CM
The SML/NJ Compilation and Library Manager

(for SML/NJ version 110.40 and later)

User Manual

Matthias Blume
Lucent Technologies, Bell Labs

May 21, 2002

Contents

1 Introduction

This manual describes a new implementation of CM, the “Compilation and Library Manager” for Standard ML of New Jersey (SML/NJ). Like its previous incarnation, CM is in charge of managing separate compilation and facilitates access to stable libraries.

Most programming projects that use CM are composed of separate *libraries*. Libraries are collections of ML compilation units. These collections themselves can be internally sub-structured using CM’s notion of *library components*.

CM offers the following features to the programmer:

- separate compilation and type-safe linking [AM94]
- hierarchical modularity [BA99]
- automatic dependency analysis [Blu99]
- optimization of the compilation process via *cutoff*-recompilation techniques [ATW94]
- management of program libraries, distinguishing between libraries that are *under development* and libraries that are *stable*
- operating-system independent file naming (with optional escape to native file names)
- adaptability to changing environments using the *path anchor* facility
- an extensible set of auxiliary *tools* that lets CM seemlessly interoperate with other program generators, source-control systems, literate-programming facilities, or shell-scripts
- checked version numbers on libraries
- (still rudimentary) support for access control
- access to libraries from the interactive toplevel loop
- sharing of code that is common to several programs or code that is common to both user programs and SML/NJ itself
- management of (the sharing of) link-time state
- conditional compilation (at compilation unit granularity)
- support for parallel and distributed compilation
- facilities for generating stand-alone ML programs
- a mechanism for deriving dependency information for use by other compilation managers (e.g., Unix’ **make**)
- an API to access all these facilities at SML/NJ’s interactive prompt or from user programs

CM puts emphasis on *working with libraries*. This contrasts with previous compilation managers for SML/NJ where the focus was on compilation management while libraries were added as an afterthought.

2 The CM model

2.1 Basic philosophy

The venerable **make** of Unix [Fel79] is *target-oriented*: one starts with a main goal (target) and applies production rules (with associated actions such as invoking a compiler) until no more rules are applicable. The leaves of the resulting derivation tree[1] can be seen as defining the set of source files that are used to make the main target.

CM, on the other hand, is largely *source-oriented*: Whereas with **make** one specifies the tree and lets the program derive the leaves, with CM one specifies the leaves and lets the program derive the tree. Thus, the programmer writes down a list of sources, and CM will calculate and then execute a series of steps to make the corresponding program. In **make** terminology, this resulting program acts as the "main goal", but under CM it does not need to be explicitly named. In fact, since there typically is no corresponding single file system object for it, a "natural" name does not even exist.

For simple projects it is literally true that all the programmer has to do is tell CM about the set of sources: a description file lists little more than the names of participating ML source files. However, larger projects typically benefit from a hierarchical structuring. This can be achieved by grouping ML source files into separate libraries and library components. Dependencies between such libraries have to be specified explicitly and must form an acyclic directed graph (DAG).

CM's own semantics, particularly its dependency analysis, interact with the ML language in such a way that for any well-formed project there will be exactly one possible interpretation as far as static semantics are concerned. Only well-formed projects are accepted by CM; projects that are not well-formed will cause error messages. (Well-formedness is *defined* to enforce a unique definition-use relation for ML definitions [Blu99].)

2.2 Description files

Technically, a CM library is a (possibly empty) collection of ML source files and may also contain references to other libraries. Each library comes with an export interface which specifies the set of all toplevel-defined symbols of the library that shall be exported to its clients (see section 4). A library is described by the contents of its *description file*.[2]

Example:

```
Library
    signature BAR
    structure Foo
is
    bar.sig
    foo.sml
    helper.sml

    $/basis.cm
    $/smlnj-lib.cm
```

This library exports two definitions, one for a structure named `Foo` and one for a signature named `BAR`. The specification for such exports appear between the keywords `Library` and `is`. The *members* of the library are specified after the keyword `is`. Here we have three ML source files (`bar.sig`, `foo.sml`, and `helper.sml`) as well as references to two external libraries (`$/basis.cm` and `$/smlnj-lib.cm`). The entry `$/basis.cm` typically denotes the description file for the *Standard ML Basis Library* [RG99]; most programs will want to list it in their own description file(s). The other library in this example (`$/smlnj-lib.cm`) is a library of data structures and algorithms that comes bundled with SML/NJ.

[1] "Tree" is figurative speech here since the derivation really yields a DAG.

[2] The description file may also contain references to input files for *tools* like `ml-lex` or `ml-yacc` that produce ML source files. See section 7.

2.3 Invoking CM

Once a library has been set up as shown in the example above, one can load it into a running interactive session by invoking function `CM.make`. If the name of the library's description file is, say, `fb.cm`, then one would type

```
CM.make "fb.cm";
```

at SML/NJ's interactive prompt. This will cause CM to

1. parse the description file `fb.cm`,
2. locate all its sources and all its sub-libraries,
3. calculate the dependency graph,
4. issue warnings and errors (and skip the remaining steps) if necessary,
5. compile those sources for which that is required,
6. execute module initialization code,
7. and augment the toplevel environment with bindings for exported symbols, i.e., in our example for `signature BAR` and `structure Foo`.

CM does not compile sources that are not "reachable" from the library's exports. For every other source, it will avoid recompilation if all of the following is true:

- The *binfile* for the source exists.
- The binfile has the same time stamp as the source.
- The current compilation environment for the source is precisely the same as the compilation environment that was in effect when the binfile was produced.

2.4 Members of a library

Members of a library do not have to be listed in any particular order since CM will automatically calculate the dependency graph. Some minor restrictions on the source language are necessary to make this work:

1. All top-level definitions must be *module* definitions (structures, signatures, functors, or functor signatures). In other words, there can be no top-level type-, value-, or infix-definitions.
2. For a given symbol, there can be at most one ML source file per library (or—more correctly—one file per library component; see Section 2.6) that defines the symbol at top level.
3. If more than one of the listed libraries or components is exporting the same symbol, then the definition (i.e., the ML source file that actually defines the symbol) must be identical in all cases.
4. The use of ML's **open** construct is not permitted at the top level of ML files compiled by CM. (The use is still ok at the interactive top level.)

Note that these rules do not require the exports of imported libraries to be distinct from the exports of ML source files in the current library. If an ML source file f re-defines a name n that is also imported from library l, then the disambiguating rule is that the definition from f takes precedence over that from l in all sources except f itself. Free occurences of n in f refer to l's definition. This rule makes it possible to easily write code for exporting an "augmented" version of some module. Example:

```
structure A = struct (* defines augmented A *)
    open A           (* refers to imported A *)
    fun f x = B.f x + C.g (x + 1)
end
```

Rule 3 may come as a bit of a surprise considering that each ML source file can be a member of at most one library (see section 2.7). However, it is indeed possible for two libraries to (re-)export the "same" definition provided they both import that definition from a third library. For example, let us assume that `a.cm` exports a structure `X` which was defined in `x.sml`—one of `a.cm`'s members. Now, if both `b.cm` and `c.cm` re-export that same structure `X` after importing it from `a.cm`, it is legal for a fourth library `d.cm` to import from both `b.cm` and `c.cm`.

The full syntax for library description files also includes provisions for a simple "conditional compilation" facility (see Section 8), for access control (see Section 9), and it accepts ML-style nestable comments delimited by `(*` and `*)`.

2.5 Name visibility

In general, all definitions exported from members (i.e., ML source files, sublibraries, and components) of a library are visible in all other ML source files of that library. The source code in those source files can refer to them directly without further qualification. Here, "exported" means either a top-level definition within an ML source file or a definition listed in a sublibrary's export list.

If a library is structured into library components using *groups* (see Section 2.6), then—as far as name visibility is concerned—each component (group) is treated like a separate library.

Cyclic dependencies among libraries, library components, or ML source files within a library are detected and flagged as errors.

2.6 Library components (groups)

CM's group model eliminates a whole class of potential naming problems by providing control over name spaces for program linkage. The group model in full generality sometimes requires bindings to be renamed at the time of import. As has been described separately [BA99], in the case of ML this can also be achieved using "administative" libaries, which is why CM can get away with not providing more direct support for renaming.

However, under CM, the term "library" does not only mean namespace management (as it would from the point of view of the pure group model) but also refers to actual file system objects (e.g., CM description files and stable library files). It would be inconvenient if name resolution problems would result in a proliferation of additional library files. Therefore, CM also provides the notion of library components ("groups"). Name resolution for groups works like name resolution for entire libraries, but grouping is entirely internal to each library.

When a library is *stabilized* (via `CM.stabilize` – see Section 2.8), the entire library is compiled to a single file (hence groups do not result in separate stable files).

During development, each group has its own description file which will be referred to by the surrounding library or by other groups of that library. The syntax of group description files is the same as that of library description files with the following exceptions:

- The initial keyword `Library` is replaced with `Group`. It is followed by the name of the surrounding library's description file in parentheses.
- The export list can be left empty, in which case CM will provide a default export list: all exports from ML source files plus all exports from subcomponents of the component. from other libraries will not be re-exported in this case.[3] (Notice that an export list that is not *syntactically* empty but which effectively contains zero symbols because of conditional compilation—see Section 8—does not count as being "left empty" in the above sense. Instead, the result would be an almost certainly useless component with truly no exports.)

[3]This default can be spelled out as `source(-) group(-)`. See section 4.

- There are some small restrictions on access control specifications (see Section 9).

As an example, let us assume that `foo-utils.cm` contains the following text:

```
Group (foo-lib.cm)
is
    set-util.sml
    map-util.sml
    $/basis.cm
```

This description defines group `foo-utils.cm` to have the following properties:

- it is a component of library `foo-lib.cm` (meaning that only foo-lib.cm itself or other groups thereof may list `foo-utils.cm` as one of their members)
- `set-utils.sml` and `map-util.sml` are ML source files belonging to this component
- exports from the Standard Basis Library are available when compiling these ML source files
- since the export list has been left blank, the only (implicitly specified) exports of this component are the top-level definitions in its ML source files

With this, the library description file `foo-lib.cm` could list `foo-utils.cm` as one of its members:

```
Library
    signature FOO
    structure Foo
is
    foo.sig
    foo.sml
    foo-utils.cm
    $/basis.cm
```

No harm is done if `foo-lib.cm` does not actually mention `foo-utils.cm`. In this case it could be that `foo-utils.cm` is mentioned indirectly via a chain of other components of `foo-lib.cm`. The other possibility is that it is not mentioned at all (in which case CM would never know about it, so it cannot complain).

