave out lexical scope. Lexical scoping is not a Lisp invention. It is a concept introduced with ALGOL. ALGOL was a joint venture of many early computer scientists as representatives of very concrete up to abstract approaches, but to refine *Fortran* in the end of the 1950s. Lexical scope was integrated in Lisp with the development of Scheme in the 1970s. Common Lisp adopted this inspiration in the 1980s. Logo did not. I think it would be erroneous to assume that the Logo language concept somehow was seen as sacrosanct, completely matured for all times. A lot of different Logo implementations were developed, also integrating object-orientation, the

hot stuff of the 1980s.

I think, I read somewhere, that supporting only dynamic scoping was understood as a helpful simplification for child programmers. This would be the main reason to rephrase Logo expressions in our setup in a way, that any local variable *will be declared special*, if possible. The second reason would be, that Harveys textbook relys on this property of the Logo language – not only his, any of the brilliant Logo teacdhing materials of the 1980s and beginning 1990s. The examples would not work with our setup when this behavior is used. But because the authors of these materials were the only ones that devoted their research interest to using programming as a core mean of school education in learning labs and in the field already since 20 years in the 1980s, these materials still are the best there is to design learning environments for developing computational thinking, which we might comprehend as a superset of algebraic thinking insofar it also entails concrete programming *language* competences. It should also not be forgotten, that Logo is the origin of *all* following educative approaches in computer science. 'Lego Mindstorms' already references Paperts influential book 'Mindstorms: Children, Computers and Powerful Ideas'. The origin is 'Logo Blocks', the beginning of visual block programming which was commercialised as 'LEGO/Logo'.[7] But in parallel, Logo was and is used until today in teaching and also in biology and sociology research ('Netlogo'). So, the Logo teaching expert knowledge in the USA and Canada is already 58 years old. Through Logo we can make this invaluable potential directly usable for current school education everywhere instead of arrogantly claiming that we are pioneers who are brilliantly reinventing the wheel.

I would therefore prefer to add something like this to the rephrasing, if the special-special form `describe` can be added more or less like this by a macro:

```
(defun talk ()
  "A procedure that asks for a person's full name and
     parses it."
  (let ((name))
    (declare (special name))
    ;;----------------------
    (princ "Please type your full name.")
    (setf name (readlist))
    (logo-print (sentence "Your first name is "
                          (logo-first name)))
    (when (> (logo-count name) 2)
      (logo-print (sentence "Your middle name is "
                    (logo-first (butfirst name)))))
    (logo-print (sentence "Your last name is: "
                          (logo-last name)))))
```

(A funny punch line of this short, somewhat grim 'attitude essay', directed towards some voices in educational research, especially in Germany.)

After a bit of contemplation, I think, we should do both. Maybe following the 'Lisp-1/Lisp-2 distinction' we could use a 'Logo-1/Logo-2 switch'. Yes, I like that. It

[7] Actually, I suspect that also the standards of the US American National Council of Teachers of Mathematics with their emphasis of algebraic thinking and problem solving is somehow or immediately influenced by that movement.

is of pedagogical value to focus on one concept at a time before we add a second. The learners begin with the concept of *dynamic scope* in *Logo-1*. And as they internalise this concept, they may encounter situations where they wish that this one function couldn't use the local variable of that other function. And then, they can explore the concept of *lexical scope* with the switch to *Logo-2*. In this way, a reflected conceptual understanding of dynamic and lexical variables can develop and dynamic ones will also be recognized as special variables. At least, this is the idea.

> **So, technically we will have *two* macros: `logo-1` and `logo-2`, that mark the difference. The first *adds* `(declare (special ...))` and the second does not. (I hope this can be realised. Somehow, I'm slowly starting to get the feeling that we're going to have to do everything by hand and not just expand the `logo` macros into a `defun`.)**

We could see already, that Logo – as a special Lisp dialect – is also homoiconic. And of course, it also provides anonymous functions.[8]

Anonymous functions in Logo are called *expression templates* in an *explicit-slot form* or *question mark form* and I also remember that I needed a while to grasp them, so now I have to start again, since it was a long time ago. Try this in the Logo REPL:

```
? show map [? * ?] [2 3 4 5]
```

This is a very compact notation that is context-dependend and is mainly used as a tool for iteration in Logo, in form of an instruction list or expression list[9] passed to a function that takes functions as arguments,`Higher-order functions`, in this case `map` that returns: `[4 9 16 25]`.

