How to Antilisp

Automating programming with code generation

This manual documents Antilisp Guile version 3.0.10.
Copyright (C) 2025 Yacine Cheikhrouhou

Not for selling
Only for personal use

Table of Contents

About this book....4
Why Antilisp?....4
    Abstracting platforms with cross-language programming....5
    Automating code with syntax extensions....6
Representing code....8
Metaprogramming....10
    A first example....11
    Quasiquotes....12
    Hygiene....13
    Gensym....13
Conclusion....14

About this book

This is a guide to the concepts used by the Antilisp language. Unlike most other languages, Antilisp does not try to be a general purpose programming language. Instead, Antilisp focuses on code generation and processing, which is something most developers are not used to.

In addition, Antilisp is a Lisp-based language, and supports metaprogramming. However, advanced Lisp concepts, like hygienic macros, continuations or image-based development are simplified or even removed, in favor of keeping the language easy to approach for people who never used Lisp.

This book explains the most common patterns used in Antilisp, namely AST modeling, code generation and metaprogramming.

Why Antilisp?

Antilisp is a language designed for code generation. This means Antilisp programs are only supposed to output valid code. This includes code in other languages, like C or Javascript, as well as domain-specific languages[1], like Nginx configurations or CI/CD pipelines.

In practical terms, Antilisp is an abstraction over lower level languages. There are two main uses for Antilisp: abstracting platforms and automating code.

The next sections explain in more detail what these terms mean. Please keep in mind that Antilisp still lacks the libraries needed to support common general purpose languages, and these examples cannot be implemented without them.

1 DSLs are languages made for a specific purpose, like configuring a reverse proxy or writing database queries. More information can be found at https://en.wikipedia.org/wiki/Domain-specific_language

Abstracting platforms with cross-language programming

This section explains how Antilisp could, in theory, be used to unify incompatible platforms (namely Python, the web and the JVM)

If you have a Python code base and some day decide it should run in the browser, you either have to rewrite it in JS, because that's what browsers can run, or write a web frontend for a remote session in your server. Both choices are a lot of work, and that's why many people don't do that.

If you use a transpiling language like Haxe or Nim, you can also transpile to Javascript, but then you have to use a new language VM, which will lose some performance due to the glue required to emulate the same runtime in different environments.

Antilisp gives you a new approach: instead of writing your code in Python or Javascript, you can write an Antilisp script that outputs the code you want to run. If you do that, you can easily write the same high level algorithm (with loops, functions, variables,…) and output the same algorithm in both Python and Javascript.

If at some point Python becomes too slow and you want to run the program in Java, you can do the same. You will need to translate the dynamic abstractions you use from Python into valid Java or refactor your code to stop using them, but you will eventually get a program that runs on the JVM, in Python and in a web browser, without ever having to rewrite your code or manage duplicate code bases.

Automating code with syntax extensions

This section explains in details how Antilisp could be used to extend Java with features that are not supported by the language and would usually require boilerplate to reproduce (namely, tagged unions from functional languages[1]).

Let's say you need to use Java, and you also need to use Rust enums (also known as tagged unions).

In Rust, enum values can contain additional data, which is very useful for types that can contain different kinds of content.

In Java, the only way to have data polymorphism is through inheritance: defining a base class (ideally abstract), and then defining child classes which contain specific attributes.

To define a binary tree for example, you would need 3 classes: Tree (abstract class), Node (extends Tree, with a value and left/right subtrees), and Leaf (extends Tree, with no attribute). In Rust, this definition would be as simple as this:

```
enum Tree<T> {
   Node(T, Tree<T>, Tree<T>),
   Leaf
}
```

For the sake of exhaustiveness, the same definition in a functional language looks like this:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

But in Java, you would write 3 classes, in 3 different files:

```
// Tree.java
abstract class Tree<T> {}

// Node.java
class Node<T> extends Tree<T> {
  T value;
  Tree<T> left;
  Tree<T> right;
  public Node(T value_, Tree<T> left_, Tree<T> right_) {
        value = value_;
        left = left_;
        right = right_;
  }
}

// Leaf.java
class Leaf<T> extends Tree<T> {}
```

With Antilisp macros[2], it is very easy to create a new syntax specifically for this pattern:

```
(tagged-union (Tree T)
```

1 https://en.wikipedia.org/wiki/Tagged_union

2 Antilisp did not invent macros, they come from Lisp, the family of programming languages Antilisp belongs to

Why Antilisp?

```
2    (Leaf)
3    (Node
4      value T
5      left (Tree T)
6      right (Tree T)))
```