2.7 Multiple occurences of the same member

The following rules apply to multiple occurences of the same ML source file, the same library, or the same group within a program:

- Within the same description file, each member can be specified at most once.
- Libraries can be referred to freely from as many other groups or libraries as the programmer desires.
- A group cannot be used from outside the uniquely defined library (as specified in its description file) of which it is a component. However, within that library it can be referred to from arbitrarily many other groups.
- The same ML source file cannot appear more than once. If an ML source file is to be referred to by multiple clients, it must first be “wrapped” into a library (or—if all references are from within the same library—a group).

2.8 Stable libraries

CM distinguishes between libraries that are *under development* and libraries that are *stable*. A stable library is created by a call of `CM.stabilize` (see Section 5.1.1).

Access to stable libraries is subject to less internal consistency-checking and touches far fewer file-system objects. Therefore, it is typically more efficient. Stable libraries play an additional semantic role in the context of access control (see Section 9).

From the client program's point of view, using a stable library is completely transparent. When referring to a library—regardless of whether it is under development or stable—one *always* uses the name of the library's description file. CM will check whether there is a stable version of the library and provided that is the case use it. This means that in the presence of a stable version, the library's actual description file does not have to physically exists (even though its name is used by CM to find the corresponding stable file).

2.9 Top-level groups

Mainly to facilitate some superficial backward-compatibility, CM also allows groups to appear at top level, i.e., outside of any library. Such groups must omit the parenthetical library specification and then cannot also be used within libraries. One could think of the top level itself as a "virtual unnamed library" whose components are these top-level groups.

3 Naming objects in the file system

3.1 Motivation

The main difficulty with file naming lies in the fact that files or even whole directories may move after CM has already partially (but not fully) processed them. For example, this happens when the *autoloader* (see Section 5.2) has been invoked and the session (including CM's internal state) is then frozen (i.e., saved to a file) via `SMLofNJ.exportML`.

CM's configurable *path anchor* mechanism enables it to resume such a session even when operating in a different environment, perhaps on a different machine with different file systems mounted, or a different location of the SML/NJ installation. Evaluation of path anchors always takes place as late as possible, and CM will re-evaluate path anchors as this becomes necessary due to changes to their configuration.

3.2 Basic rules

CM uses its own "standard" syntax for pathnames which for the most part happens to be the same as the one used by most Unix-like systems:

- Path name components are separated by "**/**".
- Special components "**.**" and "**..**" denote *current* and *previous* directory, respectively.
- Paths beginning with "**/**" are considered *absolute*.
- Other paths are *relative* unless they start with "**$**".

There is an important third form of standard paths: *anchored* paths. Anchored paths always start with "**$**".

Since this standard syntax does not cover system-specific aspects such as volume names, it is also possible to revert to "native" syntax by enclosing a path name in double-quotes. Of course, description files that use path names in native syntax are not portable across operating systems.

Absolute pathnames are resolved in the usual manner specific to the operating system. However, it is advisable to avoid absolute pathnames because they are certain to "break" if the corresponding file moves to a different location.

Relative pathnames that occur in some CM description file whose name is *path*/*file*`.cm` will be resolved relative to *path*, i.e., relative to the directory that contains the description file.

Relative pathnames that have been entered interactively, usually as an argument to one of CM's interface functions, will be resolved in the OS-specific manner, i.e., relative to the current working directory. However, notice that some of CM's operations (see section 5.2—autoload) will be executed lazily and, thus, can occur interleaved with arbitary other operations—including changes of the working directory. This is handled by CM in such a way that it appears as if all path derived from an interactive relative path p had been completely resolved at the time p was entered. As a result, two names specified using identical strings but at different times when different working directories were in effect will be kept apart and continue to refer to their respective original file system locations.

Anchored paths consist of an anchor name (of non-zero length) and a non-empty list of additional arcs. The name is enclosed by the path's leading **$** on the left and the path's first **/** on the right. The list of arcs follows the first **/**. As with all standard paths, the arcs themselves are also separated by **/**. An error is signalled if the anchor name is not known to CM. If a is a know anchor name currently bound to some directory name d, then the standard path `$`a`/`p (where p is a list of arcs) refers to d`/`p. The frequently occuring case where a coincides with the first arc of p can be abbreviated as `$/`p.

3.3 Anchor environments

Anchor names are resolved in the *anchor environment* that is in effect at the time the anchor is read.

The basis for all anchor environments is the *root environment*. Conceptually, the root environments is a fixed mapping that binds every possible anchor to a mutable location. The location can store a native directory name or can be marked "undefined". Most locations initially start out undefined. The contents of each location is configurable (see Section 3.4).

At the time a CM description file a`.cm` refers to another library's or library component's description file b`.cm`, it can augment the current anchor environment with new bindings. The new bindings are in effect while b`.cm` (including any description files *it* mentions!) is being processed. If a new binding binds an anchor name that was already bound in the current environment[4], then the old binding is being hidden. The effect is scoping for anchor names.

Using CM's *tool parameter* mechanism (see Section 7.1), a new binding is specified as a pair of anchor name and anchor value. The value has the form of another path name (standard or native). Example:

```
a.cm (bind:(anchor:lib value:$mystuff/a-lib)
      bind:(anchor:support value:$lib)
      bind:(anchor:utils value:/home/bob/stuff/ML/utils))
```

As shown in this example, it is perfectly legal for the specification of the value to involve the use of another anchor. That anchor will be resolved in the original anchor environment. Thus, a path anchored at `$lib` in `a.cm` will be resolved using the binding for `$mystuff` that is currently in effect. The point here is that a re-configuration of the root environment that affects `$mystuff` now also affects how `$lib` is resolved as it occurs within `a.cm`.

CM processes the list of `bind`-directives "in parallel." This means that the anchor `$support` will refer to the original meaning of `$lib` and is *not* being bound to `$mystuff/a-lib/asupport`.

The example also demonstrates that `value`-paths can be single anchors. In other words, the restriction that there has to be at least one arc after the anchor does not apply here. This makes it possible to "rename" anchors, or, to put it more precisely, for one anchor name to be established as an "alias" for another anchor name.

[4] which is technically always the case given our explanation of the root environment

3.4 Anchor configuration

Anchor configuration is concerned with the values that are stored in the root anchor environment. At startup time, the root environment is initialized by reading two configuration files: an installation-specific one and a user-specific one. After that, the contents of root locations can be maintained using CM's interface functions `CM.Anchor.anchor` and `CM.Anchor.reset` (see Section 5.1.4).

Although there is a hard-wired default for the installation-specific configuration file[5], this default in rarely being used. Instead, in a typical installation of SML/NJ the default will be a file *r*`/lib/pathconfig` where *r* is the *root* directory into which SML/NJ had been installed. (The installation procedure establishes this new default by setting the environment variable `CM_PATHCONFIG` at the time it produces the heap image for the interactive system.) The user can specify a new location at startup time using the same environment variable `CM_PATHCONFIG`.

The default location of the user-specific configuration file is `.smlnj-pathconfig` in the user's home directory (which must be given by the `HOME` environment variable). At startup time, this default can be overridden by a fixed location which must be given as the value of the environment variable `CM_LOCAL_PATHCONFIG`.

The syntax of all configuration files is identical. Lines are processed from top to bottom. White space divides lines into tokens.

- A line with exactly two tokens associates an anchor (the first token) with a directory in native syntax (the second token). Neither anchor nor directory name may contain white space and the anchor should not contain a **/**. If the directory name is a relative name, then it will be expanded by prepending the name of the directory that contains the configuration file.
- A line containing exactly one token that is the name of an anchor cancels any existing association of that anchor with a directory.
- A line with a single token that consists of a single minus sign **-** cancels all existing anchors. This typically makes sense only at the beginning of the user-specific configuration file and erases any settings that were made by the installation-specific configuration file.
- Lines with no token (i.e., empty lines) will be silently ignored.
- Any other line is considered malformed and will cause a warning but will otherwise be ignored.

3.5 When to use anchor environments

The following sample scenario for using anchor environments will provide some background on the reasons that led to the inclusion of this feature into CM. In short, anchor environments will typically be used to disambiguate between two or more different uses of the same anchor name. This is something that the root environment alone (i.e., a fixed mapping from anchors to directory names) cannot do because the root environment binds each anchor to at most one "meaning."

Suppose you are developing a project `P.cm` for which you obtain two utility libraries `A.cm` and `B.cm` from other programmers. Unfortunately, since these programmers did not coordinate their work (perhaps because they did not even know of each other), both `A.cm` and `B.cm` each uses its own helper library that is expected to be found under the name `$H/H.cm`.

Your task is to put references to `A.cm` and `B.cm` into `P.cm` so that everything "works." Now, mentioning `A.cm` and `B.cm` is easy enough, but if you do so without special precautions, you arrive at a situation where both helpers end up being the same—no matter how you configure the binding for anchor `$H`.

If you have access to the description files for `A.cm` and `B.cm`, you could work around this problem by changing the reference to `$H` in one of them into something else. Remember, however, that in general you may not be able to do this because it could be that either you do not have permissions to change the description files, or you might not even have any description file that you could change because the two libraries were given to you in "stable" form.

The correct solution is to add `bind:`-directives where you *use* `A.cm` and `B.cm` in your own description file `P.cm`. For example, you could write there:

[5] which happens to be `/usr/lib/smlnj-pathconfig`

```
A.cm (bind:(anchor:H value:$AH))
B.cm (bind:(anchor:H value:$BH))
```

With this, the two uses of anchor `$H` will occur in different *local* anchor environments. In the example as shown, the first such local environment effectively renames `$H` into `$AH` while the second renames `$H` into `$BH`. Thus, you can put independent bindings for `$AH` and `$HB` into your root configuration, these independent bindings will propagate to the respective local environment, and the two uses of `$H` will be resolved differently—as was intended.

