CodeMark Annotation

CodeMark annotations are an attempt to provide an easily maintainable type check and documentation system for Lua application development. The main idea, and not a new one, is that documentation needs to be included close to its relevant implementation. The documentation can then be extracted from implementations to create summaries of the exported elements of a file and project wide repositories of reference material. This material can be used by integrated development environments to provide information to developers and maintainers when and where it is most pertinent.

One example of a development environment, and the focus of a particular CodeMark implementation is the ZeroBrane Studio , ZBS. Another is Visual Studio Code , VSC. CodeMark provides support for both environments. The CodeMark package generates type definitions, function signatures, and tooltips for ZBS and VSC from a common set of concise code annotations. VSC support includes provision for type checking by the Lua Language Server, LLS. The annotations are used to generate summary documentation in both Markdown and HTML formats. There's a copy facility so that comments and definitions can be supplied once and reapplied with relevent local changes wherever appropriate.

CodeMark deals with more than function signatures and type definitions. Descriptions of Command Line Interfaces (CLIs) are used to generate simple text help files. CLI descriptions can be documented in their implementations and copied to where they're used.

CodeMark has been used in a modestly sized project, MUSE, as a test of its utility. That project batches calls on the ZBS, VSC, and summary generators to provide all the documentation for the project as a whole. As mentioned, this includes help file generation. It also provides for documentation of the auxilliary files generally found as part of a project.

What follows is a description of the CodeMark annotations. There are two elements of the CodeMark system: *mark* and *reference* annotations. First up are the mark annotations.

File Marks

File marks on a line of the file identify the *Kind* of a CodeMark annotated file it is and carry information needed to produce output for that kind. There are three kinds of file marks, distinguished by a leading *Mark*.

API annotated files define a module (which may contain multiple libraries). While summary file output is generated in *Markdown* format for each of the three kinds of files here, API annotated files also create and update the project wide repository mentioned earlier. This can be used by ZBS and VSC/LLS, for example, to generate completion hints and function descriptions available during editing.

LIST annotated files are the auxiliary files of a project. CodeMark operations just output a summary of annotations for such files.

HELP annotated files generate *help* text for programs (which generally provide Command Line Interfaces, CLIs). For some applications, the help file for a program needs to be in a particular place associated with a project. The *Sign* field for HELP annotated files (see below) specifies this place.

Here are some examples of the three kinds for file marks:

Kind	Mark	Sign	End	Text	End	Out
API	`--:!`	`{map: []: (:) }`	<-	**CLI Library for** `map`	->	`muse/docs/lib/map.md`
LIST	`--:~`	`":"`	<-	**Chart 40 Block Farm**	->	`muse/docs/charts/farm40.md`
HELP	`--:?`	`rom/help/amuse.txt`	<-	**Prepare a Turtle**	->	`muse/docs/amuse.md`

Each of these three kinds of CodeMark annotated files supply *Sign*, *Text*, and *Out* fields in their file marks. The *Out* field specifies where to put a Markdown summary of the interfaces exposed by the file. It indicates the project name that the file is part of as the first part of that specification. Summary files and LLS libraries will be in the directory in the project file tree found with that project name. The *Text* field is just a short description of what the file is for. The *Sign* field shows CodeMark type annotation for the return values of the execution of API and LIST files.

(As an aside, the API example above indicates that the `map` module's return value is a table indexed by `map` which is in turn a dictionary of functions. All to be described later.)

Reference Marks

Reference marks provide the bulk of CodeMark annotations in a file. They are included in the summary file output for the CodeMark annotated file and may either generate a help file or make changes to the project repository, a file where project wide information is kept .

LIB marks for a module create project repository entries with descriptions of the module's libraries. They provide information needed to produce a LLS signature file for that library. There is one such mark for a module.

FACE marks document the library, name, and arguments of a library function, that is, its (inter)face, as its *Sign*. (signature) They document what the element returns as *Out* as well as a description of the element as *Text*. The library and name of the element are separated by a dot (as shown in the example below). The *Sign* and *Out* fields are used as a key for code completion and type check information in the project repository. The *Text* field is also retrieved for inclusion as part of ZBS and VSC tool tips. As a special case, if the library name for the (internal) function begins with an underscore, the mark generates an entry in the project repository for type checking but is not included in the summary file output. The *Sign* and *Out* fields use CodeMark type annotation syntax described later.

TYPE marks provide for descriptions and completion hints for variables expected to be bound to Lua types that may or may not be specific to any one library.

CLI marks describe a command line interface implemented by an associated function. They put the first word of their *Sign* (stripped of formatting) in the project wide repository as a key for retrieval of its *Text*. The key is reformatted ("hidden") in such a way so that it does not trigger ZBS or VSC completion hints.

