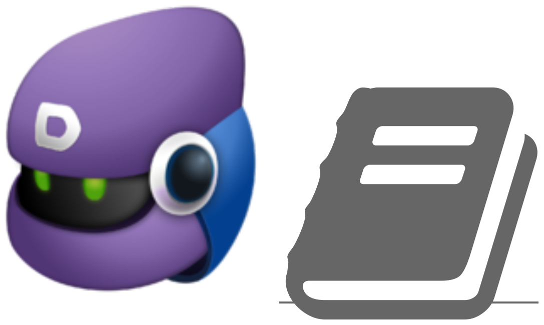
Mork Structure

Read content offline
Did you know that you can read content offline by using one of these tools? If you would like to read offline MDN content in another format, let us know by commenting on Bug 665750.



Terms

First let's introduce terms describing elements of Mork documents. Each term describes an entity used in Mork. Each type of entity has it's own style of markup in Mork.
Before we describe the markup, let's define each type of entity.
Each entry will include a short name in *italic*, a typical representative letter in **bold**, and a full name in *blue*.
Sometimes we prefer short names, using (eg) lit in preference to literal.

- *val* - (**V**) *value*: the value part of a cell; a *lit*eral or a *ref*erence
- *lit* - (**L**) *literal*: a binary octet sequence; e.g., a text value; e.g., a cell value in a row.
- *tab* - (**T**) *table*: a sparse matrix; a collection of *row*s; a garbage collection root.
- *row* - (**R**) *row*: a sequence of *cell*s; attributes of one object; a horizontal *tab*le slice.
- *col* - (**C**) *column*: a *lit*eral used to name an attribute; a vertical *tab*le slice.
- *cell* - (**.**) *cell*: a pair (*col val*) at the intersection of *col* and *row* in a *tab*le.
- *id* - (**I**) *identity*: a unique hex number in some namespace *scope*, naming a literal or object.
- *oid* - (**O**) *object identity*: a scoped unique name or *id* of an object or entity, large or small.
- *ref* - (**^**) *reference*: a *val*ue expressed using the *id* of an object or *lit*eral, instead of a *lit*eral.
- *dict* - (**D**) *dictionary*: a *map* of *lit*erals associated with assigned *id*s inside some *scope*.
- *atom* - (**A**) *atom*: a *lit*eral which is not used as a *col*umn name; e.g., a text value.
- *map* - (**M**) *hashmap*: a hash table mapping keys to values (e.g. ids to objects).
- *scope* - (**S**) *scope*: a (colon suffixed) namespace within which *id*s are unique
- *group* - (**G**) *group*: a set of related changes applied *atomically*; a markup transaction.

Model

The basic Mork content model is a *table* (or synonymously, a sparse *matrix*) composed of *rows* containing *cells*, where each cell is a member of exactly one *column* (*col*). Each cell is one attribute in a row. The name of the attribute is a *lit*eral designating the *col*umn, and the content of the attribute is the *value*. The content *val*ue of a cell is either a literal (*lit*) or a reference (*ref*). Each ref points to a lit or row or table, so a a cell can "contain" another shared object by reference.

Mork content exists as mentioned, without need of forward references or declarations. By default, new content is considered additive to old content if something already exists due to earlier mention. Mork markup is update oriented, with syntax to edit existing objects, or objects newly mentioned. Mork updates can be gathered into *group*s of related edits that must be applied atomically (all edits or none) as a transaction. A typical Mork file might contain a sequence of groups in a log structured format. Any group that starts but does not end correctly must be ignored by a reader.

Roots

When copying an old Mork store to a freshly written new file, unused content in the old copy can be omitted. But what defines *unused*? Any or root or literal not reachable from at least one table is considered unused, and can be pruned or scavenged to reduce space consumption.

Any table is a garbage collection root making referenced content reachable for preservation. Tables are never collected. To reduce table space consumption, one can only remove all row members.

Naive Mork implementations must avoid a subtle refcounting bug that can arise due to content written in nearly random hash based order. Markup might say to remove a row from one table early on, then add this same row to another table later. In between, the refcount might be zero, but the row's content must not be destroyed eagerly. A zero refcount only means dead content after all markup has been applied.

Parser

You're expected to write a parser by hand. The grammar is only a clue, not a spec. The hard part is not the grammar, which is is simple and unambiguous. The hard part is the meaning of the Mork markup.