Another situation where anchor name clashes would definitely happen is when two different versions (i.e., development stages) of the same set of libraries are being used at the same time. In this case, one should let the libraries within each set refer to each other using anchored names and provide (different) bindings for these names using `bind:`-directives from within their respective clients.

A good way of encapsulating the construction of the required local anchor environment for a library is to create a *proxy* for it (see Section 4).

4 Export lists

In the simplest case, the export list is given as a sequence of SML symbol names such as shown in earlier examples. But for more complicated scenarios, CM provides a little "set calculus" for writing export lists. Formally, the export section specifies the union of a sequence of *symbol sets*. Each individual set in this sequence can be one of the following:

- a singleton set given by the name of its sole member
- the set difference of two other sets s_1 and s_2, written as s_1-s_2
- the intersection of two sets s_1 and s_2, written as s_1*s_2
- the union of sets $s_1, \ldots, s_n$ for $n \geq 0$, written as ($s_1 \ldots s_n$)
- the top-level defined symbols of one of the SML source files f that the given library or group consists of, written as `source(`f`)`
- the top-level defined symbols of all constituent SML source files that are not marked *local* (see section 7.1.1), written as `source(-)`
- the exported symbols of one of the subgroups g, written as `group(`g`)`
- the exported symbols of *all* subgroups, written as `group(-)`
- the exported symbols of another library l
  In most situations, the notation for this is `library(`l`)`. However, notice that unlike in the case of subgroups and SML source files, l may or may not be listed as one of the members in the current description. For the case that l is not a member, its elaboration—determining its set of exported symbols—may require *tool parameters* to be specified (see section 7.1). When tool parameters are necessary, they are given within an extra pair of parentheses following the name l. Example:

  ```
  library(foo/bar.cm (bind:(anchor:x value:y)))
  ```

  If l is a member of the current description, then CM will re-use the earlier elaboration result. Tool parameters within the export set specification are neither necessary nor permitted in this case.

Notes:

- Intersection has higher precedence than set difference.
- Conditional compilation constructs (see section 8) may be used within set union costructs—both the parenthesized variety and at top level.

Using the export list calculus, it is easy to set up "proxy" libraries. A proxy library A is a library with a single member which is another library B so that the export lists of A and B coincide. Proxy libraries are mainly used for the purpose of managing anchor names and anchor environments (see Section 3.3). Using the `library(`l`)` construct one can avoid the cumbersome repetition of lengthy explicit export lists and, thus, improve maintainability.

5 Using CM

5.1 Structure CM

Functions that control CM's operation are accessible as members of a structure named `CM` which itself is exported from a library called `$smlnj/cm.cm` (or, alternatively, `$smlnj/cm/cm.cm`). This library is pre-registered for auto-loading at the interactive top level.

Other libraries can exploit CM's functionality simply by putting a `$smlnj/cm.cm` entry into their own description file. Section 16 shows one interesting use of this feature.

Here is a description of all members:

5.1.1 Compiling

Two main activities when using CM are to compile ML source code and to build stable libraries:

```
val recomp : string -> bool
val stabilize : bool -> string -> bool
```

`CM.recomp` takes the name of a program's "root" description file and compiles or recompiles all ML source files that are necessary to provide definitions for the root library's export list. (*Note:* The difference to `CM.make` is that no linking takes place.)

`CM.stabilize` takes a boolean flag and then the name of a library and *stabilizes* this library. A library is stabilized by writing all information pertaining to it, including all of its library components (i.e., subgroups), into a single file. Sublibraries do not become part of the stabilized library; CM records stub entries for them. When a stabilized library is used in other programs, all members of the library are guaranteed to be up-to-date; no dependency analysis work and no recompilation work will be necessary. If the boolean flag is `false`, then all sublibraries of the library must already be stable. If the flag is `true`, then CM will recursively stabilize all libraries reachable from the given root.

After a library has been stabilized it can be used even if none of its original sources—including the description file—are present.

The boolean result of `CM.recomp` and `CM.stabilize` indicates success or failure of the operation (`true` = success).

5.1.2 Linking and execution

In SML/NJ, linking means executing top-level code (i.e., module creation and initialization code) of each compilation unit. The resulting bindings can then be registered at the interactive top level.

```
val make : string -> bool
val autoload : string -> bool
```

`CM.make` first acts like `CM.recomp`. If the (re-)compilation is successful, then it proceeds by linking all modules that require linking. Provided there are no link-time errors, it finally introduces new bindings at top level.

During the course of the same `CM.make`, the code of each compilation module that is reachable from the root will be executed at most once. Code in units that are marked as *private* (see Section 5.3) will be executed exactly once. Code in other units will be executed only if the unit has been recompiled since it was executed last time or if it depends on another compilation unit whose code has been executed since.

In effect, different invocations of `CM.make` (and `CM.autoload`) will share dynamic state created at link time as much as possible unless the compilation units in question have been explicitly marked private.

`CM.autoload` acts like `CM.make`, only "lazily". See Section 5.2 for more information.

As before, the result of `CM.make` indicates success or failure of the operation. The result of `CM.autoload` indicates success or failure of the *registration*. (It does not know yet whether loading will actually succeed.)

5.1.3 Registers

Several internal registers control the operation of CM. A register of type T is accessible via a variable of type T `controller`, i.e., a pair of `get` and `set` functions.[6] Any invocation of the corresponding `get` function reads the current value of the register. An invocation of the `set` function replaces the current value with the argument given to `set`.

Controllers are members of `CM.Control`, a sub-structure of structure `CM`.

```
type 'a controller = { get: unit -> 'a, set: 'a -> unit }
structure Control : sig
  val verbose : bool controller
  val debug : bool controller
  val keep_going : bool controller
  val parse_caching : int controller
  val warn_obsolete : bool controller
  val conserve_memory : bool controller
  val generate_index : bool controller
end
```

`CM.Control.verbose` can be used to turn off CM's progress messages. The default is *true* and can be overriden at startup time by the environment variable `CM_VERBOSE`.

In the case of a compile-time error `CM.Contol.keep_going` instructs the `CM.recomp` phase to continue working on parts of the dependency graph that are not related to the error. (This does not work for outright syntax errors because a correct parse is needed before CM can construct the dependency graph.) The default is *false*, meaning "quit on first error", and can be overriden at startup by the environment variable `CM_KEEP_GOING`.

`CM.Control.parse_caching` sets a limit on how many parse trees are cached in main memory. In certain cases CM must parse source files in order to be able to calculate the dependency graph. Later, the same files may need to be compiled, in which case an existing parse tree saves the time to parse the file again. Keeping parse trees can be expensive in terms of memory usage. Moreover, CM makes special efforts to avoid re-parsing files in the first place unless they have actually been modified. Therefore, it may not make much sense to set this value very high. The default is *100* and can be overriden at startup time by the environment variable `CM_PARSE_CACHING`.

This version of CM uses an ML-inspired syntax for expressions in its conditional compilation subsystem (see Section 8). However, for the time being it will accept most of the original C-inspired expressions but produces a warning for each occurrence of an old-style operator. `CM.Control.warn_obsolete` can be used to turn these warnings off. The default is *true*, meaning "warnings are issued", and can be overriden at startup time by the environment variable `CM_WARN_OBSOLETE`.

`CM.Control.debug` can be used to turn on debug mode. This currently has the effect of dumping a trace of the master-slave protocol for parallel and distributed compilation (see Section 13) to TextIO.stdOut. The default is *false* and can be overriden at startup time by the environment variable `CM_DEBUG`.

Using `CM.Control.conserve_memory`, CM can be told to be slightly more conservative with its use of main memory at the expense of occasionally incurring additional input from stable library files. This does not save very much and, therefore, is normally turned off. The default (*false*) can be overridden at startup by the environment variable `CM_CONSERVE_MEMORY`.

`CM.Control.generate_index` is used to control the generation of human-readable *index files* (see section 11.2). The default setting is *false* and can be overridden at startup by the environment variable `CM_GENERATE_INDEX`.

[6] The type constructor `controller` is defined as part of `structure CM`.

5.1.4 Path anchors

Structure `CM` also provides functions to explicitly manipulate the path anchor configuration. These functions are members of structure `CM.Anchor`.

```
structure Anchor : sig
  val anchor : string -> string option controller
  val reset : unit -> unit
end
```

`CM.Anchor.anchor` returns a pair of `get` and `set` functions that can be used to query and modify the status of the named anchor. Note that the `get-set`-pair operates over type `string option`; a value of `NONE` means that the anchor is currently not bound (or, in the case of `set`, that it is being cancelled). The (optional) string given to `set` must be a directory name in native syntax (*without* trailing arc separator, e.g., / in Unix). If it is specified as a relative path name, then it will be expanded by prepending the name of the current working directory.

`CM.Anchor.reset` erases the entire existing path configuration. After a call of this function has completed, all root environment locations are marked as being "undefined".

5.1.5 Setting CM variables

CM variables are used by the conditional compilation system (see Section 8.1). Some of these variables are predefined, but the user can add new ones and alter or remove those that already exist.

```
val symval : string -> int option controller
```

Function `CM.symval` returns a `get-set`-pair for the symbol whose name string was specified as the argument. Note that the `get-set`-pair operates over type `int option`; a value of `NONE` means that the variable is not defined.

Examples:

```
#get (CM.symval "X") ();        (* query value of X *)
#set (CM.symval "Y") (SOME 1);  (* set Y to 1 *)
#set (CM.symval "Z") NONE;      (* remove definition for Z *)
```

Some care is necessary as `CM.symval` does not check whether the syntax of the argument string is valid. (However, the worst thing that could happen is that a variable defined via `CM.symval` is not accessible[7] because there is no legal syntax to name it.)

5.1.6 Library registry

To be able to share associated data structures such as symbol tables and dependency graphs, CM maintains an internal registry of all stable libraries that it has encountered during an ongoing interactive session. The `CM.Library` substructure of structure `CM` provides access to this registry.

```
structure Library : sig
  type lib
  val known : unit -> lib list
  val descr : lib -> string
  val osstring : lib -> string
  val dismiss : lib -> unit
  val unshare : lib -> unit
end
```

[7] from within CM's description files

`CM.Library.known`, when called, produces a list of currently known stable libraries. Each such library is represented by an element of the abstract data type `CM.Library.lib`.