Oh right! (I slap my forehead with the palm of my hand.) So this *is* a *lambda expression*. It's just its *body*. So, the variable *binding* is implicit: $\lambda(?).[? * ?]$, or *(lambda (?)* `[product ? ?]` *)* which is equivalent. The children just use the anonymous function without calling it by this strange name without needing to have a concept of a (mathematical) function at all for appropriately comprehend this expression. They do not have to care about the idea of a *binding form* and certainly not of *closures* which both have nothing to do with their actual experience realm. *But*, the *template* makes it easy to connect to the more general concepts when the learners experienced and integrated *templates* and *functions* in their realm. Then they are able to step aside on the metalevel.

The Common Lisp equivalent to Logo `map` is `mapcar`. So:

```
(logo-1 show map [? * ?] [2 3 4 5])
```

would be rephrased to:

```
(mapcar #'(lambda (x) (* x x)) '(2 3 4 5))
```

The same applies to:

[8] In my perception, the Logo notation is a bit odd, but this may be a question of habit. I remember thinking it was pretty fancy back then, when I was exposed to the idea of anonymous functions with them for the first time.

[9] The distinction becomes comprehensible in the next subsection.

```
(logo-1 show map [(word ?1 ?2 ?1)] [a b c] [d e f])
```

The expression returns `[ada beb cfc]`. This is especially a strength of Logo: Conveniently concatenating words and sentences.

As an extension of Common Lisp we could also think of our Logo setup as a DSL. It then is a DSL for the domain of developing computational thinking and by that it also happens to be a DSL for the domain of symbolic language processing. Because Logo is a complete general-purpose language, we can as well think of our Logo setup as a further abstraction layer of Common Lisp. Thus the transition can easily become: Logo → Common Lisp → C++ → C.[10]

Back to topic: This Logo template is said to be in *named-procedure form* because its first element, its *head* is the name of a function. But anyway, it is the body of a Lambda expression as well. Since we can distinguish two *slots* in this *template* (in Logo parlance) it apparently rephrases to:

```
(mapcar #'(lambda (x y) (word x y x)) (sentence (word a)
                                                (word b)
                                                (word c))
                                      (sentence (word d)
                                                (word e)
                                                (word f)))
```

Actually there is no difference between an 'explicit-slot' or 'question mark' form and a 'named-procedure' form. The reason for the superficial difference is the difference between infix and prefix notation which are both applicable in Logo: Again, `[? * ?]` and `[product ? ?]` are equivalent. And again we can see the unique 'bracket issues' of terms notated in the infix.

Now I understand, why the Logo Lambda expression appeared a bit odd to me, when I met it again after years. The lambda keyword and the Lambda list helps me to keep track of the informations in the body. But, sure, that is a question of habit. However, I would say, it already indicates a little idiosyncratic aspect of the decision to use both kind of notation.

But at the concrete application level it makes sense to distinguish between these different manifestations of Logo templates.

So, this example:[11]

```
(logo-1
  to hangword :secret :guessed
    description [Compares two textual inputs. If they are
       equal, it
returns the first input if not it returns "-.]
    output map [ifelse memberp ? :guessed [?] ["-]] :secret
  end)
```

basically 'rephrases' to

[10] If we will start the parallel project in Clojure, which I see as an intermediary between Scheme and Common Lisp, it is also for this possible transition: Logo → Clojure → Java, such as Logo → Clojure → Clojure Script → Java Script – apart from some features of Scheme which are not part of the Common Lisp standard.

[11] Which I first came across yesterday, when searching for the anonymous function in Harvey's textbook – the above examples are from his Berkeley Logo manual.

```
(defun (secret guessed)
  "Compares two textual inputs. If they are equal, it
     returns the
first input if not it returns "-."
  (declare (special secret))
  (declare (special guessed))
  (mapcar #'(lambda (x) (if (member x guessed)
                            (sentence x)
                            (sentence (word -))))
                        secret))
```

It confused me a little at first.