Start

In version 1.4, a Mork file begins with this on the first line:

```
// <!-- <mdb:mork:z v="1.4"/> -->
```

You can ignore this magic header because it's meaningless. All it does is announce the file format. Presumably later versions of Mork will have different values where 1.4 appears.
Even if you did not ignore the first line, this is a C++ style comment which Mork takes as equivalent to whitespace, hence meaningless.

```
* start ::= magic (items | group)+
```

Comments

Your Mork tokenizer should treat a C++ style comment -- starting with // and ending at line's end -- as a single whitespace byte.
Mork considers any combination of CR or LF (#xD or #xA) to be a line ending, taken singly or in pairs.

Space

Whitespace is optional in most places where Mork allows it to appear.
The grammar is apt to sprinkle space where it's likely to occur.

- space ::= (white)*
- white ::= (#xA | #xD | #x9 | #x20 | comment | sp)
- sp ::= /*other bytes you might wish to consider whitespace*/
- comment ::= #xA not-endline endline

Header

This entire section might be obsolete. The 1.4 source code doesn't seem to process the 1.1 grammar showing a file row. So perhaps this was dropped before version 1.4. The parser in morkParser.cpp appears to recognize @ only as part of group syntax.

After the first line, Mork typically begins with exactly one file row which is metainfo about the Mork file itself, expressed as a row. This special purpose row is its own gc root, so it doesn't need to be a member of a table to persist. It looks different from any other row because it surrounds the normal row syntax with @ bytes. Here's an example with a single cell inside:
@[(author=rys)]@

Rows are enclosed in [] square brackets. This row contains a single cell in column author with literal value rys. This example is arbitrary. Presumably the file row metainfo is a good place for apps to embed their own magic head info, without disturbing the other content of the file. If a file has more than one file row, or perhaps none, you might tolerate this to be lenient and permissive since no harm is done.

Content

After the magic first line and the distinguished file row, the rest of the content in a Mork file is a sequence of items: either objects (rows, dicts, tables) or updates.
Items are optionally collected into groups applied as atomic transactions.

- items ::= (object | update)*
- group ::= gather items (commit | abort)

The syntax for groups is most complex, so let's address this one first.

Groups

A Mork group is a related set of top level content applied all together in one atomic unit, either in its entirety or not at all, as a mechanism for handling transactions in a log structured format.
In version 1.4, @ is always followed by $$ as part of the group markup syntax.

- group ::= gather items (commit | abort)
- gather ::= space @$${ id {@
- commit ::= space @$$} id }@
- abort ::= space @$$}~abort~ id }@

In practice, a parser should expect to see $$ after @ is seen. Then { says a transaction group is starting, and *id* is the hex identity of this transaction. For example, suppose *id* is equal to FACE.
The same value for id must appear in *commit* or *abort* for an end-of-group to be recognized.
How should a parser go about ensuring that content through either *commit* or *abort* is applied atomically (all or none)?

Basically, a parser should remember the byte position immediately following *gather*, so it can reseek this position after locating the group end. Then it should scan forward looking for @$$}FACE}@ or for @$$}~abort~FACE}@. If the first is found, then the parser should seek the saved position after *gather* and parse all the content inside the group. Otherwise the group should be ignored.

If a parser sees a new group start before the old one ends, this should probably be seen as equivalent to an abort of the first group. This interpretation allows updates to be added on the end of stores that have previously aborted due to truncated file writes.

Note: the choice of group markup syntax seen here and literal escape metachar syntax shown later are related. Well-formed Mork documents should not be able to embed $$ inside a literal, because $ is a metachar which *only* escapes hex and not itself. This was done on purpose, so content inside a literal value would not accidentally terminate a transaction group, just because @$$ appears inside a literal. A well-formed literal would encode this sequence as @$\$ instead. This is the answer to jwz's question, about why there is more than one escape metachar:

because it helps avoid corruption of user data.

As a result, a parser can generally get along with one non-space character lookahead. Seeing @ means group markup is expected. Then a parser looks for $$ which can't appear inside literals when Mork writers follow the rules. This is *exactly* why $$ was chosen here.

Tokens

You were probably expecting to see material about *item*s here, right after *group*s. But here we switch tactics momentarily and take a bottom-up look at tokens and primitive data representation used to compose the higher level constructs used by items further below.