`CM.Library.descr` extracts a string describing the location of the CM description file associated with the given library. The syntax of this string is almost the same as that being used by CM's master-slave protocol (see section 13.1).

`CM.Library.osstring` produces a string denoting the given library's description file using the underlying operating system's native pathname syntax. In other words, the result of a call of `CM.Library.osstring` is suitable as an argument to `TextIO.openIn`.

`CM.Library.dismiss` is used to remove a stable library from CM's internal registry. Although removing a library from the registry may recover considerable amounts of main memory, doing so also eliminates any chance of sharing the associated data structures with later references to the same library. Therefore, it is not always in the interest of memory-conscious users to use this feature.

While dependency graphs and symbol tables need to be reloaded when a previously dismissed library is referenced again, the sharing of link-time state created by this library is *not* affected. (Link-time state is independently maintained in a separate data structure. See the discussion of `CM.unshare` below.)

`CM.Library.unshare` is used to remove a stable library from CM's internal registry, and—at the same time—to inhibit future sharing with its existing link-time state. Any future references to this library will see newly created state (which will then be properly shared again). (**Warning:** *This feature is not the preferred way of creating unshared state; use functors for that. However, it can come in handy when two different (and perhaps incompatible) versions of the same library are supposed to coexist—especially if one of the two versions is used by SML/NJ itself. Normally, only programmers working on SML/NJ's compiler are expected to be using this facility.*)

5.1.7 Internal state

For CM to work correctly, it must maintain an up-to-date picture of the state of the surrounding world (as far as that state affects CM's operation). Most of the time, this happens automatically and should be transparent to the user. However, occasionally it may become necessary to intervene expliticly.

Access to CM's internal state is facilitated by members of the `CM.State` structure.

```
structure State : sig
  val pending : unit -> string list
  val synchronize : unit -> unit
  val reset : unit -> unit
end
```

`CM.State.pending` produces a list of strings, each string naming one of the symbols that are currently registered (i.e., "virtually bound") but not yet resolved by the autoloading mechanism.

`CM.State.synchronize` updates tables internal to CM to reflect changes in the file system. In particular, this will be necessary when the association of file names to "file IDs" (in Unix: inode numbers) changes during an ongoing session. In practice, the need for this tends to be rare.

`CM.State.reset` completely erases all internal state in CM. To do this is not very advisable since it will also break the association with pre-loaded libraries. It may be a useful tool for determining the amount of space taken up by the internal state, though.

5.1.8 Compile servers

On Unix-like systems, CM supports parallel compilation. For computers connected using a LAN, this can be extended to distributed compilation using a network file system and the operating system's "rsh" facility. For a detailed discussion, see Section 13.

Sub-structure `CM.Server` provides access to and manipulation of compile servers. Each attached server is represented by a value of type `CM.Server.server`.

```
structure Server : sig
  type server
  val start : { name: string,
                cmd: string * string list,
                pathtrans: (string -> string) option,
                pref: int } -> server option
  val stop : server -> unit
  val kill : server -> unit
  val name : server -> string
end
```

CM is put into "parallel" mode by attaching at least one compile server. Compile servers are attached using invocations of `CM.Server.start`. The function takes the name of the server (as an arbitrary string) (`name`), the Unix command used to start the server in a form suitable as an argument to `Unix.execute` (`cmd`), an optional "path transformation function" for converting local path names to remote pathnames (`pathtrans`), and a numeric "preference" value that is used to choose servers at times when more than one is idle (`pref`). The optional result is the handle representing the successfully attached server.

An existing server can be shut down and detached using `CM.Server.stop` or `CM.Server.kill`. The argument in either case must be the result of an earlier call of `CM.Server.start`. Function `CM.Server.stop` uses CM's master-slave protocol to instruct the server to shut down gracefully. Only if this fails it may become necessary to use `CM.Server.kill`, which will send a Unix TERM signal to destroy the server.

Given a server handle, function `CM.Server.name` returns the string that was originally given to the call of `CM.Server.start` used to created the server.

5.1.9 Plug-ins

As an alternative to `CM.make` or `CM.autoload`, where the main purpose is to subsequently be able to access the library from interactively entered code, one can instruct CM to load libraries "for effect".

```
val load_plugin : string -> bool
```

Function `CM.load_plugin` acts exactly like `CM.make` except that even in the case of success no new symbols will be bound in the interactive top-level environment. That means that link-time side-effects will be visible, but none of the exported definitions become available. This mechanism can be used for "plug-in" modules: a core library provides hooks where additional functionality can be registered later via side-effects; extensions to this core are implemented as additional libraries which, when loaded, register themselves with those hooks. By using `CM.load_plugin` instead of `CM.make`, one can avoid polluting the interactive top-level environment with spurious exports of the extension module.

CM itself uses plug-in modules in its member-class subsystem (see section 12). This makes it possible to add new classes and tools very easily without having to reconfigure or recompile CM, not to mention modify its source code.

5.1.10 Support for stand-alone programs

CM can be used to build stand-alone programs. In fact SML/NJ itself—including CM—is an example of this. (The interactive system cannot rely on an existing compilation manager when starting up.)

A stand-alone program is constructed by the runtime system from existing binfiles or members of existing stable libraries. CM must prepare those binfiles or libraries together with a list that describes them to the runtime system.

```
val mk_standalone : bool option ->
                    { project: string, wrapper: string, target: string } ->
                    string list option
```

Here, `project` and `wrapper` name description files and `target` is the name of a heap image—with or without the usual implicit heap image suffix; see the description of `SMLofNJ.exportFn` from the (SML/NJ-specific extension of the) Basis Library [RG99].

A call of `mk_standalone` triggers the following three-stage procedure:

1. Depending on the optional boolean argument, `project` is subjected to the equivalent of either `CM.recomp` or `CM.stabilize`. `NONE` means `CM.recomp`, and `(SOME` *r*`)` means `CM.stabilize` *r*. There are tree ways of how to continue from here:
   (a) If recompilation of `project` failed, then a result of `NONE` will be returned immediately.
   (b) If everything was up-to-date (i.e, if no ML source had to be compiled and all these sources were older than the existing `target`), then a result of `SOME []` will be returned.
   (c) Otherwise execution proceeds to the next stage.
2. The *wrapper library* named by `wrapper` is being recompiled (using the equivalent of `CM.recomp`). If this fails, `NONE` is returned. Otherwise execution proceeds to the next stage.
3. CM.mk_standalone constructs a topologically sorted list l of strings that, when written to a file, can be passed to the runtime system in order to perform stand-alone linkage of the program given by `wrapper`. The final result is `SOME` l.

The idea is that `project` names the library that actually implements the main program while `wrapper` names an auxiliary wrapper library responsible for issuing a call of `SMLofNJ.exportFn` (generating `target`) on behalf of `project`.

The programmer should normally never have a need to invoke `CM.mk_standalone` directly. Instead, this function is used by an auxiliary script called `ml-build` (see Section 15.1).

5.1.11 Finding all sources

The `CM.sources` function can be used to find the names of all source files that a given library depends on. It returns the names of all files involved with the exception of skeleton files and binfiles (see Section 11). Stable libraries are represented by their library file; their description file or consitutent members are *not* listed.

Normally, the function reports actual file names as used for accessing the file system. For (stable) library files this behavior can be inconvenient because these names depend on architecture and operating system. For this reason, `CM.sources` accepts an optional pair of strings that then will be used in place of the architecture- and OS-specific part of these names.

```
val sources :
  { arch: string, os: string } option ->
  string ->
  { file: string, class: string, derived: bool } list option
```

In case there was some error analyzing the specified library or group, `CM.sources` returns `NONE`. Otherwise the result is a list of records, each carrying a file name, the corresponding class, and information about whether or not the source was created by some tool.

Examples:

generating "make" dependencies: To generate dependency information usable by Unix' `make` command, one would be interested in all files that were not derived by some tool application. Moreover, one would probably like to use shell variables instead of concrete architecture- and OS-names:

```
Option.map (List.filter (not o #derived))
  (CM.sources (SOME { arch = "$ARCH", os = "$OPSYS" })
       "foo.cm");
```

A call of `CM.sources` similar to the one shown here is used by the auxiliary script `ml-makedepend` (see Section 15.2).

finding all `noweb` sources: To find all `noweb` sources (see Section 7.2.6), e.g., to be able to run the document preparation program `noweave` on them, one can simply look for entries of the `noweb` class. Here, one would probably want to include derived sources:

```
Option.map (List.filter (fn x => #class x = "noweb"))
  (CM.sources NONE "foo.cm");
```

5.2 The autoloader

From the user's point of view, a call of `CM.autoload` acts very much like the corresponding call of `CM.make` because the same bindings that `CM.make` would introduce into the top-level enviroment are also introduced by `CM.autoload`. However, most work will be deferred until some code that is entered later refers to one or more of these bindings. Only then will CM go and perform just the minimal work necessary to provide the actual definitions.

The autoloader plays a central role for the interactive system. Unlike in earlier versions, it cannot be turned off since it provides many of the standard pre-defined top-level bindings.

The autoloader is a convenient mechanism for virtually "loading" an entire library without incurring an undue increase in memory consumption for library modules that are not actually being used.

5.3 Sharing of state

Whenever it is legal to do so, CM lets multiple invocations of `CM.make` or `CM.autoload` share dynamic state created by link-time effects. Of course, sharing is not possible (and hence not "legal") if the compilation unit in question has recently been recompiled or depends on another compilation unit whose code has recently been re-executed. The programmer can explicitly mark certain ML files as *shared*, in which case CM will issue a warning whenever the unit's code has to be re-executed.

State created by compilation units marked as *private* is never shared across multiple calls to `CM.make` or `CM.autoload`. To understand this behavior it is useful to introduce the notion of a *traversal*. A traversal is the process of traversing the dependency graph on behalf of `CM.make` or `CM.autoload`. Several traversals can be executed interleaved with each other because a `CM.autoload` traversal normally stays suspended and is performed incrementally driven by input from the interactive top level loop.

As far as sharing is concerned, the rule is that during one traversal each compilation unit will be executed at most once. This means that the same "program" will not see multiple instantiations of the same compilation unit (where "program" refers to the code managed by one call of `CM.make` or `CM.autoload`). Each compilation unit will be linked at most once during a traversal and private state will not be confused with private state of other traversals that might be active at the same time.

