The Turing-Complete Text Editor (**Elisp**)

Today we explore Emacs Lisp (or Elisp for short). Elisp is one component of the Emacs, the one and only text editor (unbelievers have rumored that other programs such as Vim are suitable for editing text - this is a lie). It is also just one member of a large family of Lisp languages, (the most prominent today are Scheme and Common Lisp, others include AutoLisp, MacLisp, XLISP, and Lisp Machine Lisp). This gives us two good perspectives to view the language: First in the context of its problem domain of text editing, secondly in comparison to the many other dialects of Lisp.

1 Emacs

Emacs is an historic program, dating back to 1976. Unlike, say, Microsoft Word, programmability has always been a key feature. Emacs grew out of an even older text editor, TECO (dating from 1962!) which was arguably more of a programming language than a text editor. Those of you less engulfed in programming may wonder the point of making a text editor highly programmable, so let's discuss the major uses:

- Configuration: Like any program of similar size, Emacs has myriad options available to the user. Instead of setting options on a GUI (which is also an option in Emacs), users often set them programatically in a config file (their .emacs file). Some may ask what the point of this is. One simple advantage is transparency: if you've customized your setup, it's very clear from your .emacs file what changes you've made, and you can also copy them to a new installation just by copying that file. Of course the same goes for any config file format that *doesn't* offer full programming support.
- Support for new file formats: Emacs has a design goal (or at least a habit) of allowing users to edit almost every programming language (and other structured text) ever invented. Seeing as many programming languages are invented by people who do not know the Emacs developers, this goal would be difficult to achieve if adding support for new programming languages required changing code deep in the core of Emacs. Thankfully, the language author (or anyone else) can write an Emacs Lisp file defining an *editing mode*, a set of commands and options to make editing their favorite language pleasant.
- Text-Heavy Applications: In pursuit of the previous goals, Elisp has a rich library of commands for editing text. As somewhat of a side effect, this means that if you want to write a program that requires manipulating lots of text, it might actually be easier to write in Elisp that a language that doesn't have lots of text facilites (or where UI libraries are all large and complicated). In particular, Elisp has a surprisingly-full-featured Text UI library that can work well for simply applications and mockups.
- Integration: Often the goal of writing an app in Emacs Lisp (such as a terminal emulator, chat client or email client, all of which Emacs has) is not to have lots of good features, but to be integrated seamlessly with the rest of Emacs. It is surprisingly convenient to have a terminal where you can use all the Emacs text-editing commands to massage the input or output to some command-line program.

2 Lisp History

The Lisp family is ancient, dating back to a fateful night in 1958 [1] when John McCarthy (some nerd at MIT) decided to base a practical programming language on Alonzo Church's lambda calculus (which while technically a programming language is much more of a theoretical construct than something usable) and totally misread all the papers. Taking a few steps back:

[1] For all you know it only took a night

In the untyped lambda calculus, everything is a function. We won't torture you with them, but one can use various tricks to encode common data such as integers, lists, pairs, etc. as functions and in fact the notion of a function is enough to express any computation (in the sense that lambda calculus is Turing-Complete). In an attempt to be more practical, Lisp does not make everything a function (it has explicit support for integers, pairs, etc), but it embraces the idea that functions are data and can be manipulated as easily as any other data (the *functional programming* paradigm that most of you have seen in 15-150).

We can classify members of the Lisp familiy in a few ways:

3 Lisp, the Language

3.1 Basic Syntax

Lisp has a shockingly simple, but occasionally unreadable syntax. The primary piece of Lisp syntax is function application, written in parentheses and with arguments separated by spaces, e.g. `(function arg1 arg2)`. Numbers are written as you would expect, in decimal with no extra notation needed, and strings are written in quotes. If you want to write a list, you can either write `(list elem1 elem2 elem3 ... )` or `‘(elem1 elem2 elem3)` (that's a backtick, not an apostrophe).

That's most of the language. While it's not true that every operation in Lisp is a function, most operations at least use this general syntax of space-delimited lists.

