70.8.5 How do I convert a string to UTF-8?

If you're mixing UTF-8 and non-UTF-8 strings, you might find it necessary to upgrade one of the strings to UTF-8. If you've got an SV, the easiest way to do this is:

```
sv_utf8_upgrade(sv);
```

However, you must not do this, for example:

```
if (!SvUTF8(left))
    sv_utf8_upgrade(left);
```

If you do this in a binary operator, you will actually change one of the strings that came into the operator, and, while it shouldn't be noticeable by the end user, it can cause problems.

Instead, `bytes_to_utf8` will give you a UTF-8-encoded **copy** of its string argument. This is useful for having the data available for comparisons and so on, without harming the original SV. There's also `utf8_to_bytes` to go the other way, but naturally, this will fail if the string contains any characters above 255 that can't be represented in a single byte.

70.8.6 Is there anything else I need to know?

Not really. Just remember these things:

- There's no way to tell if a string is UTF-8 or not. You can tell if an SV is UTF-8 by looking at is `SvUTF8` flag. Don't forget to set the flag if something should be UTF-8. Treat the flag as part of the PV, even though it's not - if you pass on the PV to somewhere, pass on the flag too.
- If a string is UTF-8, **always** use `utf8_to_uv` to get at the value, unless `UTF8_IS_INVARIANT(*s)` in which case you can use `*s`.
- When writing a character `uv` to a UTF-8 string, **always** use `uv_to_utf8`, unless `UTF8_IS_INVARIANT(uv))` in which case you can use `*s = uv`.
- Mixing UTF-8 and non-UTF-8 strings is tricky. Use `bytes_to_utf8` to get a new string which is UTF-8 encoded. There are tricks you can use to delay deciding whether you need to use a UTF-8 string until you get to a high character - `HALF_UPGRADE` is one of those.

70.9 Custom Operators

Custom operator support is a new experimental feature that allows you to define your own ops. This is primarily to allow the building of interpreters for other languages in the Perl core, but it also allows optimizations through the creation of "macro-ops" (ops which perform the functions of multiple ops which are usually executed together, such as `gvsv, gvsv, add`.)

This feature is implemented as a new op type, `OP_CUSTOM`. The Perl core does not "know" anything special about this op type, and so it will not be involved in any optimizations. This also means that you can define your custom ops to be any op structure - unary, binary, list and so on - you like.

It's important to know what custom operators won't do for you. They won't let you add new syntax to Perl, directly. They won't even let you add new keywords, directly. In fact, they won't change the way Perl compiles a program at all. You have to do those changes yourself, after Perl has compiled the program. You do this either by manipulating the op tree using a `CHECK` block and the `B::Generate` module, or by adding a custom peephole optimizer with the `optimize` module.

When you do this, you replace ordinary Perl ops with custom ops by creating ops with the type `OP_CUSTOM` and the `pp_addr` of your own PP function. This should be defined in XS code, and should look like the PP ops in `pp_*.c`. You are responsible for ensuring that your op takes the appropriate number of values from the stack, and you are responsible for adding stack marks if necessary.