5.3.1 Sharing annotations

ML source files in CM description files can be specified as being *private* or *shared*. This is done by adding a *tool parameter* specification for the file in the library- or group description file (see Section 7). To mark an ML file as *private*, follow the file name with the word `private` in parentheses. For *shared* ML files, replace `private` with `shared`.

An ML source file that is not annotated will typically be treated as *shared* unless it statically depends on some other *private* source. It is an error, checked by CM, for a *shared* source to depend on a *private* source.

5.3.2 Sharing with the interactive system

The SML/NJ interactive system, which includes the compiler, is itself created by linking modules from various libraries. Some of these libraries can also be used in user programs. Examples are the Standard ML Basis Library `$/basis.cm`, the SML/NJ library `$/smlnj-lib.cm`, and the ML-Yacc library `$/ml-yacc-lib.cm`.

If a module from a library is used by both the interactive system and a user program running under control of the interactive system, then CM will let them share code and dynamic state. Moreover, the affected portion of the library will never have to be "relinked".

6 Version numbers

A CM library can carry a version number. Version numbers are specified in parentheses after the keyword `Library` as non-empty dot-separated sequences of non-negative integers. Example:

```
Library (1.4.1.4.2.1.3.5)
    structure Sqrt2
is
    sqrt2.sml
```

6.1 How versions are compared

Version numbers are compared lexicographically, dot-separated component by dot-separated component, from left to right. The components themselves are compared numerically.

6.2 Version checking

An importing library or library component can specify which version of the imported library it would like to see. See the discussion is section 7.1.2 for how this is done. Where a version number is requested, an error is signalled if one of the following is true:

- the imported library does not carry a version number
- the imported library's version number is smaller than the one requested
- the imported library's version number has a first component (known as the "major" version number) that is greater than the one requested

A warning (but no error) is issued if the imported library has the same major version but the version as a whole is greater than the one requested.

Note: *Version numbers should be incremented on every change to a library. The major version number should be increased on every change that is not backward-compatible.*

7 Member classes and tools

Most members of groups and libraries are either plain ML files or other description files. However, it is possible to incorporate other types of files—as long as their contents can in some way be expanded into ML code or CM descriptions. The expansion is carried out by CM's *tools* facility.

CM maintains an internal registry of *classes* and associated *rules*. Each class represents the set of source files that its corresponding rule is applicable to. For example, the class `mlyacc` is responsible for files that contain input for the parser generator ML-Yacc [TA90]. The rule for `mlyacc` takes care of expanding an ML-Yacc specifications `foo.grm` by invoking the auxiliary program `ml-yacc`. The resulting ML files `foo.grm.sig` and `foo.grm.sml` are then used as if their names had directly been specified in place of `foo.grm`.

CM knows a small number of built-in classes. In many situations these classes will be sufficient, but in more complicated cases it may be worthwhile to add a new class. Since class rules are programmed in ML, adding a class is not as simple a matter as writing a rule for UNIX' `make` program [Fel79]. Of course, using ML has also advantages because it keeps CM extremely flexible in what rules can do. Moreover, it is not necessary to learn yet another "little language" in order to be able to program CM's tool facility.

When looking at the member of a description file, CM determines which tool to use by looking at clues like the file name suffix. However, it is also possible to specify the class of a member explicitly. For this, the member name is followed by a colon **:** and the name of the member class. All class names are case-insensitive.

In addition to genuine tool classes, there are four member classes that refer to facilities internal to CM:

`sml` is the class of ordinary ML source files.

`cm` is the class of CM library or group description files.

`tool` is the class of *plugin tools*. Its purpose is to trigger the loading of an auxiliary plugin module—usually with the purpose of extending the set of tool classes that CM understands. See section 12.3 for more information.

`suffix` is a class similar to `tool`. Its purpose is to declare additional filename suffixes and their associated classes. See section 12.3.

By default, CM automatically classifies files with a `.sml` suffix, a `.sig` suffix, or a `.fun` suffix as ML-source, file names ending in `.cm` as CM descriptions. Failure to classify a member will be reported as an error.

7.1 Tool parameters

In many cases the name of the member that caused a rule to be invoked is the only input to that rule. However, rules can be written in such a way that they take additional parameters. Those parameters, if present, must be specified in the CM description file between parentheses following the name of the member and the optional member class.

CM's core mechanism parses these tool options and breaks them up into a list of items, where each item is either a filename (i.e., *looks* like a filename) or a named list of sub-options. However, CM itself does not interpret the result but passes it on to the tool's rule function. It is in each rule's own responsibility to assign meaning to its options.

All named sub-option lists (for any class) are specified by a name string followed by a colon **:** and a parenthesized list of other tool options. If the list contains precisely one element, then its parentheses may be omitted.

7.1.1 Parameters for class `sml`

The `sml` class accepts four optional parameters. One is the *sharing annotation* that was explained earlier (see Section 5.3). The sharing annotation must be one of the two strings `shared` and `private`. If `shared` is specified, then dynamic state created by the compilation unit at link-time must be shared across invocations of `CM.make` or `CM.autoload`. The `private` annotation, on the other hand, means that dynamic state cannot be shared across such calls to `CM.make` or `CM.autoload`.

The second possible parameter for class `sml` is a sub-option list labeled `setup` and can be used to specify code that will be executed just before and just after the compiler is invoked for the ML source file. Code to be executed before compilation is labeled `pre`, code to be executed after compilation is complete is labeled `post`; either part is optional. Executable code itself is specified using strings that contain ML source text.

For example, if one wishes to disable warning messages for a specific source file `poorlywritten.sml` (but not for others), then one could write:

```
poorlywritten.sml (setup:(pre: "local open Compiler.Control\n\
                               \   in val w = !printWarnings before\n\
                               \              printWarnings := false\n\
                               \  end;"
                          post:"Compiler.Control.printWarnings := w;"))
```

Note that neither the pre- nor the post-section will be executed if the ML file does not need to be compiled.

The pre-section is compiled and executed in the current toplevel-environment while the post-section uses the toplevel-environment augmented with definitions from the pre-section. After the ML file has been compiled and the post-section (if present) has completed execution, definitions made by either section will be erased. This means that setup code for other files *cannot* refer to them, and neither can code that in the future might be entered at top level.

The third possible parameter for class `sml` is a sub-option labelled `lambdasplit`. It controls the cross-module inlining mechanism of SML/NJ.[8] The value of the option can either be a non-negative decimal integer or one of the following words: `default`, `on`, `off`, or `infinity`. The effect of this parameter also depends on a system-wide setting (accessible via structure `Compiler.Control.LambdaSplitting`). In the following table, the per-file `lambdasplit` parameter is shown at the top and the system-wide default is shown on the left side. Table entries show the combined effect of the two: -1 means "no inlinable exports from this file", ∞ means "as many inlinable exports as possible", and a non-negative numeric value specifies some intermediate "aggressiveness" of the splitting engine.

	`default`	`on`	`off`	`infinity`	n
`Off`	-1	-1	-1	-1	-1
`Default NONE`	-1	0	-1	∞	n
`Default (SOME` m`)`	m	m	-1	∞	n

Finally, the last possible parameter for class `sml` is the string `local`. A file marked `local` will be ignored when calculating the symbol set for an occurence of `source(-)` within the export list (see section 4).

7.1.2 Parameters for class `cm`

The `cm` class understands two kinds of parameters. The first is a named parameter labeled by the string `version`. It must have the format of a version number. CM will interpret this as a version request, thereby insuring that the imported library is not too old or too new. (See section 6 for more on this topic.)

Example:

```
euler.cm (version:2.71828)
pi.cm    (version:3.14159)
```

Normally, CM looks for stable library files in directory `CM/`*arch*−*os* (see section 11). However, if an explicit version has been requested, it will first try directory `CM/`*version*`/`*arch*−*os* before looking at the default location. This way it is possible to keep several versions of the same library in the file system.

However, CM normally does *not* permit the simultaneous use of multiple versions of the same library in one session. The disambiguating rule is that the version that gets loaded first "wins"; subsequent attempts to load different versions result in warnings or errors. (See the discussion of `CM.unshare` in section 5.1.6 for how to to circumvent this restriction.)

The second kind of parameter understood by `cm` is a named parameter labeled by the string `bind` (see Section 3.3). It can occur arbitrarily many times and each occurence must be a suboption-list of the form `(anchor:`a `value:`v`)`. The set of `bind`-parameters augments the current anchor environment to form the environment that is used while processing the contents of the named CM description file.

[8] The label is named after the technique ("λ-splitting" [BA97]) used to achieve the effect of cross-module inlining.

7.1.3 Parameters for classes `tool` and `suffix`

Class `tool` (see the discussion is section 12.3.2) does not accept any parameters.

Class `suffix` (see section 12.3.3) takes one mandatory parameter which is either simply a class name or the same class name labeled by `class`. Thus, the following two lines are equivalent:

```
ml : suffix (sml)
ml : suffix (class:sml)
```

There are no recognized filename suffixes for these two classes.

7.2 Built-in tools

7.2.1 ML-Yacc

The ML-Yacc tool is responsible for files that are input to the ML-Yacc parser generator. Its class name is `mlyacc`. Recognized file name suffixes are `.grm` and `.y`. For a source file f, the tool produces two targets f`.sig` and f`.sml`, both of which are always treated as ML source files. The `mlyacc` class accepts two optional tool parameters labeled `sigoptions` and `smloptions`. They specify tool options to be passed on to the generated `.sig`- and `.sml`-files, respectively. Example[9]:

```
  lang.grm (sigoptions:(setup:(pre:"print \"compiling lang.grm.sig\\n\";"))
            smloptions:(private))
```

The tool invokes the `ml-yacc` command if the targets are "outdated". A target is outdated if it is missing or older than the source. Unless anchored using the path anchor mechanism (see Section 3.3), the command `ml-yacc` will be located using the operating system's path search mechanism (e.g., the `$PATH` environment variable).

7.2.2 ML-Lex

The ML-Lex tool governs files that are input to the ML-Lex lexical analyzer generator [AMT89]. Its class name is `mllex`. Recognized file name suffixes are `.lex` and `.l`. For a source file f, the tool produces one targets f`.sml` which will always be treated as ML source code. Tool parameters are passed on without change to that file.

