Just Enough Nixlang (Nix From First Principles: Flake Edition #4)

Oct 05
2022

#Nix

This is part 4 of the Nix from First Principles: Flake Edition series.

This is a crash course in Nix's language, which I'll also sometimes call Nixlang from here on, to avoid confusing it with the Nix package manager. For more detailed instructions see the manual. The Nix language is used heavily within Nix, for defining packages, environments, runnable commands and library functions.

Nix's language is expression oriented. This means everything, including up to the level of an entire program, has an output value. It will be most familiar if you have experimented with functional langages such as Haskell, but it is less complicated than those languages, so you don't need to have mastered one to start.

To start experimenting with it, you can run the `nix repl` command which will let you type in Nix expressions and see the output.

Simple data types

Nix's basic data types include some you will be familiar with from any language, like numbers or strings.

```
# Numbers are typed as is
nix-repl> 1
1


# Strings are surrounded by double quotes
nix-repl> "a"
"a"

# Lists are surrounded in square brackets,
# and list items are seperated by white space
nix-repl> [ "a" "b" ]
[ "a" "b" ]
```

However, since the language was designed for package management, it also has some first class data types that other languages do not. The most common of these is the path.

```
nix-repl> ./README.md
/home/tony/code/nix-guide/README.md
```

Path literals are defined by the presence of the `/` forward slash character. This means that for paths in the current directory, you will need to prefix them with `./` to seperate them from identifiers.

Nix sets (remember, these are maps/dictionaries in more common terms) are defined with curly braces, with `=` to seperate keys from values and `;` at the end of every entry. You'll also see them referred to as *attribute sets* or *attrs* - these all have the same meaning.

Sets

```
# Sets are written in curly braces

nix-repl> { a = 1; b = 2; }
{ a = 1; b = 2; }
```

Notice that every key value pair is terminated by a semicolon - this is not optional, even on the final entry, or on single entry lists.

Because set operations, including on nested sets are so common in Nix, there's a lot of syntax sugar for sets.

```
# You can define nested sets using dots in key names
nix-repl> { a.x = 1; a.y = 2; b.c = 4; }
{ a = { ... }; b = { ... }; }

# Use the :p command at the repl to print recursively
nix-repl> :p { a.x = 1; a.y = 2; b.c = 4; }
{ a = { x = 1; y = 2; }; b = { c = 4; }; }
```

Functions

For the final major type of the language, functions are written as a `argument: expression` and called by using the function name and a space.

```
nix-repl> x = arg: { foo = arg; }

nix-repl> x 123
{ foo = 123; }
```

You can use destructuring to extract fields from sets as arguments. This gives an argument passing style similar to keyword arguments in languages like Python or options objects in JavaScript.

```
nix-repl> x = { foo, bar }: {a = foo; b = bar; }
```

```
nix-repl> x { foo = 1; bar = 2; }
{ a = 1; b = 2; }
```

But what if you want multiple positional arguments? Similar to languages like Haskell, functions use what is known as currying to handle multiple arguments. This means that any function which take multiple arguments is actually represented as functions which take a single argument and return another function. If you are familiar with the concept, feel free to skip the next section.

Curried functions for functional programming novices

This can be a hard thing to wrap your head around if you've never encountered it before, so for people who're encountering this the first time I'm going to give an example in Javascript first.

```
// Your standard two argument function
function add(a, b) {
    return a + b;
}

console.log(add(1, 2)); // 3
```

You could also write this as:

```
function add(a) {
    return function (b) {
        return a + b;
    }
}

console.log(add(1)(2)); // 3
```

Why would you want to write a function like this? The reason is that it lets you provide the arguments at different times to each other.

```
function runOnUsersData(username) {
    const data = fetchData(username); // Some big expensive operat

    return function(f) {
        return f(data);
    }
}

function average(executor) {
    const sum = executor((data) => sum(data));
    const count = executor((data) => count(data));
    return sum / count;
}

function bobsAverage() {
    const executor = runOnUsersData("bob");
    average(executor);
}
```

In this simplified case, you could have just loaded data into a variable, but the advantage here is this interface is the same regardless of whether you can pull all the data into a data structure or if you're shipping the functions themselves off to run on a hadoop job or lambda function or some other novel setup. By using this style, you can change these kind of details and the average function doesn't need to change.

Regrouping

Now that the functional and non-functional audiences are back together, how do you use function currying in *Nix*, not javascript? Since this is a much more standard practice in Nix, the syntax for this is very minimal.

```
adder = amountToAdd: x: amountToAdd + x
```

This is effectively a function which takes an argument `amountToAdd`, then returns the function `x: amountToAdd + x`

Three final pieces of Syntax

Ok, you're nearly done with this crash course in Nixlang. There's three final commonly used pieces of syntax that need to be covered, and then you'll be able to start writing your own nix expressions.

Let

So far you'll notice that I've never actually explained how to set a variable. While I've used the `x = "abc"` syntax in the REPL, this is a piece of REPL specific usage. As mentioned earlier, even a Nix program is a single expression, so there's no sequential code in your own nix programs to assign a variable on one line and use it again in another. But what if you have some value you want to use multiple times?

In that case you can use a let expression to create what is called a binding.

A basic let expression looks as follows:

```
# "abcabc"
```

```
let x = "abc"; in x + x;

# "abcxyz"
let x = "abc"; y = "xyz"; in x + y;
```

Notice that as in set definitions, the semicolon at the end is required here - it's a terminator, not a seperator. This is also not an assignment like let expressions in languages like Rust or Javascript. The values are only available for the single expression after the let keyword.

With

The next expression is also a way to introduce new values in scope. This is the `with` expression. This takes all the fields of a set and makes them available as bindings within the expression following the with. It looks like this:

```
with <some-set>; <expression>
```

Here's an example

```
let longNameSet = {
    foo = 1;
    bar = 2;
}; in ( # Brackets optional, added for clarity
    with longNameSet; {
        sum = foo + bar;
    }
)
```

This produces the set `{ sum = 3; }`. It could also have been written as:

```
let longNameSet = {
    foo = 1;
    bar = 2;
}; in {
    sum = longNameSet.foo + longNameSet.bar;
}
```

This is most like the Javascript `with` expression, which does basically the same thing. I'm still out on if I like this syntax as used here, my gut feeling is to dislike it for all the same reasons I dislike `with` in Javascript or `import *` in various languages, but `with` is everywhere in real Nix code, so you'll need to know it.

Inherit

`inherit` is used to copy values from an outer scope into a set being constructed. An `inherit` is used to copy values from an outer scope into a set being constructed. An example is below:

```
let someVariableWithALongName = 5; in {
    inherit someVariableWithALongName;
}
```

This produces the set `{ someVariableWithALongName = 5; }`. This is because it's equivalent to

```
let someVariableWithALongName = 5; in {
    someVariableWithALongName = someVariableWithALongName;
}
```

With this basic grounding of the Nix language features, next time I'm going to discuss a final fundamental structure to tie the language and package manager together.