> **If a list in Logo contains question marks in a conspicuous place, it is the function body of a lambda expression and we can rephrase it accordingly. Consequently, if a Logo higher-order function receives a Logo *sentence* it is the task of that function to process the sentence into a proper Lambda expression.**

As a reminder: Perhaps, the `defstruct` of `word` is a more complicated cons cell of the kind:

```
(defstruct word
  "The Logo core data type."
  first-string
  second-value
  third-representation
  rest
```

because the word and sentence operations indicate more than just string operations. They appear as cons cells and linked lists.

As a final note for now:

> **For one-liners at the REPL we will still use the `logo` macro as a shorthand for `logo-1`. So it feels like writing a Logo expression in the Logo REPL while we enter it in the Common Lisp REPL like this:**
> `CL-USER> (logo print word "hello)`

4.6.2 Terminology used for and with Logo

Started on 4th of May, 2024

I adopt the terminology, Brian Harvey uses in his textbooks and in the Berkeley Logo manual. Because Harvey supported the textbook of Abelson/Sussman/Sussman 'Structure and Interpretation of Computer Programs' (SICP or called by its nickname 'the Wizard Book') very much, perhaps some overlaps are due to this. But these overlaps also exist with Algol. And Harvey wrote a further complete textbook ('Simply Scheme')

as a 'Logoish' bridge to SICP in which he could focus on the SICP terminology.[12] The programming language used in SICP is Scheme, which was invented by Sussman and *Guy L. Steele*, the author of the first draft of the language standard of Common Lisp. The common background of all of these persons is not just Lisp but the orbit of the M.I.T. and its Project MAC. Insofar, common terms are very much expectable.

Anyway, the terminology used by Harvey has a Logo specific character and a general one. In the end, it is of no importance for our project how pure or general the terminology is. It is perfectly usable and so it makes a lot of sense to use it for our Logo setup. Additionally, it is absolutely harmless to use its distinctions as well in Common Lisp.

Instruction A Logo *instruction* contains exactly as much informations needed to tell Logo, what you want it to do. 'Read chapter 2 of this book.' is an instruction. 'Read' alone is none. Consequently, basically a Common Lisp function form in code mode is an instruction. The Common Lisp function *symbol* alone is none.

Procedure A Logo *procedure* is similar to a *recipe* or any other method to perform a specific task. A recipe is a set of instructions that are carried out in sequential steps. Just like a recipe with the name 'cheese cake' contains the instruction 'preheat the oven to 160 °Celsius' a procedure with the name `talk` contains the instruction 'print [Please type your full name.]' – Of course, this instruction itself names a procedure that contains instructions with the effect that `Please type your full name.` will be output on screen. But that is concealed behind the procedure name. Here we might see one good reason why a notation of a programming language in prefix form can be perceived as pleasent: The imperative form at least in many if not in all of the European natural languages is a sentence in verb-initial word order. For this reason, it is a sound idea to define Logo procedures like a dictionary entry for a verb in infinitive form e.g. `to talk ...`. So, a Common Lisp *function* (in general) is a procedure from the Logo perspective.

Primitive (Procedure) A Logo primitive (procedure) is a procedure that is written in the programming language used to implement Logo and compiled. Thus it is a procedure that exists as running machine code in the Logo environment and is not a plain Logo function written in Logo. So, a Common Lips primitive (function) corresponds with a Logo primitive (procedure).

Library Procedure A Logo library procedure is also delivered with the basic Logo system but it is written in Logo. In Common Lisp, especially its standard macros are the delivered entities written in Lisp. Since out Logo setup will be a custom Common Lisp system of Common Lisp libraries its Common Lisp functions and therefore *all* Logo procedures will be library functions/procedures.

Super- and Subprocedure The *relative* terms *super-* and *subprocedure* denote the relation of two procedures. The procedure that calls another procedure with one of its

[12] As a matter of fact, my approach of extending Common Lisp with Logo is Harvey's approach transferred from Scheme to Common Lisp that subconsciously took root in me, as I realise now.

instructions is the superprocedure to the other subprocedute. Thus, *both* are defined at *top-level* but together the build a context because one procedure calls the other and therefor the first procedure is 'Super-' and the second is 'Sub-'. Even if they are both at top-level, content-related the relative superprocedure of another subprocedure is a *higher-level* procedure relative to its *lower-level* subprocedure. In general, we also speak of a super-/subordination in Common Lisp, mostly in terms of high-level and low-level functions. Specifically, the notion of 'super-' and 'sub-' is closer associated with object-orientation in Common Lisp.