COPY marks retrieve text by using the stripped and reformatted COPY mark's *SIGN* as a key into the repository. The COPY mark's *Text*, if any, replaces the retrieved *Sign* in the retrieved *Text*. The replaced text is included in the output file and is also used to produce the help file if one has been specified. (The same mechanism is used to retrieve FACE marks.) COPY marks allow descriptive text to be used where needed even though captured near the relevant implementation. The hope is that text near the implementation is more likely to be updated when the implementation is changed.

Examples of the reference marks:

Kind	Mark	Sign		Text		Out
LIB	`--:|`	`map`	:	*Adds orientation view*	->	`map`
FACE	`--::`	`map.get(name: ":", key: ":")`	->	*Get feature value*	->	`any | false`
TYPE	`--:>`	`mineplan`	:	*For ore*	->	`{even: plan, odd: plan}`
CLI	`--:-`	`l[ook]`	->	*Detect and inspect*		
COPY	`--:=`	`l[ook ]`	:	look		

Comment Marks

Comment marks are included in the summary output file as is. There are two kinds of comment marks. The first word of the *TEXT* of a **WORD** mark is stripped of formatting and then reformatted to provide a (hidden) key into the project repository for the mark's *TEXT*. As described previously, that text can then be referenced by COPY marks. The *TEXT* is included in the summary file for the module. The other kind of comment mark, **HEAD**, just puts the heading's *TEXT* in the summary.

MORE marks extend the descriptions of a preceding TYPE, FACE, or LIB mark. Each successive MORE appends its *TEXT* to the text of the project repository entries for those marks. This is complete when the next mark is not MORE.

Kind	Mark	Text
WORD	--:<	**Places - Points, Locations, Trails, and Ranges of Maps**
HEAD	--:#	**Operations on places (points, trails, and ranges)**

Kind	Mark	Text
MORE	--:+	*Try to move to position, dig to unblock if needed, catch and raise error.*

We can look at an example of a CLI definition together with its documentation and run time hint support:

```
local function erase(name)
--:- erase name -> Remove named place, broadcast Muse eXcise (MX).
--:# Referenced through `map.op` for CLI dispatch
---------implementation---------
end; map.hints["erase "] = {["?place"] = {}}
```

The CLI itself is in another file in the `programs` directory:

```
print(core.string(map.op({"erase", ...}))) --:= erase:
```

The implementation and its documentation are kept together and just referenced where applied.

Type Annotations: Introduction

CodeMark annotations are more concise than those of LLS and more descriptive. But they do map to LLS annotations. The simple types such as numbers, `#:`, strings, `":"`, and booleans, `^:` map directly. Lua also allows for "userdata" types, `@:`, perhaps written in C, available to the Lua runtime. The boolean values `true` and `false` as well as `nil` are represented as is. It's sometimes useful to be able to represent something that may be any type, `any`. Further, it is common in Lua code to ignore some return values as well as some elements of a table. This is indicated by the dummy type, `_:` treated as `any`.

Annotations may be grouped with parentheses, `()`. They may be combined as a union of types as `|`. They may be marked with a `?` suffix to indicate that a value may be `nil`.

Functions, are annotated as `(:)` or `():`. The first case types a value as a `function` with no other information provided. The second allows for named and typed parameters as well as typed (and optionally named) return values. Here's a simple example:

`--:: move.track(enable: ^:)` -> *Set tracking condition* -> `enable: ^:`

Here's another with multiple returns and additional descriptive text (included in the summary and VSC tool tip):

`--::place.erase(name: ":")` -> *Removes named place from array of places.* -> `#:, index: #:`
--:+ *Return new length of places table and the (previous) index of the removed place.*

As mentioned, Lua provides a userdata type. The following function takes a file handle argument:

`--:: core.record(handle: @:, message: ":", filename: ":")` -> *Appends (status) message to file on player* -> `nil`

CodeMark annotations can document functions' raising and catching exceptions:

`--:: farm.plant(planting: ":")` -> *Tills and plants found planting.* -> `report: ":" &!`

It's just a way to document functions which raise an exception (`&!`), catch one (`&:`), or, perhaps, catch and raise one (`&: &!`). Lua also provides for raising an exception that includes a table, for example `&!recovery` where recovery is a TYPE of table (described below).

Function definitions can be copied. This turns out to be useful in avoiding repetitive boilerplate (and so simplifying maintenance). Here's a common definition:

`--:: move.moves(count: #:?)` -> *Count 0: just turn, 1: default* -> `"done", remaining: #:, xyzf, direction &!recovery`

And here's how it's used:

```
function move.left(count) count = count or 1; return turn(turnLeft, count, "left", "move") end --:= move.moves:: move.left:
```

There are ten of these sharing the common definition and generating type information specific to each of the functions all the same and slightly different. (As a detail, function definitions are in a separate CodeMark name space for copy operations. The extra colon `move.moves::` above references the function name space.)

