21 Data Serialization with S-Expressions

S-expressions are nested parenthetical expressions whose atomic values are strings. They were first popularized by the Lisp programming language in the 1960s, and have remained one of the simplest and most effective ways to encode structured data in a human-readable and editable form.

An example s-expression might look like this.

```
(this (is an) (s expression))
```

S-expressions play a major role in Base and Core, effectively acting as the default serialization format. Indeed, we've encountered s-expressions multiple times already, including in Chapter 8 (Error Handling), Chapter 11 (Functors), and Chapter 12 (First-Class Modules).

This chapter will go into s-expressions in more depth. In particular, we'll discuss:

- The details of the s-expression format, including how to parse it while generating good error messages for debugging malformed inputs
- How to generate converters between s-expressions and arbitrary OCaml types
- How to use annotations to control the behavior of these generated converters
- How to integrate s-expressions into your interfaces, in particular how to add s-expression converters to a module without breaking abstraction boundaries

We'll tie this together at the end of the chapter with a simple s-expression formatted configuration file for a web server

21.1 Basic Usage

The type used to represent an s-expression is quite simple:

```
module Sexp : sig
  type t =
  | Atom of string
  | List of t list
end
```

An s-expression can be thought of as a tree where each node contains a list of its children, and where the leaves of the tree are strings. Core provides good support for s-expressions in its `Sexp` module, including functions for converting s-expressions to and from strings. Let's rewrite our example s-expression in terms of this type:

```
# open Core;;
# Sexp.List [
    Sexp.Atom "this";
    Sexp.List [ Sexp.Atom "is"; Sexp.Atom "an"];
    Sexp.List [ Sexp.Atom "s"; Sexp.Atom "expression" ];
  ];;
- : Sexp.t = (this (is an) (s expression))
```

This prints out nicely because Core registers a pretty printer with the toplevel. This pretty printer is based on the functions in `Sexp` for converting s-expressions to and from strings:

```
# Sexp.to_string (Sexp.List [Sexp.Atom "1"; Sexp.Atom "2"]);;
- : string = "(1 2)"
# Sexp.of_string ("(1 2 (3 4))");;
- : Sexp.t = (1 2 (3 4))
```

Base, Core, and Parsexp

In these examples, we're using Core rather than Base because Core has integrated support for parsing s-expressions, courtesy of the `Parsexp` library. If you just use Base, you'll find that you don't have `Sexp.of_string` at your disposal.

```
# open Base;;
# Sexp.of_string "(1 2 3)";;
Line 1, characters 1-15:
Alert deprecated: Base.Sexp.of_string
[since 2018-02] Use [Parsexp.Single.parse_string_exn]
Line 1, characters 1-15:
Error: This expression has type unit
       This is not a function; it cannot be applied.
```

That's because, in an attempt to keep `Base` light, the s-expression parsing functions aren't included. That said, you can always use them by calling out to the corresponding functions from the `Parsexp` library:

```
# Parsexp.Single.parse_string_exn "(1 2 3)";;
- : Sexp.t = (1 2 3)
```

In addition to providing the `Sexp` module, most of the base types in Base and Core support conversion to and from s-expressions. For example, we can use the conversion functions defined in the respective modules for integers, strings, and exceptions:

```
# Int.sexp_of_t 3;;
- : Sexp.t = 3
# String.sexp_of_t "hello";;
- : Sexp.t = hello
# Exn.sexp_of_t (Invalid_argument "foo");;
- : Sexp.t = (Invalid_argument foo)
```

It's also possible to convert container types such as lists or arrays that are polymorphic over the type of data they contain.

```
# #show List.sexp_of_t;;
val sexp_of_t : ('a -> Sexp.t) -> 'a list -> Sexp.t
```

Notice that `List.sexp_of_t` is polymorphic and takes as its first argument another conversion function to handle the elements of the list to be converted. Base and Core use this scheme more generally for defining sexp converters for polymorphic types. Here's an example of it in action.

```
# List.sexp_of_t Int.sexp_of_t [1; 2; 3];;
- : Sexp.t = (1 2 3)
```

The functions that go in the other direction, *i.e.*, reconstruct an OCaml value from an s-expression, use essentially the same trick for handling polymorphic types, as shown below.

```
# List.t_of_sexp Int.t_of_sexp (Sexp.of_string "(1 2 3)");;
- : int list = [1; 2; 3]
```

Such a function will fail with an exception when presented with an s-expression that doesn't match the structure of the OCaml type in question.

```
# List.t_of_sexp Int.t_of_sexp (Sexp.of_string "(1 2 three)");;
Exception:
(Of_sexp_error "int_of_sexp: (Failure int_of_string)" (invalid_sexp
    three))
```

More on Top-Level Printing

The values of the s-expressions that we created were printed properly as s-expressions in the toplevel, instead of as the tree of `Atom` and `List` variants that they're actually made of.