The tool invokes the `ml-lex` command if the target is outdated (just like in the case of ML-Yacc). Unless anchored using the path anchor mechanism (see Section 3.3), the command `ml-lex` will be located using the operating system's path search mechanism (e.g., the `$PATH` environment variable).

7.2.3 ML-Burg

The ML-Burg tool deals with files that are input to the ML-Burg code-generater generator [GG93]. Its class name is `mlburg`. The only recognized file name suffix is `.burg`. For a source file f`.burg`, the tool produces one targets f`.sml` which will always be treated as ML source code. Any tool parameters are passed on without change to the target.

The tool invokes the `ml-burg` command if the target is outdated. Unless anchored using the path anchor mechanism (see Section 3.3), the command `ml-lex` will be located using the operating system's path search mechanism (e.g., the `$PATH` environment variable).

[9] Since the generated `.sig`-file contains nothing more than an ML signature definition, it is typically not very useful to pass any options to it.

7.2.4 Shell

The Shell tool can be used to specify arbitrary shell commands to be invoked on behalf of a given file. The name of the class is `shell`. There are no recognized file name suffixes. This means that in order to use the shell tool one must always specify the `shell` member class explicitly.

The rule for the `shell` class relies on tool parameters. The parameter list must be given in parentheses and follow the `shell` class specification.

Consider the following example:

```
foo.pp : shell (target:foo.sml options:(shared)
                      /lib/cpp -P -Dbar=baz %s %t)
```

This member specification says that file `foo.sml` can be obtained from `foo.pp` by running it through the C preprocessor `cpp`. The fact that the target file is given as a tool parameter implies that the member itself is the source. The named parameter `options` lists the tool parameters to be used for that target. (In the example, the parentheses around `shared` are optional because it is the only element of the list.) The command line itself is given by the remaining non-keyword parameters. Here, a single **%s** is replaced by the source file name, and a single **%t** is replaced by the target file name; any other string beginning with **%** is shortened by its first character.

In the specification one can swap the positions of source and target (i.e., let the member name be the target) by using a `source` parameter:

```
foo.sml : shell (source:foo.pp options:shared
                       /lib/cpp -P -Dbar=baz %s %t)
```

Exactly one of the `source` and `target` parameters must be specified; the other one is taken to be the member name itself. The target class can be given by writing a `class` parameter whose single sub-option must be the desired class name.

The usual distinction between native and standard filename syntax applies to any given `source` or `target` parameter.

For example, if one were working on a Win32 system and the target file is supposed to be in the root directory on volume `D:`, then one must use native syntax to write it. One way of doing this would be:

```
"D:\\foo.sml" : shell (source : foo.pp options : shared
                             cpp -P -Dbar=baz %s %t)
```

As a result, `foo.sml` is interpreted using native syntax while `foo.pp` uses standard conventions (although in this case it does not make a difference). Had we used the `target` version from above, one would have to write:

```
foo.pp : shell (target : "D:\\foo.sml" options : shared
                               cpp -P -Dbar=baz %s %t)
```

The shell tool invokes its command whenever the target is outdated with respect to the source.

7.2.5 Make

The Make tool (class `make`) can (almost) be seen as a specialized version of the Shell tool. It has no source and one target (the member itself) which is always considered outdated. As with the Shell tool, it is possible to specify target class and parameters using the `class` and `options` keyword parameters.

The tool invokes the shell command `make` on the target. Unless anchored using the path anchor mechanism 3.3, the command will be located using the operating system's path search mechanism (e.g., the `$PATH` environment variable).

Any parameters other than the `class` and `options` specifications must be plain strings and are given as additional command line arguments to `make`. The target name is always the last command line argument.

Example:

```
bar-grm : make (class:mlyacc -f bar-grm.mk)
```

Here, file `bar-grm` is generated (and kept up-to-date) by invoking the command:

```
make -f bar-grm.mk bar-grm
```

The target file is then treated as input for `ml-yacc`.

Cascading Shell- and Make-tools is easily possible. Here is an example that first uses Make to build `bar.pp` and then filters the contens of `bar.pp` through the C preprocessor to arrive at `bar.sml`:

```
bar.pp : make (class:shell
                   options:(target:bar.sml cpp -Dbar=baz %s %t)
               -f bar-pp.mk)
```

7.2.6 Noweb

The `noweb` class handles sources written for Ramsey's *noweb* literate programming facility [Ram94]. Files ending with suffix `.nw` are automatically recognized as belonging to this class.

The list of targets that are to be extracted from a noweb file must be specified using tool options. A target can then have a variety of its own options. Each target is specified by a separate tool option labelled `target`. The option usually has the form of a sub-option list. Recognized sub-options are:

name the name of the target

root the (optional) root tag for the target (given to the `-R` command line switch for the `notangle` command); if `root` is missing, `name` is used instead

class the (optional) class of the target

options (optional) options for the tool that handles the target's class

lineformat a string that will be passed to the `-L` command line option of `notangle`

Example:

```
project.nw (target:(name:main.sml options:(private))
            target:(name:grammar class:mlyacc)
            target:(name:parse.sml))
```

In place of the sub-option list there can be a single string option which will be used for `name` or even an unnamed parameter (i.e., without the `target` label). If no targets are specified, the tool will assume two default targets by stripping the `.nw` suffix (if present) from the source name and adding `.sig` as well as `.sml`.

The following four examples are all equivalent:

```
foo.nw (target:(name:foo.sig) target:(name:foo.sml))
foo.nw (target:foo.sig target:foo.sml)
foo.nw (foo.sig foo.sml)
foo.nw
```

If `lineformat` is missing, then a default based on the target class is used. Currently only the `sml` and `cm` classes are known to CM; other classes can be added or removed by using the `NowebTool.lineNumbering` controller function exported from library `$/noweb-tool.cm`:

```
val lineNumbering: string -> { get: unit -> string option,
                               set: string option -> unit }
```

The `noweb` class accepts two other parameter besides `target`:

subdir specifies a sub-option that is used to specify a directory where derived files (i.e., target files and witness files as far as they have been specified using relative path names) are created. If the `subdir` option is missing, its value defaults to `NW`.

witness specifies an auxiliary derived file whose time stamp is used by CM to avoid recompiling extracted files whose contents have not changed. If `witness` has not been specified, then CM uses time stamps on extracted files directly to determine whether `notangle` needs to be run. Thus, with no witness, any change to the master file causes time stamps on all extracted files to be updated as well. If a witness was specified, then CM will write over extracted files, causing their time stamps to change, only if their contents have also changed. The `subdir` specification also applies to the name of the witness file.

Example:

```
foo.nw (subdir:NOWEBFILES
        witness:foo.wtn
        target:(name:main.sml))
```

Here, the files named `main.sml` and `foo.wtn` will be created as

```
NOWEBFILES/main.sml
NOWEBFILES/foo.wtn
```

while without the `subdir`-option it would have been

```
NW/main.sml
NW/foo.wtn
```

To avoid the creation of such a sub-directory, one can use the *current arc* “**.**” and write:

```
foo.nw (subdir:.
        witness:foo.wtn
        target:(name:main.sml))
```

7.2.7 Dir

Using the Dir tool one can use directory names in description files. There are two possible uses for the Dir tool:

1. Factoring out common directory names.
2. Scanning the contents of directories for files with ML code.

Directory factoring: The main purpose of the Dir tool (class `dir`) is to simplify CM descriptions that mention a large number of files all of which are located in in the same directory. For this style of usage, tool options have to be specified.

For example, writing

```
Group is
  long/directory/name : dir (a.sml b/c.sml d.sml)
```

is equivalent to the following verbose description:

```
Group is
  long/directory/name/a.sml
  long/directory/name/b/c.sml
  long/directory/name/d.sml
```

Since CM automatically classifies directory names as members of class `dir`, the example can be further simplified:

```
Group is
  long/directory/name (a.sml b/c.sml d.sml)
```

Options for class `dir` consist of a list of items, each item having one of two possible forms:

1. The item can be a sub-option list of the form `member:(`m `class:`c `options:`o`)` which emulates an ordinary member specification with class and options. The `class`- and `options`-fields may (independently of each other) be missing, but the order of fields that are present is fixed.
2. The item can be a simple name (as shown in the example). Such a simple name m is equivalent to the longer form `member:(`m`)`.

Members m must always be specified using *relative* path names.

For example, the description

```
Group is
  alpha/a.sml (private)
  alpha/b.ml : sml (shared)
  alpha/c.cm
  beta/d.sml
  beta/e.sml
```

can be simplified as:

```
Group is
  alpha (member:(a.sml options:private)
         member:(b.ml class:sml options:shared)
         c.cm)
  beta (d.sml e.sml)
```

Directory scanning: Another use of the Dir tool, indicated by the absence of tool options, is to include all ML code in a given directory "whole-sale style". For example, a member of the form

```
  projects/ml/foo : dir
```

lets CM scan the contents of directory `projects/ml/foo` and proceed as if a list of all discovered ML files had been written in place of the `dir` member. For this, the usual classification mechanism is used to decide which directory entries are to be considered files containing ML code.

As before, the example could be further simplified by omitting the class name. Thus, a very quick way of putting together a small project is to use a generic description file of the form:

```
Group is $/basis.cm .
```

As usual, the dot denotes the current directory. Therefore, CM will scan the current directory and include any ML code it finds there. (Library `$/basis.cm` is necessary for most non-trivial programs, so we included it also.)

This is deceptively simple, but be warned: The technique of letting CM scan the physical directory is to be avoided for any serious project because it is very fragile. It does not mix well with the use of other tools, it will break when certain otherwise unrelated ML files are present, and so on, and so forth. In short, for serious programming the Dir tool should not be used without specifying options.

8 Conditional compilation

In its description files, CM offers a simple conditional compilation facility inspired by the preprocessor for the C language [KR88]. However, it is not really a *pre*-processor, and the syntax of the controlling expressions is borrowed from SML.

Sequences of members can be guarded by `#if`-`#endif` brackets with optional `#elif` and `#else` lines in between. The same guarding syntax can also be used to conditionalize the export list. `#if`-, `#elif`-, `#else`-, and `#endif`-lines must start in the first column and always extend to the end of the current line. `#if` and `#elif` must be followed by a boolean expression.