3.2 Constructing and Destructing Data

Lists are considered the core data structure of Lisp, though really lists can be decomposed into pairs, which are truly the most primitive data structure. A list is either the empty list (written `nil` or `()`) or a pair containing the first element (*the head*) and the rest of the list (*the tail*). Thus we process lists using the functions for processing pairs:

- `(cons x y)` (short for "construct") returns the pair (x, y)
- `(car xy)` returns the first element of `xy`
- `(cdr xy)` returns the second element of `xy`

As an aside note, `car` stands for "contents of address register" and `cdr` (pronounced "could-er") stands for "contents of decrement register". These abbreviations come from details of the original Lisp implementation, which is a terrible, terrible way to name your language constructs. One wonders why the weren't named "first" and "second" or "fst" and "snd" or Anything. Else.

Elisp also sports four functions `caar, cadr, cdar, cddr` which are just different combinations of `car` and `cdr`. For example `cadr` is the `car` of the `cdr` (i.e. the second element of a list).

For concrete examples: `(car (cons 2 3))` returns 2 while `(cadr ‘(2 3 4))` returns 3.

3.3 Defining Functions

Functions in Elisp are defined using the `defun` keyword (short for "define function") and the following syntax:

```
(defun function-name (arg1 arg2 ... argn)
  BODY)
```

Where BODY is a series of Lisp statements. The result of the function is the value of the last statement. For example the following two definitions of `make2` both return 2:

```
(defun make2 ()
  2)

(defun make2 ()
  3
  2)
```

This is of course a stupid example of a function containing multiple statements. Usually the earlier statements modify the state of the program in some meaningful way, perform I/O, etc.

This is actually not the most general form of a `defun` declaration. You can also include a `doc-string` like so, which provides a handy description of your function's behavior when someone looks it up with Emacs' excellent built-in help system.[2]

This is *still* not the most general form. In the argument list, you can write `&optional ARG-NAME` to specify an optional argument `ARGNAME`, or you can specify `&rest ARGLIST-NAME` to specify that *all* remaining arguments are put in a list named `ARGLIST-NAME`. Whenever you see a variadic function (one that can take any number of arguments), it's probably implemented with `&rest`.

Since Elisp is supposed to be a functional language, it's expected that you can write down a function like any other value, without assigning it a name. This uses the `lambda` keyword, with a similar syntax to `defun`. in general

```
(lambda (arg1 arg2 .. argn)
  BODY)
```

is the function taking n arguments named `arg1` through `argn` with the body BODY. Those of you without any functional programming experience may not see the point of this yet - we'll get there once we've seen more of the language.

3.4 Common Operators

Elisp has most of the common operators that any other language would have, but they're all expressed as function. Common operations:

[2] If you don't already use Emacs' help system, you should. All the help commands start with `Ctrl-h`. Following up with `a` gives you "apropos" searching (fuzzy search by keyword), `f` looks up a function by name, `k` looks up a function by its key-binding, and `m` summarizes all commands in the current mode.

Operation	Example	Result
Addition	`(+ 1 2 3)`	6
Subtraction	`(- 10 1 2 3)`	4
Multiplication	`(* 2 3)`	6
Division	`(/ 6 2)`	3
Equal	`(= 0 0)`	`'t`
Less	`(< 2 1)`	`nil`
Less or Equal	`(<= 2 1)`	`nil`
Greater	`(> 2 1)`	`'t`
Greater or Equal	`(>= 2 2)`	`'t`
Negation	`(not (cons 1 2))`	`nil`

The later examples exhibit one oddity of Lisp: anything except the special constant `nil` is considered true. In the last example, `(cons 1 2)` is considered true, so it's negation is false, which is represented by `nil`. Also note that the arithmetic operations can take many arguments if you so wish.