This is due to OCaml's facility for installing custom *top-level printers* that can rewrite some values into more top-level-friendly equivalents. They are generally installed as `ocamlfind` packages ending in `top`:

```
$ ocamlfind list | grep top
astring.top          (version: 0.8.3)
cohttp.top           (version: n/a)
compiler-libs.toplevel (version: [distributed with Ocaml])
core.top             (version: v0.10.0)
...
uri.top              (version: 1.9.6)
utop                 (version: 2.1.0)
```

The `core.top` package (which you should have loaded by default in your `.ocamlinit` file) loads in printers for the Core extensions already, so you don't need to do anything special to use the s-expression printer.

21.1.1 S-Expression Converters for New Types

But what if you want a function to convert a brand new type to an s-expression? You can of course write it yourself manually. Here's an example.

```
# type t = { foo: int; bar: float };;
type t = { foo : int; bar : float; }
# let sexp_of_t t =
    let a x = Sexp.Atom x and l x = Sexp.List x in
    l [ l [a "foo"; Int.sexp_of_t t.foo  ];
        l [a "bar"; Float.sexp_of_t t.bar]; ];;
val sexp_of_t : t -> Sexp.t = <fun>
# sexp_of_t { foo = 3; bar = -5.5 };;
- : Sexp.t = ((foo 3) (bar -5.5))
```

This is somewhat tiresome to write, and it gets more so when you consider the parser, i.e., t_of_sexp, which is considerably more complex. Writing this kind of parsing and printing code by hand is mechanical and error prone, not to mention a drag.

Given how mechanical the code is, you could imagine writing a program that inspects the type definition and automatically generates the conversion code for you. As it turns out, there's a *syntax extension* called ppx_sexp_conv which does just that, creating the required functions for every type annotated with [@@deriving sexp]. To enable ppx_sexp_conv, we're going to enable ppx_jane, which is a larger collection of useful extensions that includes ppx_sexp_conv.

```
# #require "ppx_jane";;
```

We can use the extension as follows.

```
# type t = { foo: int; bar: float } [@@deriving sexp];;
type t = { foo : int; bar : float; }
val t_of_sexp : Sexp.t -> t = <fun>
val sexp_of_t : t -> Sexp.t = <fun>
# t_of_sexp (Sexp.of_string "((bar 35) (foo 3))");;
- : t = {foo = 3; bar = 35.}
```

The syntax extension can be used outside of type declarations as well. As discussed in Chapter 8 (Error Handling), [@@deriving sexp] can be attached to the declaration of an exception to improve the quality of errors printed by OCaml's top-level exception handler.

Here are two exception declarations, one with an annotation, and one without:

```
exception Ordinary_exn of string list;;
exception Exn_with_sexp of string list [@@deriving sexp];;
```

And here's the difference in what you see when you throw these exceptions.

```
# raise (Ordinary_exn ["1";"2";"3"]);;
Exception: Ordinary_exn(_).
# raise (Exn_with_sexp ["1";"2";"3"]);;
Exception: (//toplevel//.Exn_with_sexp (1 2 3))
```

ppx_sexp_conv also supports inline declarations that generate converters for anonymous types.

```
# [%sexp_of: int * string ];;
- : int * string -> Sexp.t = <fun>
# [%sexp_of: int * string ] (3, "foo");;
- : Sexp.t = (3 foo)
```

The syntax extensions bundled with Base and Core almost all have the same basic structure: they auto-generate code based on type definitions, implementing functionality that you could in theory have implemented by hand, but with far less programmer effort.

Syntax Extensions and PPX

OCaml doesn't directly support deriving s-expression converters from type definitions. Instead, it provides a mechanism called *PPX* which allows you to add to the compilation pipeline code for transforming OCaml programs at the syntactic level, via the `-ppx` compiler flag.