~~Your Mork tokenizer might want to see @[and]@ as special file row tokens~~. Most tokens are single bytes, or can be treated as if single bytes had been seen. Some complex multibyte tokens that are not *lit*erals have the same meaning as a single #x20 whitespace byte after having whatever parsing effect is needed.

For example, markup for indicating a *group* is equivalent to whitespace once it serves the purpose of clustering other content inside a group.

Note that a *line continuation*, consisting of a backslash \ followed by *line-end*, is not a token at all when it appears inside a *lit*eral. Instead, a tokenizer should ignore a line continuation as if it never appeared.

In addition, Mork uses the following set of single byte tokens:

- < - *open angle* - begins a dict (inside a dict, begins metainfo row)
- > - *close angle* - ends a dict
- [- *open bracket* - begins a row (inside a row, begins metainfo row)
-] - *close bracket* - ends a row
- { - *open brace* - begins a table (inside a table, begins metainfo row)
- } - *close brace* - ends a table
- (- *open paren* - begins a cell
-) - *close paren* - ends a cell
- ^ - *up arrow* - dereference following *id* for *lit*eral value
- r - *lower r* - dereference following *oid* for *row* (by *ref*) value
- t - *lower t* - dereference following *oid* for *tab*le (by *ref*) value
- : - *colon* - next value is *scope* namespace for preceding *id*
- = - *equals* - begin a *lit*eral value inside a cell
- + - *plus* - add update: insert content
- - - *minus* - cut update: remove content
- ! - *bang* - put update: clear and set content

Ids

Under Mork, object *id*entities are written in hex. So an *id* token is just a naked sequence of hex bytes, either upper or lowercase. Because the interpretation is an integer, case is not significant.

Some namespace is always understood as the *scope* for every *id*, but this does not appear in the *id* token itself. Whenever a *scope* is explicitly given, it appears after a colon following the *id*, as described next for *oid*s.

- id ::= hex+.
- hex ::= [0-9a-fA-F]

Oids

A Mork *oid* is an object *id*, and includes both the hex *id* and the namespace *scope*. When not explicitly stated, the scope is implicitly understood as some default for each context.

- oid ::= id | id:scope
- scope ::= literal | ^id | ^oid

Note the third option for *scope* might not be supported or used in practice, since it would make *oid* and *scope* recursive. You might expect to see ^id:literal and ^id:^id, but probably not ^id:^id:literal.

According to the Mork parser source code the actual syntax for the Oid is as follows:

- oid ::= id | id:scope
- scope ::= name | ^id
- name ::= [A-Za-z_:]+

Literals

A Mork *lit*eral is a binary octet sequence encoded as plain text. Every byte becomes literally the next byte of the value, unless certain bytes are seen which have metacharacter status. (Why is there more than one metachar? Because it might shrink markup. Complexity here is worth some compression.)

-) - *close paren* - end of literal
- $ - *dollar* - escape the next two hex bytes which encode a single octet
- \ - *backslash* - if the next byte is either #xA or #xD, omit linebreak from literal; otherwise escape next byte. For example, \ removes metachar status from an immediately following \, $, or).

The first metachar is *close paren*. A *lit*eral always appears at the tail end of a *cell* which is always terminated by a *close paren*), so in practice every *lit*eral is terminated by). The only way to get) inside a literal is by escaping the) byte one way or other.

The second metachar is *dollar* $, which allows you to encode any octet as two digits of hex. Some writers might encode all non-ascii octets this way, and the year 2000 version of Mozilla did this, but it's not required. You are never required to use $ to escape bytes when writing, but readers must escape hex following $ when $ itself is not escaped, say using \. (Why did I choose $ for this metachar? Because I thought URLs might be common Mork content, and I wanted to use a byte that might appear less often in URLs.)

The third metachar is *backslash* \, which was added to allow escaping metachars using C like syntax, and to allow line continuation in a C like manner so very long lines need not be generated unless a writer insists.

(If I was going to extend Mork for base 64, I'd probably extend the meaning of the $ metachar to understand a following non-hex byte was the start of a base 64 sequence. For example, ${ABC} might mean ABC was base 64 content. I've seen folks online bitch about Mozilla's verbose encoding of unicode under Mork using $ to encode every null byte. Why didn't you speak up during the year I discussed it online? In five years, why did no one tweak Mork so version 1.5 would do better? Why not just write unicode as raw binary, since Mork supports that? Why does Mork suck because no one spends an hour making changes? Whatever.)