Functions A Logo procedure *represents* a *function* in the mathematical sense, if the procedure exclusively maps its input arguments out of a set of possible arguments (the function's *domain*) to a set of possible return arguments (the function's *range*). Thus there are no functions in Logo, only procedures that represents functions. This strict point of view was introduced with Algol which is why in Pascal and its descendents as the languages which are closest to Algol the two keywords `procedure` and `function` are used to distinguish those procedures that do not represent a function and those that do. Technically we can state: Procedures with the sole effect of manipulating exactly the data that has been passed to them and determining and returning a new datum or several data from these input data, are called 'functions'. Because of this, the narrower sense of 'side-effect' in programming is handling variables which were not defined within the procedure that is carried out. If we *use* a procedure, we only know its name and the arguments it takes. So we can assume, that something happens to this arguments which yields a return value on the basis of the input values. This effect is reliable and conceivable. If something else happens additionally, we have no informations to foresee what happens in the black box we are using. That's why its a side-effect. So, also a Common Lisp `function` keeps a procedure in general and only *represents* a function, if it has no side-effects. But the careless use of language which is common to most of the programming languages is also typical for Common Lisp. So we go on calling Common Lisp procedures `function`s but be aware that they are functions in a broader sense.

Output and Input The return value of a Logo procedure is called `output`. And this `output` *has to* become the *input* of another procedure i.e. the or one of the procedure's argument(s). Thus any procedure with an output or several outputs must be *composed* with another procedure.

Operation A procedure that `output`s one or more values, regardless of whether there are also side-effects.

Command A procedure with an *effect* like printing on screen, writing to a storage medium, reading from keyboard and so on is a *command*. It does something with its input values that happens outside of the procedure *and* outside of the register memory. So, these effects are different from the effect of an operation (issuing a return value, regardless of whether that value is based on multiple side effects). When executing a command, many locations in the register memory that cannot be traced from the outside must be manipulated until the word `hello` appears on the printer paper. So achieving an effect has many side-effects. This is the reason, why I distinguish the

output from the effect and the side-effects of a 'function' even in Common Lisp. I keep in mind these Logo concepts of *operation* and *command*, which were taught to me from the beginning of my journey in 'the Land of Lisp'.

Complete Instruction The chain of Logo operations always ends in *one* Logo command that takes the output of the operation directly connected to it. So, a *complete statement* in Logo *always* consists of a command at the beginning, followed by as many *expressions* as the command requires input values.

Expression A Logo expression is either a proper instruction with an operation and its complete arguments, like `(sum 2 3 4)` or `product (2 / 3) (2 + 5)`. Or it is another compound or atomic expression like a `sentence`, `[a beauty of a sentence]` or a `word`, `"me`.

Sequential Programming Paradigm The approach of writing and organising the procedures of a program in recipe-like sequences of steps carried out one after another to yield the intended result. It is characterised by *iterative strategies*: An instruction becomes an expression of a looping instruction like `for` and be repeated for several times. Sequential programming provides one intuitive approach to programming and is supported by Logo. It is typical for beginning Logo learners while they explore the language and its environment. And it is closest to the inner workings of computers which happen sequentially. In Common Lisp any explicit or implicit `progn` special form provides sequential programming and of course the `do` and the `loop` macro. We already know, that Common Lisp prefers iteration over recursion even if we certainly can use recursion in any pattern.

Functional Programming Paradigm *Functional programming* is the approach of writing the procedures of a program primarily as representatives of functions without side effects in the sense of rules for mapping elements of a domain to elements of a range, which are composed in such a way that they pass their output value as input to another procedure. This approach especially supports recursion since recursive calls of a procedure is composing the procedure with itself (with modified output that becomes the input of the next call of that procedure). The composition of functions replaces the sequence of steps of an iteration in a sequentially designed procedure. Here especially, side-effects are undesired since there happens a lot on the *call stack* which has to be traceable. We cannot use untraceable side-effects in that process. The functional programming paradigm is encouraged by Logo. It is an advanced technique for more experienced learners. Common Lisp also provides it. We might say, if we don't *need* to use `setf` a lot and only use variables within the lexical scope of their functions (in the sense of procedures) – because we make no use of special variables and only pass *global* variables to the functions as arguments of their function calls – and so we mainly manipulate our data within a long chain of composed functions, we pretty much program in a functional style. 'Plumbing' the functions is therefore a catchy picture.