Boolean expressions can be formed by comparing arithmetic expressions (using operators <, <=, =, >=, >, or <>), by logically combining two other boolean expressions (using operators `andalso`, `orelse`, =, or <>, by querying the existence of a CM symbol definition, or by querying the existence of an exported ML definition.

Arithmetic expressions can be numbers or references to CM symbols, or can be formed from other arithmetic expressions using operators +, − (subtraction), `*`, `div`, `mod`, or ˜ (unary minus). All arithmetic is done on signed integers.

Any expression (arithmetic or boolean) can be surrounded by parentheses to enforce precedence.

8.1 CM variables

CM provides a number of "variables" (names that stand for certain integers). These variables may appear in expressions of the conditional-compilation facility. The exact set of variables provided depends on SML/NJ version number, machine architecture, and operating system. A reference to a CM variable is considered an arithmetic expression. If the variable is not defined, then it evaluates to 0. The expression `defined`(v) is a boolean expression that yields true if and only if v is a defined CM variable.

The names of CM variables are formed starting with a letter followed by zero or more occurences of letters, decimal digits, apostrophes, or underscores.

The following variables will be defined and bound to 1:

- depending on the operating system:
  `OPSYS_UNIX`, `OPSYS_WIN32`, `OPSYS_MACOS`, `OPSYS_OS2`, or `OPSYS_BEOS`
- depending on processor architecture:
  `ARCH_SPARC`, `ARCH_ALPHA`, `ARCH_MIPS`, `ARCH_X86`, `ARCH_HPPA`, `ARCH_RS6000`, or `ARCH_PPC`
- depending on the processor's endianness: `BIG_ENDIAN` or `LITTLE_ENDIAN`
- depending on the native word size of the implementation: `SIZE_32` or `SIZE_64`
- the symbol `NEW_CM`

Furthermore, the symbol `SMLNJ_VERSION` will be bound to the major version number of SML/NJ (i.e., the number before the first dot) and `SMLNJ_MINOR_VERSION` will be bound to the system's minor version number (i.e., the number after the first dot).

Using the `CM.symval` interface one can define additional variables or modify existing ones.

8.2 Querying exported definitions

An expression of the form `defined`(n s), where s is an ML symbol and n is an ML namespace specifier, is a boolean expression that yields true if and only if any member included before this test exports a definition under this name. Therefore, order among members matters after all (but it remains unrelated to the problem of determining static dependencies)! The namespace specifier must be one of: `structure`, `signature`, `functor`, or `funsig`.

If the query takes place in the "exports" section of a description file, then it yields true if *any* of the included members exports the named symbol.

Example:

```
Library
    structure Foo
#if defined(structure Bar)
    structure Bar
#endif
is
#if SMLNJ_VERSION > 110
    new-foo.sml
#else
    old-foo.sml
#endif
#if defined(structure Bar)
    bar-client.sml
#else
    no-bar-so-far.sml
#endif
```

Here, the file `bar-client.sml` gets included if `SMLNJ_VERSION` is greater than 110 and `new-foo.sml` exports a structure `Bar` *or* if `SMLNJ_VERSION <= 110` and `old-foo.sml` exports structure `Bar`. Otherwise file `no-bar-so-far.sml` gets included instead. In addition, the export of structure `Bar` is guarded by its own existence. (Structure `Bar` could also be defined by `no-bar-so-far.sml` in which case it would get exported regardless of the outcome of the other `defined` test.)

8.3 Explicit errors

A pseudo-member of the form `#error ...`, which—like other #-items—starts in the first column and extends to the end of the line, causes an explicit error message to be printed unless it gets excluded by the conditional compilation logic. The error message is given by the remainder of the line after the word `error`.

9 Access control

The basic idea behind CM's access control is the following: In their description files, groups and libraries can specify a list of *privileges* that the client must have in order to be able to use them. Privileges at this level are just names (strings) and must be written in front of the initial keyword `Library` or `Group`. If one group or library imports from another group or library, then privileges (or rather: privilege requirements) are being inherited. In effect, to be able to use a program, one must have all privileges for all its libraries, sub-libraries and library components, components of sub-libraries, and so on.

Of course, this alone would not yet be satisfactory. The main service of the access control system is that it can let a client use an "unsafe" library "safely". For example, a library `LSafe.cm` could "wrap" all the unsafe operations in `LUnsafe.cm` with enough error checking that they become safe. Therefore, a user of `LSafe.cm` should not also be required to possess the privileges that would be required if one were to use `LUnsafe.cm` directly.

In CM's access control model it is possible for a library to "wrap" privileges. If a privilege P has been wrapped, then the user of the library does not need to have privilege P even though the library is using another library that requires privilege P. In essence, the library acts as a go-between who provides the necessary credentials for privilege P to the sub-library.

Of course, not everybody can be allowed to establish a library with such a "wrapped" privilege P. The programmer who does that should at least herself have privilege P (but perhaps better, she should have *permission to wrap* P—a stronger requirement).

In CM, wrapping a privilege is done by specifying the name of that privilege within parenthesis. The wrapping becomes effective once the library gets stabilized via `CM.stabilize`. The (not yet implemented) enforcement mechanism must ensure that anyone who stabilizes a library that wraps P has permission to wrap P.

Note that privileges cannot be wrapped at the level of CM groups.

Access control is a new feature. At the moment, only the basic mechanisms are implemented, but there is no enforcement. In other words, everybody is assumed to have every possible privilege. CM merely reports which privileges "would have been required".

10 The pervasive environment

The *pervasive environment* can be thought of as a compilation unit that all compilation units implicitly depend upon. The pervasive enviroment exports all non-modular bindings (types, values, infix operators, overloaded symbols) that are mandated by the specification for the Standard ML Basis Library [RG99]. (All other bindings of the Basis Library are exported by `$/basis.cm` which is a genuine CM library.)

The pervasive environment is the only place where CM conveys non-modular bindings from one compilation unit to another, and its definition is fixed.

11 Files

CM uses three kinds of files to store derived information during and between sessions:

1. *Skeleton files* are used to store a highly abbreviated version of each ML source file's abstract syntax tree—just barely sufficient to drive CM's dependency analysis. Skeleton files are much smaller and (for a program) easier to read than actual ML source code. Therefore, the existence of valid skeleton files makes CM a lot faster because usually most parsing operations can be avoided that way.

2. *Binfiles* are the SML/NJ equivalent of object files. They contain executable code and a symbol table for the associated ML source file.

3. *Library files* (sometimes called: *stablefiles*) contain dependency graph, executable code, and symbol tables for an entire CM library including all of its components (groups). Other libraries used by a stable library are not included in full. Instead, references to those libraries are recorded using their (preferably anchored) pathnames.

Normally, all these files are stored in a subdirectory of directory `CM`. `CM` itself is a subdirectory of the directory where the original ML source file or—in the case of library files—the original CM description file is located.

Skeleton files are machine- and operating system-independent. Therefore, they are always placed into the same directory `CM/SKEL`. Parsing (for the purpose of dependency analysis) will be done only once even if the same file system is accessible from machines of different type.

Binfiles and library files contain executable code and other information that is potentially system- and architecture-dependent. Therefore, they are stored under `CM/`*arch*−*os* where *arch* is a string indicating the type of the current CPU architecture and *os* a string denoting the current operating system type.

As explained in Section 2.8, library files are a bit of an exception in the sense that they do not require any source files or any other derived files of the same library to exist. As a consequence, the location of such a library file should be described as being relative to "the location of the original CM description file if that description file still existed". (Of course, nothing precludes the CM description file from actually existing, but in the presence of a corresponding library file CM will not take any notice of that.)

Note: As discussed in section 7.1.2, CM sometimes looks for library files in `CM/`*version*`/`*arch*−*os*. However, library files are never *created* there by CM. If several versions of the same library are to be provided, an administrator must arrange the directory hierarchy accordingly "by hand".

11.1 Time stamps

For skeleton files and binfiles, CM uses file system time stamps (i.e., modification time) to determine whether a file has become outdated. The rule is that in order to be considered "up-to-date" the time stamp on skeleton file and binfile has to be exactly the same[10] as the one on the ML source file. This guarantees that all changes to a source will be noticed—even those that revert to an older version of a source file.[11]

CM also uses time stamps to decide whether tools such as ML-Yacc or ML-Lex need to be run (see Section 7). However, the difference is that a file is considered outdated if it is older than its source. Some care on the programmers side is necessary since this scheme does not allow CM to detect the situation where a source file gets replaced by an older version of itself.

11.2 Index files

CM can optionally generate a human-readable index file for each description file. An index files alphabetically lists all symbols defined or imported within the given library or library component. Index-file generation is normally disabled. To enable it, `CM.Control.generate_index` must be set to true (see section 5.1).

With index-file generation enabled, index files will be written for all description files involved every time CM performs a dependency analysis. (In other words, it is a side-effect to other CM operations such as `CM.make` etc.) If the name of the description files is p`/`d`.cm`, then the corresponding index file will be in p`/CM/INDEX/`d`.cm`.

12 Extending the tool set

CM's tool set is extensible: new tools can be added by writing a few lines of ML code. The necessary hooks for this are provided by a structure `Tools` which is exported by the `$smlnj/cm/tools.cm` library.

[10] CM explicitly sets the time stamp to be the same.

[11] except for the pathological case where two different versions of the same source file have exactly the same time stamp

12.1 Adding simple shell-command tools

If the tool is implemented as a "typical" shell command, then all that needs to be done is a single call of:

```
Tools.registerStdShellCmdTool
```

For example, suppose you have made a new, improved version of ML-Yacc ("New-ML-Yacc") and want to register it under a class called `nmlyacc`. Here is what you write:

```
val _ = Tools.registerStdShellCmdTool
  { tool = "New-ML-Yacc",
    class = "nmlyacc",
    suffixes = ["ngrm", "ny"],
    cmdStdPath = "new-ml-yacc",
    template = NONE,
    extensionStyle =
        Tools.EXTEND [("sig", SOME "sml", fn _ => NONE),
                      ("sml", SOME "sml", fn x => x)],
    dflopts = [] }
```

This code can be packaged as a CM library and loaded via `CM.make` or `CM.load_plugin`. (`CM.autoload` is not enough because of its lazy nature which prevents the required side-effects to occur.) Alternatively, the code could also be entered at the interactive top level after loading library `$smlnj/cm/tools.cm`.