3.5 Conditionals

The simplest form of conditional in Elisp is `if`. The `if` expression takes a condition, one statement for the true case and any number of statements for the false case. For an example, see our first realistic function: the factorial function!

```
(defun fact (n)
  (if (= n 0)
      1
    (* n (fact (- n 1)))))
```

You may ask: what happens if you want the "true" case of an `if` to contain multiple statements? Elisp has a `progn` keyword that combines multiple statements into one. More precisely, it's a single statement that executes all its arguments in order, so `(progn STATEMENT1 STATEMENT2)` executes `STATEMENT1` and then `STATEMENT2`.

The `cond` keyword is a generalization of `if`: it takes a bunch of condition-body pairs, and executes the body of the first condition that succeeds. Take as our example the `signum` function, which roughly speaking returns the sign of an integer, but as another integer (0 is unchanged, other numbers keep their sign but the magnitude is changed to 1):

```
(defun signum (n)
  (cond ((> n 0) 1)
        ((< n 0) -1)
        (0)))
```

In the first two cases, we check if the number is positive or negative, then return the appropriate result. In this example, we used another feature of `cond`: you can make the last case a catch-all that executes if all the previous clauses fail. To do this, you need only leave off the condition and put the body by itself. So if a number is neither positive or negative, we return 0 without a redundant check that n is indeed 0. The `cond` keyword has no more power than a chain of `if`'s, but is often much more readable.

3.6 Variables

The only variables we've introduced so far are function arguments. In my humble opinion, these are all you really need, but in an appeal to the common man, Elisp offers several alternatives.

Local variables are defined with `let` using the following syntax:

```
(let ((var1 value1)
      (var2 value2)
      ...)
  BODY)
```

The variables defined in the `let` can then be used in `BODY` (but not changed) and become unbound after the end of the `let` statement. There are some more subtleties to variable scope which we'll cover later.

For now, note that the following `let` is invalid:

```
(let ((x 1)
      (y x))
  (+ x y))
```

The problem is that the definition for `y` refers to the variable `x`. For better or worse, one variable's definition cannot refer to another variable defined in the same `let`. Elisp has a variation of `let` called `let*` which makes the variable declarations sequential: you're allowed to refer to any variable defined before you. Using `let*` in the example above makes it behave as intended.

I mentioned that local variables defined in this manner cannot be changed. If you want a variable to be mutable, you (optionally) define it with `(defvar VARNAME)`. You can also use the extended syntax `(defvar VARNAME VALUE)` to give it an initial value. This is not always a good idea. If you run a program multiple times, Emacs won't always restore the initial value of a variable, which can have undesired effects if you're making lots of changes to your program without restarting Emacs.

Once you've defined your variable (or not), use `(set VARNAME VALUE)` to change it. You don't have to do anything special to inspect the value of a mutable variable - in this regard they're identical to other variables.

3.7 Macros

Macros are possibly the coolest feature of the Lisp. While most of their use cases can be handled well be other features in other languages, and sometimes other features of Lisp, they're an incredibly creative and general feature.

Like a function, a macro computes a value, except you don't return that value, you interpret it as Lisp code and execute it. This is getting handwavy, so here's an example. `(defid FUNNAME)` defines `FUNNAME` to be the identity function.

```
(defmacro defid (str)
  (list 'defun str '(x) 'x))
```

Don't get lost in all the quoting (ticks and backticks): The call to `list` returns a list that looks like `(defun str (x) x)` for whatever value of `str` we passed in. This of course defines `str` to be the identity

function. The critical difference between functions and macros we see here is that macros **do not** evaluate their arguments. This gives us freedom to play with the syntax! The "arguments" to a macro don't have to be meaningful Lisp statements as long as they're syntactically valid. In particular, if I call `(defid myid)`, then `myid` wasn't a valid expression because the variable `myid` isn't defined yet, but this is totally allowable in a macro. This makes macros very useful for writing your own constructs that have to define functions, and for extending the language in general.

4 Emacs, the Library

Due to its age and popularity, Emacs has an extensive library, so we won't try to teach everything. The Emacs manual and help system are much more comprehensive resources for that purpose. We're just going to cover some basic text-editing commands.