Imperative Programming Paradigm Harvey focuses on the two mentioned *paradigms*. I have written down my opinion on the term 'paradigm' high above. In the end it does not matter whether it is used or it is replaced by 'style' or something else. However,

I want to mention the *imperative programming paradigm* as well, because it often is confused with the sequential paradigm. Or perhaps it is more appropriate to say that the names *imperative* and *sequential* denote aspects of the same approach and are often used indiscriminately. When *sequential* is used in such a precise way as Harvey does (see above) and emphasises the aspect of iteration, the term *imperative* can be used on a more general level *as the consequence* of the sequential approach: Imperative programming relies heavily on side-effects by straightly mutating places in the register memory. Any iteration needs variables for bookkeeping which are seperated from the repeated expression. On the contrary, with functional programming, the specific local arguments are modified *as a part* of the repeated expression. So the term *imperative programming paradigm* is used for any sequential approaches that uses value assignments massively. It is not a sin. It is just worth it to think about solutions with lesser or no side-effects to avoid misusing them. Special variables have a lot of potential to warm to that theme. However in languages which support functional programming still at the end of a long composed chain of functions representing procedures stands a procedure that causes and uses side-effects to achieve a desired effect. Logo as well as Common Lisp provide imperative programming. – So we might say, if we design our program in a way where we *have to* use `setf` a lot so that we frequently manipulate the relevant places directly and do not control their mutation by letting them run through some functions we 'plumb' together, and if we mainly arrange our functions sequentially with a focus on their effect because for us they represent a step of the recipe we want to define – then we probably program in a more imperative style. So we could say that if we design our program in such a way that we *have to* use `setf` frequently, so that we often manipulate the relevant places directly and do not control their mutation by running them through some functions we have composed, but arrange our functions mainly sequentially and focus on their effect because they represent for us a step of the recipe we want to create - then we are probably programming in an (sequential) imperative style.

Object-oriented Programming Paradigm The *object-oriented programming paradigm* is not *one* approach. It is a set of ideas that are implemented with different focuses. What they do have in common: It is all about the manipulation of places in the register memory so it is all about side-effects. Object-oriented programming is imperative programming but not sequential programming if we look at the whole, even if the single procedures/functions (called *methods*) are also designed sequentially. But the program flow is different: Entities called objects 'come to life' independently from each other seemingly at the same time but also have family relationships. This requires a constant reassignment of the corresponding values to the variables. – Harvey suggests an object-oriented approach called *prototype-based programming* for Logo. This appears as a good idea to me, because it is less complex and therefore, *if* object-orientation shall be part of the Logo learning environment, this should be the most approriate. Object-orientation has a pedagogical value since the organizing of code and data demands the awareness of the relations of data and code and building up a proper taxonomy. This is of great pedagogical value because it supports thinking on a metalevel. – The Common Lisp Object System (CLOS) as mentioned shortly above is of a different kind, so a prototype-based object system has to be designed on top of Common Lisp for the Logo setup. There already exists a custom library for such an prototype-based extension of

Common Lisp which will then be the starting point.

Description of the Behaviour of Logo Procedures This is not a term but a way of using the terms defined so far. A Logo procedure is described by answering these four questions, formulated in Harvey's textbook: 1. Command or operation? 2. How many inputs? 3. What *type* of datum must each input be? 4. If the procedure is an operation, what is its *output*? If a command, what is its *effect*? So applied in two examples from there:

1. '`Sum` is an *operation*. It has two inputs. Both inputs must be numbers. The output from `sum` is a number, the result of adding the two inputs.'

2. '`The` *`command`* `print has one input. The input can be of any datum. The` *`effect`* `of print is to print the input datum on the screen.`'

I think of two layers of describing Logo procedures. As Common Lisp functions and macros their docstrings follow the typical scheme of descriptions of Common Lisp functions. In a Logo environment their description will follow that scheme above.

4.7 Common Lisp Macros

4.8 Developing Macro Solutions for the Setup