Which would translate into something like this (assuming this is the API defined by the library you use to generate Java code):

```
 1  (begin
 2    (class (Tree T) abstract)
 3
 4    (class (Leaf T) extends (Tree T))
 5
 6    (class (Node T) extends (Tree T)
 7      (attributes
 8        value T
 9        left (Tree T)
10        right (Tree T))
11      (constructor (value_ T left_ (Tree T) right_ (Tree T))
12          (set value value_)
13          (set left left_)
14          (set right right_))))
```

With these code examples, you can see that by itself, the syntax of Antilisp is not as convenient as having a ready-made support like Rust has for abstractions built into the Rust compiler with existing IDE support. This is something you may want to consider if you have productivity constraints, especially since Antilisp currently lacks IDE support.

However, Antilisp gives you the power to define customized language semantics tailored for your use case, without losing the control over Java. Unlike Clojure, a Lisp for the JVM, you retain full control over the Java code with no performance loss, and without introducing a new general purpose programming language into your code base.

This book gives you the knowledge you need to start writing your own abstractions and code generators for the languages you have to use.

Representing code

Since we use Antilisp to process and generate code, we need a way to represent code in a format that makes it easy to manipulate.

While source code is usually written as human-readable text, this format is difficult to process with code. Programs that manipulate code usually convert the source code into an Abstract Syntax Tree, a hierarchical tree of nodes that represent the contents of a program. If you are not familiar with trees or ASTs, you can think of the DOM (Document Object Model) in web browsers: the DOM is also a tree of objects, where nodes contain child nodes as attributes, and the DOM represents a document whereas an AST represents a program.

The exacts structure of an AST is dictated by the language we are representing. For example, this code:

```
print(“hello world”)
```

Would be represented as a function call object, with the print identifier as the function expression and the string “hello world” as the only argument. Let’s quickly implement all of the classes that are used to represent this program:

```
(defclass node ())
(defclass expression (node))
(defclass identifier (expression)
  name)
(defclass string (expression)
  value)
(defclass funcall (expression)
  function
  arguments)
```

With these primitives, it becomes very easy to represent the previous code with objects, although this API would be too cumbersome in practice:

```
(make-funcall
  (make-identifier “print”)
  (list
    (make-string “hello world”)))
```

From this object representation, we can, of course, generate the text-based source code to match this program:

```
(defgeneric (render x))
(defmethod (render (x identifier))
  # “print” is rendered as print
  (name identifier))
(defmethod (render (x string))
  # “hello world” is rendered as “hello world”
```

Representing code

```
 7    (repr (value x)))
 8
 9  (def (render-call-args args)
10    (let rendered (map render args)
11      (join rendered ", ")))
12  (defmethod (render (x funcall))
13    # funcall of f with args (a b c) is rendered as f(a, b, c)
14    (+
15      (render (function x))
16      "("
17      (render-call-args (arguments x))
18      ")"))
```

Rendering a function is slightly more complicated than rendering simple values like strings. Nevertheless, we can now generate back the code we started with:

```
1  (render (make-funcall
2            (make-identifier "print")
3            (list
4              (make-string "hello world"))))
5  → print("hello world")
```

To improve the syntax we use to describe programs, we can:

- use Antilisp native data types like strings and symbols instead of wrapper objects with constructors
- add wrappers to build syntax trees as if writing natural code

So that we can write this instead:

```
1  (def (print x)
2    (make-funcall 'print x))
3  (render (print "hello world"))
4  → print("hello world")
```

The implementation of this is left as an exercise to the reader.

Metaprogramming

In the previous section, we limited ourselves to using classes, functions and methods to create and render code.

While this approach works for most cases, sometimes the syntax of Antilisp will get in your way.

Let’s say I want to write Python code without declaring wrappers like (print) or (len) manually:

```
(as-python
  (print (len “hello world”)))
```

If (print) and (len) were to be executed as function calls, this wouldn’t work because they were not declared. With macros, we can transform Antilisp code inside (as-python) blocks before executing them.

In this chapter, we will implement our own if statement. The implementation of the (as-python) macro described in this introduction would be too complex for this tutorial, but can be implemented with the same principles, if you combine what you learn here with the code written in the previous chapter.