PPXs operate on OCaml's *abstract syntax tree*, or AST, which is a data type that represents the syntax of a well-formed OCaml program. Annotations like `[%sexp_of: int]` or `[@@deriving sexp]` are part of special extensions to the syntax, called *extension points*, which were added to the language to give a place to put information that would be consumed by syntax extensions like `ppx_sexp_conv`.

`ppx_sexp_conv` is part of a family of syntax extensions, including `ppx_compare`, described in Chapter 15 (Maps and Hash Tables), and `ppx_fields`, described in Chapter 6 (Records), that generate code based on type declarations.

Using these extensions from a `dune` file is as simple as adding this directive to a `(library)` or `(executable)` stanza to indicate that the files should be run through a preprocessor:

```
(executable
  (name hello)
  (preprocess (pps ppx_sexp_conv))
)
```

21.2 The Sexp Format

The textual representation of s-expressions is pretty straightforward. An s-expression is written down as a nested parenthetical expression, with whitespace-separated strings as the atoms. Quotes are used for atoms that contain parentheses or spaces themselves; backslash is the escape character; and semicolons are used to introduce single-line comments. Thus, the following file, `example.scm`:

```
((foo 3.3) ;; This is a comment
 (bar "this is () an \" atom"))
```

can be loaded as follows.

```
# Sexp.load_sexp "example.scm";;
- : Sexp.t = ((foo 3.3) (bar "this is () an \" atom"))
```

As you can see, the comment is not part of the loaded s-expression.

All in, the s-expression format supports three comment syntaxes:

; Comments out everything to the end of line
#|,|# Delimiters for commenting out a block
#; Comments out the first complete s-expression that follows

The following example shows all of these in action:

```
;; comment_heavy_example.scm
((this is included)
 ; (this is commented out
 (this stays)
 #; (all of this is commented
     out (even though it crosses lines.))
  (and #| block delimiters #| which can be nested |#
     will comment out
    an arbitrary multi-line block))) |#
   now we're done
   ))
```

Again, loading the file as an s-expression drops the comments:

```
# Sexp.load_sexp "comment_heavy.scm";;
- : Sexp.t = ((this is included) (this stays) (and now we're done))
```

If we introduce an error into our s-expression, by, say, creating a file `broken_example.scm` which is `example.scm`, without the open-paren in front of `bar`, we'll get a parse error:

```
# Sexp.load_sexp "example_broken.scm";;
Exception:
(Sexplib.Sexp.Parse_error
  ((err_msg "unexpected character: ')'") (text_line 4) (text_char 30)
    (global_offset 78) (buf_pos 78)))
```

21.3 Preserving Invariants

Modules and module interfaces are an important part of how OCaml code is structured and designed. One of the key reasons we use module interfaces is to make it possible to enforce invariants. In particular, by restricting how values of a given type can be created and transformed, interfaces let you enforce various rules, including ensuring that your data is well-formed.

When you add s-expression converters (or really any deserializer) to an API, you're adding an alternate path for creating values, and if you're not careful, that alternate path can violate the carefully maintained invariants of your code.

In the following, we'll show how this problem can crop up, and how to resolve it. Let's consider a module `Int_interval` for representing closed integer intervals, similar to the one described in Chapter 11 (Functors).

Here's the signature.

```
type t [@@deriving sexp]

(** [create lo hi] creates an interval from [lo] to [hi] inclusive,
   and is empty if [lo > hi]. *)
val create : int -> int -> t

val is_empty : t -> bool
val contains : t -> int -> bool
```

In addition to basic operations for creating and evaluating intervals, this interface also exposes s-expression converters. Note that the `[@@deriving sexp]` syntax works in a signature as well, but in this case, it just adds the signature for the conversion functions, not the implementation.

Here's the implementation of `Int_interval`.

```
open Core

(* for [Range (x,y)], we require that [y >= x] *)
type t =
  | Range of int * int
  | Empty
[@@deriving sexp]

let create x y = if x > y then Empty else Range (x, y)

let is_empty = function
  | Empty -> true
  | Range _ -> false

let contains i x =
  match i with
  | Empty -> false
  | Range (low, high) -> x >= low && x <= high
```

One critical invariant here is that `Range` is only used to represent non-empty intervals. A call to `create` with a lower bound above the upper bound will return an `Empty`.

