1 Two access models

After doing lots of tests with LUATEX and LUAJITTEX with and without just in time compilation enabled, and with and without using ffi, we came to the conclusion that userdata prevents a speedup. We also found that the checking of metatables as well as assignment comes with overhead that can't be neglected. This is normally not really a problem but when processing fonts for more complex scripts it could have quite some overhead.

Because the userdata approach has some benefits, this remains the recommended way to access nodes. We did several experiments with faster access using this model, but eventually settled for the 'direct' approach. For code that is proven to be okay, one can use this access model that operates on nodes more directly.

Deep down in TEX a node has a number which is an entry in a memory table. In fact, this model, where TEX manages memory is real fast and one of the reasons why plugging in callbacks that operate on nodes is quite fast. No matter what future memory model LUATEX has, an internal reference will always be a simple data type (like a number or light userdata in LUA speak). So, if you use the direct model, even if you know that you currently deal with numbers, you should not depend on that property but treat is an abstraction just like traditional nodes. In fact, the fact that we use a simple basic datatype has the penalty that less checking can be done, but less checking is also the reason why it's somewhat faster. An important aspect is that one cannot mix both methods, but you can cast both models.

So our advice is: use the indexed approach when possible and investigate the direct one when speed might be an issue. For that reason we also provide the `get*` and `set*` functions in the top level node namespace. There is a limited set of getters. When implementing this direct approach the regular index by key variant was also optimized, so direct access only make sense when we're accessing nodes millions of times (which happens in some font processing for instance).

We're talking mostly of getters because setters are less important. Documents have not that many content related nodes and setting many thousands of properties is hardly a burden contrary to millions of consultations.

Normally you will access nodes like this:

```
local next = current.next
if next then
    -- do something
end
```

Here `next` is not a real field, but a virtual one. Accessing it results in a metatable method being called. In practice it boils down to looking up the node type and based on the node type checking for the field name. In a worst case you have a node type that sits at the end of the lookup list and a field that is last in the lookup chain. However, in successive versions of LUATEX these lookups have been optimized and the most frequently accessed nodes and fields hav ea higher priority.

Because in practice the `next` accessor result sin a function call, there is some overhead involved. The next code does the same and performs a tiny bit faster (but not that much because it is still a function call but one that knows what to look up).

```
local next = node.next(current)
```

```
if next then
    -- do something
end
```

There are several such function based accessors now:

`getnext`	parsing nodelist always involves this one
`getprev`	used less but is logical companion to getnext
`getid`	consulted a lot
`getsubtype`	consulted less but also a topper
`getfont`	used a lot in otf handling (glyph nodes are consulted a lot)
`getchar`	idem and also in other places
`getlist`	we often parse nested lists so this is a convenient one too (only works for hlist and vlist!)
`getleader`	comparable to list, seldom used in tex (but needs frequent consulting like lists; leaders could
`getfield`	generic getter, sufficient for the rest (other field names are often shared so a specific getter

It doesn't make sense to add more. Profiling demonstrated that these fields can get accesses way more times than other fields. Even in complex documents, many node and fields types never get seen, or seen only a few times. Most function in the `node` namespace have a compation in `node.direct`, but of course not the ones that don't deal with nodes themselves. The following table summarized this:

function	**node**	**direct**
`copy`	+	+
`copy_list`	+	+
`count`	+	+
`current_attr`	+	+
`dimensions`	+	+
`do_ligature_n`	+	+
`end_of_math`	+	+
`family_font`	+	−
`fields`	+	−
`first_character`	+	−
`first_glyph`	+	+
`flush_list`	+	+
`flush_node`	+	+
`free`	+	+
`getbox`	−	+
`getchar`	+	+
`getfield`	+	+
`getfont`	+	+
`getid`	+	+
`getnext`	+	+
`getprev`	+	+
`getlist`	+	+
`getleader`	+	+
`getsubtype`	+	+
`has_glyph`	+	+
`has_attribute`	+	+

has_field	+	+
hpack	+	+
id	+	−
insert_after	+	+
insert_before	+	+
is_direct	−	+
is_node	+	+
kerning	+	−
last_node	+	+
length	+	+
ligaturing	+	−
mlist_to_hlist	+	−
new	+	+
next	+	−
prev	+	−
tostring	+	+
protect_glyphs	+	+
protrusion_skippable	+	+
remove	+	+
set_attribute	+	+
setbox	+	+
setfield	+	+
slide	+	+
subtype	+	−
tail	+	+
todirect	+	+
tonode	+	+
traverse	+	+
traverse_id	+	+
type	+	−
types	+	−
unprotect_glyphs	+	+
unset_attribute	+	+
usedlist	+	+
vpack	+	+
whatsits	+	−
write	+	+

The `node.next` and `node.prev` functions will stay but for consistency there are variants called `getnext` and `getprev`. We had to use `get` because `node.id` and `node.subtype` are already taken for providing meta information about nodes.