Items

Okay, now we'll finally cover *items*, which is where all interesting Mork content is actually found.

- items ::= (object | update)*

- object ::= (dict | row | table)
- edit ::= (+ | - | !)
- update ::= edit (row | table)

(Note *dict* does not appear in *update* only because dicts have no identity, and thus can't be updated.)

When a parser does not see @ for a group at top level, it expects to see one of these first:

- + : the next object *adds* or *inserts* content listed to existing content.
- - : the next object *cuts* or *removes* content listed from existing content.
- ! : the next object *adds* content listed, after *clearing* or *deleting* old existing content.
- < : begin a *dict*.
- [: begin a *row*.
- { : begin a *table*.

Because following sections describe *dict*s, *row*s, and *table*s, this rest of this section says more about those three *edit* bytes. The + for *add* is really Mork's default state because it is implied when missing before a row or table. If a row or table already exists, then new content listed is simply added to whatever already exists. (Of course, adding a cell with the same column as an existing column will replace the old value.)

When a writer can see that rewriting an object from scratch is more space efficient that incremental adds or cuts, it can use ! to clear all old content and start afresh.

When a writer sees an incremental cut is cheaper than rebuilding a large object from scratch, it can use - before an object to indicate content *removal* instead of content *insertion*.

Dicts

The purpose of a *dict* is to enumerate instances of literals with associated ids. A parser is expected to populate a *map* (hashmap) associating each *id* key with each *lit* value, so later references to any given *id* inside cells is understood as a reference to the *lit* value.

A Mork implementation should have more than one *map* -- one for each *scope* is needed. But version 1.4 of Mork only uses two scopes: a for *atom* literals and c for *col*umn literals. The former, a, is the default *scope* for *ids*s in a *dict*, unless explicitly changed to the latter by a *metadict* containing a (atomScope=c) cell.

(Note: to simplify this grammar, optional *space* has been omitted before each literal token, except those inside *value*. We don't want optional space before *value* because it encourages space inefficienty, but you might choose to be tolerant here.)

- dict ::= < (metadict | alias)* >
- metadict ::= < (cell)* >
- alias ::= (id value)
- value ::= ^oid | =literal
- oid ::= id | id:scope

This grammar does not show a constraint on *id*, which should be a hex value not less than 80. All *id*s less than 80 have reserved definitions: any single ascii byte *lit*eral with ascii value 0xNN is defined to have *id* NN. Mork implementations which assign *id*s dynamically should use ascending values from 80 (or from the greatest *id* already assigned).

In principle, *metadict* might contain cells of any type. But in Mork version 1.4, only cells equal to (atomScope=a) and (atomScope=c) have any meaning. These change the default *scope* for the *id* found in subsequent *alias* definitions. But instead of using (atomScope=a) in a metadict to return the default scope to a, you can simply use > < to close and reopen a new dict with fewer bytes, which also returns the default scope to a.

```
< <(atomScope=c)> (80=cards)(81=dn)(82=modifytimestamp)(83=cn)
(84=givenname)(85=mail)(86=xmozillausehtmlmail)(87=sn)>

<(90=cn=John Hackworth,mail=jhackworth@atlantis.com)(91=19981001014531Z)
(92=John Hackworth)(93=John)(94=jhackworth@atlantis.com)(95=FALSE)
(96=Hackworth)>
```

After the examples above have been parsed, the *oid* 85:c has value mail, and the *oid* 96:a has value Hackworth. Using ^ syntax to indicate an oid, these are usually written ^85 when column scope c is the default, and ^96 when atom scope a is the default.

Alias

Each *alias* inside a *dict* has syntax almost identical to that of *cell*s; they both use parens to delimit a pair. But the first element of the pair in an *alias* is an *id*, where a *cell* has a *col* in the first position. Additionally, a *cell* allows more variation in the second element of the pair, where an *alias* is constrained to a *lit*eral or an *oid* that references a *lit*eral.

- alias ::= (id value)
- value ::= ^oid | =literal

Optional *space* here is a bad idea for writers, because it decreases space efficiency. But readers might want to tolerate unexpected *space*.

Should you allow forward references to *id*s that have not yet been defined by an *alias*? Maybe, since it does little harm. It would be consistent with the system of making things exist as soon as they are mentioned. So an *oid* with no definition might refer to an empty value in general. This is what happens with *row*s and *table*s, and you might do this with *lit*eral *oid*s as well.