Now, let's demonstrate the functionality with some tests, using the expect test framework described in Chapter 18.2 (Expect Tests). First, we'll write a test helper that takes an interval and a list of points, and prints out the result of checking for emptiness, and a classification of which points are inside and outside the interval.

```
let test_interval i points =
  let in_, out =
    List.partition_tf points ~f:(fun x -> Int_interval.contains i x)
  in
  let to_string l =
    List.map ~f:Int.to_string l |> String.concat ~sep:", "
  in
  print_endline
    (String.concat
       ~sep:"\n"
       [ (if Int_interval.is_empty i then "empty" else "non-empty")
       ; "in:  " ^ to_string in_
       ; "out: " ^ to_string out
       ])
```

We can run this test on a non-empty interval,

```
let%expect_test "ordinary interval" =
  test_interval (Int_interval.create 3 6) (List.range 1 10);
  [%expect
    {|
    non-empty
    in:  3, 4, 5, 6
    out: 1, 2, 7, 8, 9 |}]
```

And also on an empty one.

```
let%expect_test "empty interval" =
  test_interval (Int_interval.create 6 3) (List.range 1 10);
  [%expect {|
    empty
    in:
    out: 1, 2, 3, 4, 5, 6, 7, 8, 9 |}]
```

Note that the result of checking `is_empty` lines up with the test of what elements are contained and not contained in the interval.

Now, let's test out the s-expression converters, starting with `sexp_of_t`. This test lets you see that a flipped-bounds interval is represented by `Empty`.

```
let%expect_test "test to_sexp" =
  let t lo hi =
    let i = Int_interval.create lo hi in
    print_s [%sexp (i : Int_interval.t)]
  in
  t 3 6;
  [%expect {| (Range 3 6) |}];
  t 4 4;
  [%expect {| (Range 4 4) |}];
  t 6 3;
  [%expect {| Empty |}]
```

The next thing to check is the `t_of_sexp` converters, and here, we run into a problem. In particular, consider what would happen if we create an interval from the s-expression `(Range 6 3)`. That's an s-expression that shouldn't ever be generated by the library, since intervals should never have swapped bounds. But there's nothing to stop us from generating that s-expression by hand.

```
let%expect_test "test (range 6 3)" =
  let i = Int_interval.t_of_sexp (Sexp.of_string "(Range 6 3)") in
  test_interval i (List.range 1 10);
  [%expect
    {|
    non-empty
    in:
    out: 1, 2, 3, 4, 5, 6, 7, 8, 9 |}]
```

You can see something bad has happened, since this interval is detected as non-empty, but doesn't appear to contain anything. The problem traces back to the fact that `t_of_sexp` doesn't check the same invariant that `create` does. We can fix this, by overriding the auto-generated s-expression converter with one that checks the invariant, in this case, by calling `create`.

```
let t_of_sexp sexp =
  match t_of_sexp sexp with
  | Empty -> Empty
  | Range (x, y) -> create x y
```

Overriding an existing function definition with a new one is perfectly acceptable in OCaml. Since `t_of_sexp` is defined with an ordinary `let` rather than a `let rec`, the call to the `t_of_sexp` goes to the derived version of the function, rather than being a recursive call.

Note that, rather than fixing up the invariant, we could have instead thrown an exception if the invariant was violated. In any case, the approach we took means that rerunning our test produces a more consistent and sensible result.

```
let%expect_test "test (range 6 3)" =
  let i = Int_interval.t_of_sexp (Sexp.of_string "(Range 6 3)") in
  test_interval i (List.range 1 10);
  [%expect
    {|
    empty
    in:
    out: 1, 2, 3, 4, 5, 6, 7, 8, 9 |}]
```

21.4 Getting Good Error Messages

There are two steps to deserializing a type from an s-expression: first, converting the bytes in a file to an s-expression; and the second, converting that s-expression into the type in question. One problem with this is that it can be hard to localize errors to the right place using this scheme. Consider the following example.

```
open Core

type t =
  { a : string
  ; b : int
  ; c : float option
  }
[@@deriving sexp]

let () =
  let t = Sexp.load_sexp "example.scm" |> t_of_sexp in
  printf "b is: %d\n%!" t.b
```

If you were to run this on a malformatted file, say, this one:

```
((a not-a-string)
 (b not-a-string)
 (c 1.0))
```

you'll get the following error. (Note that we set the `OCAMLRUNPARAM` environment variable to suppress the stack trace here.)

```
$ OCAMLRUNPARAM=b=0 dune exec -- ./read_foo.exe
Uncaught exception:

  (Of_sexp_error "int_of_sexp: (Failure int_of_string)"
   (invalid_sexp not-a-string))

[2]
```

If all you have is the error message and the string, it's not terribly informative. In particular, you know that the parsing errored out on the atom "not-an-integer," but you don't know which one! In a large file, this kind of bad error message can be pure misery.