A first example

Let’s start with a simple macro example called my-if:

```
(my-if false
  (print “true”))
```

If we were to implement (my-if) as a function, the arguments false and (print “true”) would be executed before calling (my-if):

```
(def (my-if condition code)
  (if condition
    code
    nil))
```

What we want is to only execute the print instruction if the condition is true, like (if) does. This means (my-if) must run before its parameters are evaluated, and only the result returned by (my-if) must be evaluated (to prevent the code from being executed if the condition is not true).

We can do this by defining (my-if) as a macro instead:

```
(defmacro (my-if condition code)
  (if condition
    code
    nil))
```

This macro will simply take false and (print “true”) as arguments, and return nil because false is not true. The print statement will never be executed.

You may ask: false is a normal value, but (print “true”) is code, so how can this piece of code be used as an argument ? The answer is simple: code is data[1]. (print “true”) is a list, just like (list 1 2) would return the list (1 2).

When (my-if false (print “true”)) is executed, the (my-if) macro will be called before anything else to process its arguments and return the code that should be evaluated, which is nil in this case.

Congratulations! You wrote your first macro. While not very useful, we just made a compile-time if, which many languages don’t have. Unlike the normal if syntax, my-if is executed before your program, and modifies the source code directly. In a compiled language, this could be used to implement changing behaviors for different compilation targets.

1 “Code is data”, or homoiconicity, is the reason why the syntax of Antilisp is based on space-separated parenthesis blocks: every block is in fact a list.

Quasiquotes

Sometimes we also need macros to generate new code dynamically.

If you wanted to perfectly clone the builtin if with a my-if macro, you could return a block of code that is an if block, by building the (if condition code) list manually:

```
(defmacro (my-if condition code)
  (list 'if condition code))
```

While this works for small definitions, building a program manually makes it harder to read and gets very complicated for large chunks of code. If I want to generate this program:

```
(if (is-list x)
  (print (at 0 x)))
```

I would have to write this:

```
(list 'if
      (list 'is-list 'x)
      (list 'print (list 'at 0 'x)))
→ (if (is-list x) (print (at 0 x)))
```

For static code, quote can be used to get code as data:

```
'(if (is-list x)
  (print (at 0 x)))
→ (if (is-list x) (print (at 0 x)))
```

Quasiquote is used the same way as quote, but allows us to evaluate (unquote) some parts inside the code. Essentially, quasiquote lets us write code templates with placeholders for values:

```
(quasiquote (if (unquote condition) (unquote code)))
```

Like quote ('), quasiquote (`) and unquote (,) have special characters:

```
(defmacro (my-if condition code)
  `(if ,condition ,code))
(my-if false (print "true"))  # expands to (if false (print "true"))
```

Quasiquote also accepts unquote-splicing (,@) terms, which flattens lists. This code will return (1 2 3 4 5):

```
(let dynamic (list 3 4 5)
  `(1 2 ,@dynamic))
```

Hygiene

We have one last step before completely replacing (if) with (my-if).

In the current state, (my-if) transforms the code to use (if) at runtime. This means the code generated by (my-if) will not run without a globally defined (if). Even worse, if you try to replace (if) with (my-if), this will create an infinite recursion loop, since (my-if) will return a new call to (my-if).

To make sure the code we generate still uses the original (if) even after it is replaced, we can simply use the value of if instead of the if symbol by unquoting if in the quasiquote block:

```
1  (defmacro (my-if condition code)
2    `(,if ,condition ,code))
```

That's it. We can now override (if) with our own implementation:

```
1  (define if my-if)
```

Note: the pattern of keeping macros safe from side effects is called hygiene in Lisp parlance.

Gensym

The last tool you need with macros is (gensym). Gensym is a function that returns a new, unique symbol. We can use it to define new variables in the code we generate without having conflicts with the variables created by the user of our macros:

```
1  (defmacro (time-it . body)
2    (let start (gensym)
3      `(let ,start (get-time)
4        ,@body
5        (print (- (get-time) ,start)))))
6  (time-it
7    (for i (range 10000000)
8      (print i)))
```

Note: the (get-time) function does not actually exist in Antilisp, and cannot be implemented currently because Antilisp has no way of interacting with the environment.

Conclusion

Thank you for reading up to this point.

While there are many concepts that I had to skip to keep things simple, and some things are not yet supported by Antilisp, I hope this short introduction will be enough to help you get started with Antilisp.

This guide should evolve and stay updated with the language, so you may find new things if you come back in the future. The updated version of this book can always be found on the Antilisp website.