Rows

A *row* is a logically unordered sequence of *cell*s. Any physical ordering is not considered semantically meaningful. All that matters is which *col*umns appear in the *cell*s, and what *lit*erals are referenced by values.

(To simplify the grammar, optional *space* before tokens is not shown.)

- row ::= [roid (metarow | cell)*]
- roid ::= oid /*default scope is rowScope from table*/
- metarow ::= [(cell)*]
- cell ::= (col slot)
- col ::= ^oid /*default scope is c*/ | name
- slot ::= ^oid /*default scope is a*/ | =literal

In the *col* position, the *scope* of all *id*s defaults to c, and in the *slot* position, the *scope* defaults to a. The default *scope* for *roid* depends on context -- a containing *table* will supply the default. If the scope is itself an *oid* for a *lit*eral, the default scope is c; so when *roid* is 1:^80 this means 1:^80:c.

```
< <(atomScope=c)> (80=cards)(81=dn)(82=modifytimestamp)(83=cn)
(84=givenname)(85=mail)(86=xmozillausehtmlmail)(87=sn)>

<(90=cn=John Hackworth,mail=jhackworth@atlantis.com)(91=19981001014531Z)
```

```
(92=John Hackworth)(93=John)(94=jhackworth@atlantis.com)(95=FALSE)
(96=Hackworth)>

[1:^80 (^81^90)(^82^91)(^83^92)(^84^93)(^85^94)(^86^95)(^87^96)]
```

This row example was written entirely with *oid*s, which is not very human readable. So let's rewrite this row using all *lit*erals with identical resulting content. Note that all *col*s default to *scope* c and all *slot*s default to *scope* a.

```
[ 1:cards (dn=cn=John Hackworth,mail=jhackworth@atlantis.com)
(modifytimestamp=19981001014531Z)(cn=John Hackworth)(givenname=John)
(mail=jhackworth@atlantis.com)(^xmozillausehtmlmail=FALSE)(sn=Hackworth)]
```

This version of the row has identical content, but is much more readable, at the expense of duplicating strings in places when other rows have the same values, which is often the case.

Cells

A *cell* is basically a named value, where the name is a *col*umn *lit*eral constant. If you line up cells with the same *col* inside a table, the cells will form a column in the sparse table matrix.

- cell ::= (col slot)
- col ::= ^oid /*default scope is c*/ | name
- name ::= [a-zA-Z:_] [a-zA-Z:_+-?!]*
- slot ::= ^oid /*default scope is a*/ | =literal
- oid ::= id | id:scope

Note cells can also refer to a row or a table by *oid*, even if this feature is not actually used by specific Mork applications.

In practice, a reader might allow *col*umn *name*s to include any byte except ^, =, or). Notice that an empty literal value in column sn will be written like this: (sn=).

Tables

A *table* is a collection of *row*s. I can't recall whether they are ordered; my guess is no, since otherwise you'd plausibly be able to add a *twice* twice in different positions. I think each *table* is a *map* (hashmap) of *row*s, mapping *roid*s to *row*s, in which case member *row*s might appear in any order.

But I seem to recall the idea of using tables to represent the sorted ordering of other tables. So maybe tables are ordered, in which case an array representation is also a good idea. In any case, the Mork syntax for serialization is immune to such concerns.

- table ::= { toid (metatable | row | roid)* }
- toid ::= oid /*default scope is c*/
- metatable ::= { (cell)* }

```
{ 1:cards {(rowScope=cards)(tableKind=Johns)} 1 2 }
```

The *metatable* contains *cell*s of metainfo, but Mork only considers a few possible *col*umns interesting. Here we see the default *scope* for *roid* row *id*s has been set to cards using (rowScope=cards). This means the row *id*s 1 and 2 are understood the same as if they had been written 1:cards and 2:cards, or alternatively as 1:^80 and 2:^80 (assuming ^80 resolves to cards).