But there's hope! We can make a small change to the code to improve the error message greatly:

```
open Core

type t =
  { a : string
  ; b : int
  ; c : float option
  }
[@@deriving sexp]

let () =
  let t = Sexp.load_sexp_conv_exn "example.scm" t_of_sexp in
  printf "b is: %d\n%!" t.b
```

If we run it again, we'll see a more informative error.

```
$ OCAMLRUNPARAM=b=0 dune exec -- ./read_foo.exe
Uncaught exception:

  (Of_sexp_error example.scm:2:4 "int_of_sexp: (Failure
    int_of_string)"
   (invalid_sexp not-an-integer))

[2]
```

Here, `example.scm:2:5` tells us that the error occurred in the file `"example.scm"` on line 2, character 5. This is a much better start for figuring out what went wrong. The ability to find the precise location of the error depends on the sexp converter reporting errors using the function `of_sexp_error`. This is already done by converters generated by `ppx_sexp_conv`, but you should make sure to do the same when you write custom converters.

21.5 Sexp-Conversion Directives

`ppx_sexp_conv` supports a collection of directives for modifying the default behavior of the auto-generated sexp converters. These directives allow you to customize the way in which types are represented as s-expressions without having to write a custom converter.

21.5.1 `@sexp.opaque`

The most commonly used directive is `[@sexp_opaque]`, whose purpose is to mark a given component of a type as being unconvertible. Anything marked with the `[@sexp.opaque]` attribute will be presented as the atom `<opaque>` by the to-sexp converter, and will trigger an exception from the from-sexp converter.

Note that the type of a component marked as opaque doesn't need to have a sexp converter defined. By default, if we define a type without a sexp converter and then try to use it as part of another type with a sexp converter, we'll error out:

```
# type no_converter = int * int;;
type no_converter = int * int
# type t = { a: no_converter; b: string } [@@deriving sexp];;
Line 1, characters 15-27:
Error: Unbound value no_converter_of_sexp
```

But with `[@sexp.opaque]`, we can embed our opaque `no_converter` type within the other data structure without an error.

```
type t =
  { a: (no_converter [@sexp.opaque]);
    b: string
  } [@@deriving sexp];;
```

And if we now convert a value of this type to an s-expression, we'll see the contents of field a marked as opaque:

```
# sexp_of_t { a = (3,4); b = "foo" };;
- : Sexp.t = ((a <opaque>) (b foo))
```

Note that `t_of_sexp` is still generated, but will fail at runtime when called.

```
# t_of_sexp (Sexp.of_string "((a whatever) (b foo))");;
Exception:
(Of_sexp_error "opaque_of_sexp: cannot convert opaque values"
  (invalid_sexp whatever))
```

It might seem perverse to create a parser for a type containing a `[@sexp.opaque]` value, but it's not as useless as it seems. In particular, such a converter won't necessarily fail on all inputs. Consider a record containing a list of opaque values:

```
type t =
  { a: (no_converter [@sexp.opaque]) list;
    b: string
  } [@@deriving sexp];;
```

The `t_of_sexp` function can still succeed, as long as the list is empty.

```
# t_of_sexp (Sexp.of_string "((a ()) (b foo))");;
- : t = {a = []; b = "foo"}
```

Sometimes, though, one or other of the converters is useless, and you want to explicitly choose what to generate. You can do that by using `[@@deriving sexp_of]` or `[@@deriving of_sexp]` instead of `[@@deriving sexp]`.

21.5.2 `@sexp.list`

Sometimes, sexp converters have more parentheses than one would ideally like. Consider the following variant type.

```
type compatible_versions =
  | Specific of string list
  | All
[@@deriving sexp];;
```

Here's what the concrete syntax looks like.

```
# sexp_of_compatible_versions
    (Specific ["3.12.0"; "3.12.1"; "3.13.0"]);;
- : Sexp.t = (Specific (3.12.0 3.12.1 3.13.0))
```

The set of parens around the list of versions is arguably excessive. We can drop those parens using the [@sexp.list] directive.

```
type compatible_versions =
  | Specific of string list [@sexp.list]
  | All [@@deriving sexp]
```