Type Annotations: Tables and Subtypes of Tables

Tables are a critical building block for Lua scripts. A Lua table can be an *array*, a *tuple*, a *dictionary*, or what LLS calls *table literals*. (The last are accessed using the dot notation as seen in the function names above.) Each has its own annotation. A table can also be typed as just a table with no other information as `{:}`. Here's an example of a TYPE annotation describing an array of strings:

`--:> plan.path:` *array of space separated character sequence strings describing path* -> `":"[]`

CodeMark provides for naming parts of tuple and dictionaries. This information is stripped away for LLS:

`--:> xyz:` *Minecraft coordinates: +x: east, +y: up, +z: south* -> `:[x: #:, y: #:, z: #:]`

`--:> features:` *Dictionary of string key, any value pairs* -> `[key: ":"]: any`

Specific kinds of tables can be thought of as being a new type which is a subtype of tables as in the examples above. Here's another example. The following function has one argument, `targets`, an array of strings. It returns a `detail` or `nil`. A `detail` is a table accessed by table literals: `name`, `count`, and `damage`. Each of these is typed and described.

```
--:: core.findItems(targets: ":"[]) -> Selects found slot. -> detail?
--:> detail: Defined by Computercraft -> {name: detail.name, count: detail.count, damage: detail.damage}
--:> detail.name: Prepended by the mod name "minecraft:". -> ":"
```

`--:> detail.count:` *Available in inventory* -> `#: --:> detail.damage:` *Distinguishing value* -> `#:`

The subtype information, once provided, can be used throughout the project:

`--:: turtle.check(targets: ":"[], :detail:)` -> *Tries to match each target against* `detail.name`. -> `matched: ^:`

This function takes a table with some number of string elements and a `:detail:` table. As a convenience, when both the type and parameter name is the same, the name can be surrounded by colons (as, for example, `:detail:`) rather than specifying something like `detail:detail`.

As promised, types (and return values of functions) can be unions. Here's one that is either an array of `xyz` elements or a dictionary keyed by `core.faces` (a type defined elsewhere) whose vaules are `xyz` elements.

`--:> xyzMap:` *Table of vectors either an array or dictionary* -> `xyz[] | [core.faces]: xyz`

Type declarations may, of course, be build up from other declared types:

`--:: core.vectorPairs(start: bounds, addend: xyz, number: #:, partial: bounds?)` -> *Make plots.* -> `bounds[]`
`--:> bounds:` *Vector pair defining a rectangular solid* -> `:[xyz, xyz]`

The function takes a `bounds` table for its `start`, an addend, `addend: xyz:` (which, as indicated, is an `xyz` table), a `number`, `#:`, and an additional optional `bounds` table. It returns an array of `bounds`.

Making Marks

CodeMark provides the programs to generate project documentation and LLS library files:

- `apiFiles` generates markdown summaries and the repository (used by ZBS and `signfiles`)
- `signFiles` generates LLS libraries, one for each source module, from the repository
- `downFiles` generates HTML from summary files and the source modules themselves for each source module

The `apiFiles` and `downFiles` programs are front ends for `apiMark` and `downMark` respectively. They are both used by the `opFiles` program to navigate a project directory for batch operations. Support is also provided by `apiMark` as a ZBS plug-in operating on the currently edited file with a symlink in the ZBS `packages` directory. ZBS code completions access the repository file through an appropriately named symlink in the ZBS `api/lua` directory.

All the CodeMark operations are invoked by `Mark` in the `CodeMark` directory:

```
Mark(apiDirectory, apiFile, sourceDirectories, docsDirectories, codeDirectories, verbose)
```

The `apiDirectory` is the path to the directory for the repository and the LLS libraries (as a string). The `apiFile` is the filename (string) for the repository. The `sourceDirectories` argument is a table of strings, each entry the path to a directory of source modules. The `docsDirectories` and

`codeDirectories` arguments are tables of paths (as strings) to stipulate where the code summaries and the code itself as HTML are to be put. The `verbose` argument, a boolean, is `true` for more detail on the progress of `Mark` operations.

For windows, set the `LUA_PATH` environment variable for `require` calls to include `%LUA_PROJECTS%\CodeMark\?.lua;;` where `LUA_PROJECTS` is the directory where `CodeMark` can be found. In Windows, there's a GUI, `sysdm.cpl`, for that.

Conclusions

There's a lot of power in Lua's representational richness. It's a dynamically typed language with all the expressive freedom and opportunity for run time error that that entails. CodeMark provides type checking (through LLS) with concise, expressive annotations and a comprehensive documentation system for two IDEs (your choice) to ease maintenance in those environments.