The role (or purpose or type) of the table is specified by (tableKind=Johns). This supports an examination of tables at runtime by *kind* instead of merely by *toid* table *oid*.

```
{ 1:^80 {(rowScope^80:c)(tableKind=Johns)} 1:^80 2:^80 } // with oids
```

Here the table has been rewritten using *lit*eral *oid*s instead of string *lit*erals. But this table is considered identity to the last, if *oid* 80:c has been defined as cards in some *dict*.

```
{ 1:^80 {(rowScope^80:c)(tableKind=Johns)} // with explicit row cells
  [ 1:^80 (^81^90)(^82^91)(^83^92)(^84^93)(^85^94)(^86^95)(^87^96)]
  [ 2 (mail=galtj@atlantis.com)(cn=John Galt)]
}
```

This last version of the same table actually defines contents of the member *row*s using explicit [] notation. You need only define a row in one place, and any other use of that same row can include the row as a member by reference, using only the row's *oid*. The Mork writer for Mozilla in 2000 made an effort to write the actual contents of any entity one time only, by keeping track of whether it was "clean" in the sense of having been written once already in the current printing task.

Identity

Each object has a unique identity called an *oid* (for *o*bject *id*entity), composed of two parts: *id* and *scope*.

Note *scope* is a short synonym for *namespace*. Please avoid confusing this use of the word namespace with any other. (There is *no* relation to the meaning of namespace under XML, except in similar function.)

Mork reverses conventional namespace order, with namespace coming last instead of first. Mork uses *id*:*scope* instead of *namespace*:*name*. But it's all the same.

The purpose of *scope* namespaces is to allow content from multiple apps to mix in a single Mork store without possibility of identity collision, as long as apps use different namespaces. Very long scope names to ensure uniqueness has no penalty because scopes can appear by id instead of by name.

Atoms

Mork is grounded in primitive interned-string octet sequence values, usually called *atom*s. (David Bienvenu prefers *atomized* strings to *interned* strings.) This terminology makes an assumption about Mork implementations: each value namespace scope should have a map hashing string keys to *atom* values, so every instance of the same string can be shared as the same *atom* instance. You needn't do this, but it's less confusing if you do. You also want a map from id to atom, to resolve *val* hex oid references.

Unicode

Mork doesn't know or care about unicode or wide characters. All Mork content is uninterpreted octet sequences, containing whatever an app likes. All content that looks like binary is escaped, so it won't interfere with Mork's markup, which is basically ascii (or the moral equivalent). Mork's simple syntax for framing embedded content does not constrain the type of content put inside. Mork does not escape binary content very efficiently, typically using three bytes to represent each awkward byte: zero appears in hex as $00. (The choice of $ as metacharacter was arbitrary, and picked to be less common within Mozilla embedded content. Space efficiency for binary was never high priority in whimsical demands from managment.)

Brevity

Why is there more than one namespace for octet string vals? Column names have their own namespace so the ids of columns will usually be no more than two hex digits in length, assuming an app does not have a huge number of distinct attribute names.

If cols and vals were in the same namespace, Mork files would likely be larger if common column names had ids that were longer. So this is a compression strategy, making the format more complex.

Bootstrap

Parts of Mork are self describing, making it reflective. This causes a bootstrap problem where metainfo would describe itself, and well known constants would be desired for scope names. For this reason, Mork defines the ids of single byte scope names specially: the integer value of a single octet name is defined to be the id of that name. This means all scope name ids less than 0x80 are already assigned. A Mork implementation must dynamically assign ids increasing from 0x80.

Default scope names for *col*'s and ordinary atom *val*'s are defined as follows:

- col - c (for col) with id 0x63 (equal to 'c')
- val - a (for atom) with id 0x61 (equal to 'a')

Objects

Mork models an app object with a *row*, where each object attribute is a separate cell in the row. The attribute name is the column. A collection of objects is a sparse matrix in the form of a table enumerating rows.

Mork also represents the attributes as metainfo as a row, so the anotation of a table or row with metainfo appears as another row, distinguished by privileged syntax. Primary parts of metainfo include object identity, kind or type, and default scope for contained objects.

Original Document Information

- Author(s): David McCusker
- Original Location: http://www.erys.org/resume/netscape/mork/mork.html - returns 404 Not Found (<a class='"external"' href='"http://web.archive.org/web/20050324235032/http://erys.org/resume/netscape/mork/mork.html"'>parent page on archive.org</a>)

Tags (2)

- Developing Mozilla
- Mork

Contributors to this page: AlexVas, CyberShadow, Squadron76, trevorh, George3, Andreas Wuest
Last updated by: trevorh, Oct 29, 2012 3:18:35 PM
Last reviewed by: trevorh, Oct 29, 2012 3:18:35 PM