4.1 Important Emacs Concepts

Skip this bit if you're already an Emacs user. Before you can understand the Emacs editing commands, you first need to understand a few key Emacs concepts:

A *buffer* is (usually) an opened document that you can edit. A buffer does not have to correspond to a physical document on disk, for example if you haven't saved a document yet. There are also interactive buffers that are never meant to be saved. These can contain anything from a terminal emulator to Emacs configuration screens to video games.

A *window* is the UI element that lets you view a buffer. This distinction is important because you can have buffers open that aren't visible in any window, or have the same buffer open in multiple windows. Since editing operations affect the buffer and not the window, changes made in one window will affect all windows containing that buffer.

The *point* is what you call the current position of the cursor within a buffer. Visually, the point is a highlighted rectangle. In code, the point is an integer denoting the distance in characters from the beginning of the file (0 is the first character of the file, 100 is the 101st character, etc.).

The *mark* can be thought of as a "saved point". The command `Ctrl-Space` sets the mark equal to the current point. One use is that you can quickly jump between two positions by swapping mark and point with `Ctrl-x Ctrl-x`. But there is a much more important use:

The section of text between the point and mark is called the *region*. It's extremely common to want to apply some edit to just some section of your document, and most commands will restrict themselves to the current region if one exists. Certain commands, such as cut and copy, only make sense in the present of the region. The standard workflow is to move point to the beginning of the region of interest, set mark with `Ctrl-Space`, move point to the end of the desired region, then execute your command (e.g. cut, which is `Ctrl-w`). In Elisp, many editing functions take a region, expressed as two integers for the start and end positions.

4.2 The functions

To get the current mark and point, use the aptly-named functions `mark` and `point` which take no arguments. They can be set with `set-mark` and `goto-char`, each of which take one argument. The Emacs help warns that you shouldn't call `set-mark` just as a way to remember a position for later. Since positions are just integers, you can simply store the position in a variable instead. If you really do want to change the mark,

consider using `push-mark` which also sets the mark but has a stack-like behavior. That is, if you then call `pop-mark`, you will restore the old value of mark.

Since the region is defined by the mark and point, you might think there's no reason to have extra functions for reading the region. However, note that you often want to ask for the beginning or end of the region, but the mark is allowed to be either before or after the point. The function `region-beginning` returns the start of the region and is equivalent to the following definition:

```
(defun region-beginning ()
  (min (point) (mark)))
```

Similarly, `region-end` gives you the end of the region.

Navigating a document programmatically uses many of the same commands you use when editing manually. `forward-char` and `backward-char` go forward or backward by a character. If given an integer argument n, they instead move by n characters. `next-line` and `previous-line` behave similarly for lines.

The commands `beginning-of-line` and `end-of-line` move to the start or end of the line. `beginning-of-buffer` and `end-of-buffer`, likewise, move to the start or end of a buffer.

What can we do with these commands? Take a minute to write functions `line-length` and `buffer-length` that return the length of a line and buffer.

Solutions:

```
(defun line-length ()
  (beginning-of-line)
  (let ((start (point)))
    (end-of-line)
    (- (point) start)))

(defun buffer-length ()
  (beginning-of-buffer)
  (let ((start (point)))
    (end-of-buffer)
    (- (point) start)))
```

But this code has an insidious problem! Presumably if you're using this function, you just want to be told the length of the current line/buffer, without changing the state of the world. But `end-of-line` and `end-of-buffer` modify the point, so any time you run these commands it moves your cursor! Oi! Annoying! Luckily, the Emacs designers have a solution to this problem:

The `save-excursion` keyword saves the current point/mark/buffer, executes its arguments, then restores them to their original values. The fixed implementation of `line-length` looks like this:

```
(defun line-length ()
  (save-excursion
    (beginning-of-line)
    (let ((start (point)))
      (end-of-line)
      (- (point) start))))
```

Speaking of buffers, you can get the current buffer name with `buffer-name` and switch buffers with `set-buffer`. Although Emacs does not choose to implement it this way, `save-excursion` can actually be

written as a macro, using only features you've learned today. If you're feeling adventurous, try to write it (writing macros in general is tricky until you get used to it, but `save-excursion` is not much trickier than any other macro).