And here's the resulting lighter syntax.

```
# sexp_of_compatible_versions
    (Specific ["3.12.0"; "3.12.1"; "3.13.0"]);;
- : Sexp.t = (Specific 3.12.0 3.12.1 3.13.0)
```

21.5.3 @sexp.option

By default, optional values are represented either as () for None. Here's an example of a record type containing an option:

```
type t =
  { a: int option;
    b: string;
  } [@@deriving sexp]
```

And here's what the concrete syntax looks like.

```
# sexp_of_t { a = None; b = "hello" };;
- : Sexp.t = ((a ()) (b hello))
# sexp_of_t { a = Some 3; b = "hello" };;
- : Sexp.t = ((a (3)) (b hello))
```

This all works as you might expect, but in the context of a record, you might want a different behavior, which is to make the field itself optional. The [@sexp.option] directive gives you just that.

```
type t =
  { a: int option [@sexp.option];
    b: string;
  } [@@deriving sexp]
```

And here is the new syntax. Note that when the value of a is Some, it shows up in the s-expression unadorned, and when it's None, the entire record field is omitted.

```
# sexp_of_t { a = Some 3; b = "hello" };;
- : Sexp.t = ((a 3) (b hello))
# sexp_of_t { a = None; b = "hello" };;
- : Sexp.t = ((b hello))
```

21.5.4 Specifying Defaults

[@sexp.option] gives you a way of interpreting the s-expression for a record where some of the fields are left unspecified. The [@default] directive provides another.

Consider the following type, which represents the configuration of a very simple web server:

```
type http_server_config = {
  web_root: string;
  port: int;
  addr: string;
} [@@deriving sexp];;
```

One could imagine making some of these parameters optional; in particular, by default, we might want the web server to bind to port 80, and to listen as localhost. We can do this as follows:

```
type http_server_config = {
  web_root: string;
  port: int [@default 80];
  addr: string [@default "localhost"];
} [@@deriving sexp];;
```

Now, if we try to convert an s-expression that specifies only the `web_root`, we'll see that the other values are filled in with the desired defaults:

```
# let cfg =
    "((web_root /var/www/html))"
    |> Sexp.of_string
    |> http_server_config_of_sexp;;
val cfg : http_server_config =
  {web_root = "/var/www/html"; port = 80; addr = "localhost"}
```

If we convert the configuration back out to an s-expression, you'll notice that all of the fields are present, even though they're not strictly necessary:

```
# sexp_of_http_server_config cfg;;
- : Sexp.t = ((web_root /var/www/html) (port 80) (addr localhost))
```

We could make the generated s-expression also drop default values, by using the `[@sexp_drop_default]` directive:

```
type http_server_config = {
  web_root: string;
  port: int [@default 80] [@sexp_drop_default.equal];
  addr: string [@default "localhost"] [@sexp_drop_default.equal];
} [@@deriving sexp];;
```

And here's an example of it in action:

```
# let cfg =
    "((web_root /var/www/html))"
    |> Sexp.of_string
    |> http_server_config_of_sexp;;
val cfg : http_server_config =
  {web_root = "/var/www/html"; port = 80; addr = "localhost"}
# sexp_of_http_server_config cfg;;
- : Sexp.t = ((web_root /var/www/html))
```

As you can see, the fields that are at their default values are omitted from the generated s-expression. On the other hand, if we convert a config with non-default values, they will show up in the generated s-expression.

```
# sexp_of_http_server_config { cfg with port = 8080 };;
- : Sexp.t = ((web_root /var/www/html) (port 8080))
# sexp_of_http_server_config
  { cfg with port = 8080; addr = "192.168.0.1" };;
- : Sexp.t = ((web_root /var/www/html) (port 8080) (addr 192.168.0.1))
```

This can be very useful in designing config file formats that are both reasonably terse and easy to generate and maintain. It can also be useful for backwards compatibility: if you add a new field to your config record but make that field optional, then you should still be able to parse older versions of your config.

The exact attribute you use depends on the comparison functions available over the type that you wish to drop:

- `[@sexp_drop_default.compare]` if the type supports `[%compare]`
- `[@sexp_drop_default.equal]` if the type supports `[%equal]`
- `[@sexp_drop_default.sexp]` if you want to compare the sexp representations
- `[@sexp_drop_default f]` and give an explicit equality function

Most of the type definitions supplied with Base and Core provide the comparison and equality operations, so those are reasonable default attributes to use.