So far we've only moved around the document, but it would be nice to actually edit some text! The function `insert` takes a string and inserts it at the point. This will not have the intended result if you pass an integer - you must call `int-to-string` first. Similarly `(delete-char N)` deletes n characters following point. `kill-line` all text between point and the end of the line, and stores it in the kill ring (Emacs' version of a clipboard).

5 Lisp, the Review

Now that we've introduced enough of the language to do simple tasks, it's time to look at some subtleties of the language, and complain about them:

5.1 Lisp-1 vs. Lisp-2

One of the first issues is the treatment of variables names. In a Lisp-1, function names are treated the same as the names of any other variable: a named function is just a variable that happens to contain a function. This means, for example, that if I have a function named Bob, I can not have an integer named Bob in the same context.

In a Lisp-2, functions are special: the same name can be used both to name a variable and to name a function. To confuse things further, that variable can *contain* another function, which can be different than the function with that name! To resolve this confusion, you cannot directly call a variable containing a function: Lisp-2's have a `funcall` primitive for calling a function inside a variable.

Elisp is a Lisp-2. It is the considered opinion of the instructors that Lisp-2's are stupid. Lisp claims to treat functions as first-class citizens, but the imposing of a separate namespace is an unnecessary act of segregation, and a cause of unnecessary confusion for the newcomer (a.k.a. you).

5.2 Dynamic vs. Lexical Scope

Consider the following code:

```
(defun f1 (x)
  (* x y))

(defun f2 ()
  (let ((y 2))
    (f1 3)))

(f2)
```

What should this code do? One might think this would cause an error because when we evaluate `(* x y)` the variable y would be undefined (because it's only defined in `f2` and not `f1`). This is true in *lexically scoped* Lisp dialects (the vast majority of Lisps), but false in Elisp, which is *dynamically scoped*. In Elisp, the result is 6 because y is still defined. In Elisp, when you call a function, it inherits all the variables of the caller.

We mentioned earlier that McCarthy "completely misunderstood Church's papers". This is what we were referring to: Church's lambda calculus was lexically scoped, as are most modern Lisps. But the original Lisp implementation was dynamically scoped, a mistake that Elisp has inherited and is barely starting to shake off today with the introduction of optional lexical scope in Emacs 24.

It has been argued that this property of Elisp is actually a feature and not a mistake, but we don't really believe it. Here's the argument. Say that twenty years ago you wrote a function `myfunc` that takes 3 arguments `x y z`. Now you decide that you want to add a fourth argument w. You *could* go back and change every occurrence of `(myfunc x y z)` into `(myfunc x y z w)`, but you don't even know most of the people who use your function - you can't change all their code. So instead of making w an argument, you just access it directly, and if anyone wants to take advantage of this argument they have to change their code as follows:

```
(let ((w <value of w>))
  (myfunc x y z))
```

but this also means that people who don't care about w can leave their code alone, which is a Good Thing.

In short, dynamic scope gives you a cheap way to implement optional arguments and maintain backward compatibility with old code.

The tragic downside of this technique is that it makes interfaces extremely unclear in two ways:

1. It's not easy for the caller to know what options they're allowed to pass to the callee. Unlike real arguments, these aren't expressed clearly in the function definition, so the programmer might have to read dozens of functions before they know which arguments are valid.
2. It's unclear which arguments are being passed to a function. Even if you didn't mean to define some variable, maybe your caller did. Maybe your caller's caller's caller's caller did. So it's easy to accidentally invoke some behavior of a function that you really didn't want to. Good luck figuring out what you did, and *where*.

What makes this even harder to excuse is that Elisp has built-in support for functions with optional arguments, in a way that would work without dynamic scope and which avoids both of the problems mentioned above. So why does it still have dynamic scope?