In our example, the shell command name for our tool is `new-ml-yacc`. When looking for this command in the filesystem, CM first tries to treat it as a path anchor (see section 3.3). For example, suppose `new-ml-yacc` is mapped to `/bin`. In this case the command to be invoked would be `/bin/new-ml-yacc`. If path anchor resolution fails, then the command name will be used as-is. Normally this causes the shell's path search mechanism to be used as a fallback.

`Tools.registerStdShellCmdTool` creates the class and installs the tool for it. The arguments must be specified as follows:

tool a descriptive name of the tool (used in error messages); type: `string`

class the name of the class; the string must not contain upper-case letters; type: `string`

suffixes a list of file name suffixes that let CM automatically recognize files of the class; type: `string list`

cmdStdPath the command string from above; type: `string`

template an optional string that describes how the command line is to be constructed from pieces;
The string is taken verbatim except for embedded % format specifiers:

- **%c** the command name (i.e., the elaboration of `cmdStdPath`)
- **%s** the source file name in native pathname syntax
- **%nt** the n-th target file in native pathname syntax;
  (n is specified as a decimal number, counting starts at 1, and each target file name is constructed from the corresponding `extensionStyle` entry; if n is 0 (or missing), then all targets—separated by single spaces—are inserted; if n is not in the range between 0 and the number of available targets, then **%nt** expands into itself)
- **%no** the n-th tool parameter;
  (named sub-option parameters are ignored; n is specified as a decimal number, counting starts at 1; if n is 0 (or missing), then all options—separated by single spaces—are inserted; if n is not in the range between 0 and the number of available options, then **%no** expands into itself)
- **%x** the character x (where x is neither **c**, nor **s**, **t**, or **o**)

If no template string is given, then it defaults to `"%c %s"`.

extensionStyle a specification of how the names of files generated by the tool relate to the name of the tool input file; type: `Tools.extensionStyle`.
Currently, there are two possible cases:

1. "`Tools.EXTEND` l" says that if the tool source file is *file* then for each suffix *sfx* in `(map #1` l`)` there will be one tool output file named *file*`.`*sfx*. The list l consists of triplets where the first component specifies the suffix string, the second component optionally specifies the member class name of the corresponding derived file, and the third component is a function to calculate tool options for the target from those of the source. (Argument and result type of these functions is `Tools.toolopts option`.)
2. "`Tools.REPLACE` (l_1, l_2)" says that given the base name *base* there will be one tool output file *base*`.`*sfx* for each suffix *sfx* in `(map #1` l_2`)`. Here, *base* is determined by the following rule: If the name of the tool input file has a suffix that occurs in l_1, then *base* is the name without that suffix. Otherwise the whole file name is taken as *base* (just like in the case of `Tools.EXTEND`). As with `Tools.EXTEND`, the second components of the elements of l_2 can optionally specify the member class name of the corresponding derived file, and the third component maps source options to target options.

dflopts a list of tool options which is used for substituting **%**n**o** fields in `template` (see above) if no options were specified. (Note that the value of `dflopts` is never passed to the option mappers in `Tools.EXTEND` or `Tools.REPLACE`.) Type: `Tools.toolopts`.

Examples for the `EXTEND` expansion style are tools such as ML-Yacc and ML-Lex, while others, e.g., ML-Burg, use the `REPLACE` style (see section 7.2).

12.2 Adding other classes

Adding a new class whose behavior is not covered by the mechanism described in section 12.1 is not complicated either, but it requires a bit more code.

12.2.1 Filename abstractions

CM represents filenames as something that could be called a *filename closure*. Essentially, what this means is that not only a string is being remembered but also the context in which to interpret the string. For a relative path, context information is the directory in which it is to be interpreted; for an anchored path, the context takes care of the anchoring.

The `Tools` module provides two abstract types related to this filename abstraction:

```
type presrcpath
type srcpath
```

Since many tools invoke external shell commands or perform other operation on physical files, it is often necessary to obtain an actual native filename string from an abstract path:

```
val nativeSpec : srcpath -> string
val nativePreSpec : presrcpath -> string
```

It is important to remember that these two functions frequently return relative filenames, and such relative names must be interpreted from within the right directory. This "right" directory is the directory that contains the CM description file on whose behalf the tool's rule was invoked. Rules that perform physical operations on files whose names result from `nativeSpec` or `nativePreSpec` must therefore first switch to that directory. See the discussion of the `context` argument to rule functions below.

Strings that can potentially be used as pathnames are being passed around as records containing a `name`- and a `mkpath` field. The `name` field contains the string itself while `mkpath` is a "suspended" abstract version of the path.

A fresh pathmaker can be constructed from native strings using `native2pathmaker`, which like `context` is also an argument to rule functions. This function takes care of interpreting relative strings within the correct context. (For this, it is not even necessary to first switch the current working directory.)

Recall that filenames that appear in description files can be written using either "standard" or "native" syntax. The field `name` will, thus, contain a string that can be in either of these two forms. However, CM will pass an `mkpath` function that accounts for these syntactic differences and which also takes care of interpreting the string in the correct context.

When new name specifications are constructed by the tool, the appropriate `mkpath` function must be provided. For most tools this will be a value constructed by applying `native2pathmaker` to some native filename string (because most tools internally operate on native paths).

As shown above, there are two abstract path types: `presrcpath` and `srcpath`. The former can represent both directory names and source file names, the latter can represent only source file names. To convert from `presrcpath` to `srcpath`, use function `Tools.srcpath`:

```
val srcpath : presrcpath -> srcpath
```

This function enforces CM's rule that there has to be at least one arc in every such name (i.e., that it cannot be just an anchor).

One can also construct a new abstract path from an existing path by adding arcs at the end. The constructed path will share its internal context with the old one.

```
val augment : presrcpath -> string list -> presrcpath
```

The list of strings must contain simple pathname arcs.

12.2.2 Adding a class and its rule

The interface to add arbitrary classes is the routine `Tools.registerClass`:

```
val registerClass : class * rule -> unit
```

Here, type `class` is simply synonymous to `string`; a class string is the name of the class to be registered. It must not contain upper-case letters:

```
type class = string
```

Type `rule` is a function type. It describes the rule function that CM will invoke for every member of the new class. The rule function is responsible for invoking the auxiliary mechanism necessary to bring its targets up-to-date. The function result of the rule function describes to CM what the targets are. Thus, the function maps the *specification* of the given member to its (partial) *expansion*:

```
type rule =
  { spec: spec,
    native2pathmaker: string -> pathmaker,
    context: rulecontext,
    defaultClassOf: fnspec -> class option } ->
  partial_expansion
```

The specification spec consists of the name of the member together with a function to produce its corresponding abstract path (should that be necessary), the member's optional class (in case it had been given explicitly), its tool options, and a boolean flag that tells whether this member was the result of another tool:

```
type spec = { name: string,
              mkpath: pathmaker,
              class: class option,
              opts: toolopts option,
              derived: bool }
```

name: The name is the verbatim member string from the description file. Be sure not to use this string directly as a file name (although some tools might use it directly for purposes other than file names). Instead, first convert it to an abstract path (see mkpath below) and then convert back to a *native* file name string using one of nativeSpec or nativePreSpec.

mkpath This is a function of type pathmaker that produces an abstract pathname corresponding to name. CM will pass in different functions here depending on whether name was given in CM's standard pathname syntax or in the underlying operating system's native syntax.

```
type pathmaker = unit -> presrcpath
```

class: This argument carries the class name if such a class name was explicitly specified. If the class was inferred from the member name, then it will be set to NONE.

opts: Tool options are represented by a data structure resembling Lisp lists:

```
type fnspec = { name: string, mkpath: pathmaker }
datatype toolopt =
    STRING of fnspec
  | SUBOPTS of { name: string, opts: toolopts }
withtype toolopts = toolopt list
```

The nesting of SUBOPTS reflects the nesting of sub-option lists in the member's tool option specification. Again, names which are potentially to be interpreted as file names are represented by their original specification string and a function mkpath to get the corresponding abstract path, thereby taking care of interpreting the name according to its respective syntactic rules and its context. (Type fnspec is a slimmed-down version of type spec. It also appears as the argument type of function defaultClassOf. See below.)

derived: This flag is set to true if the source file represented by the specification is the result of a another, earlier tool invocation.

The other three arguments of a rule function are native2pathmaker, context, and defaultClassOf:

native2pathmaker: This function takes a string and produces a function of the same type as mkpath above. When the rule constructs the specifications for its result files, it must provide the corresponding mkpath functions for those. Since most tools internally operate on native pathnames, these mkpath functions will have to be constructed using native2pathmaker.

context: The context argument of a rule represents the directory that contains the CM description file on whose behalf the rule was invoked. It is represented as a higher-order function that invokes its function argument after temporarily setting the working directory to the context directory and returns the result of this invocation after restoring the original working directory. Not all rules require such a temporary change of directories, but those that do should encapsulate all their work into a local function and then pass this function to the context.

```
type rulefn = unit -> partial_expansion
type rulecontext = rulefn -> partial_expansion
```

defaultClassOf: This function can be used to directly invoke CM's internal classification mechanism, taking advantage of any registered classifiers. The argument to be passed is of type `fnspec`, i.e., a record consisting of a name string and a function to convert the string to its corresponding abstract path.

A (full) *expansion* consists of three lists: a list of ML files, a list of CM files, and a list of *sources*. A partial expansion is a full expansion together with a list of specifications that still need to be expanded further.

```
type expansion =
  { smlfiles: (srcpath * Sharing.request * setup) list,
    cmfiles: (srcpath * Version.t option * rebindings) list,
    sources: (srcpath * { class: class, derived: bool}) list }

type partial_expansion = expansion * spec list
```

A rule always returns a partial expansion. CM will derive a full expansion by repeatedly applying rules until the list of pending specification becomes empty.

Most rules (except those for classes `sml` and `cm`) leave the lists `smlfiles` and `cmfiles` empty. A tool that produces an ML source file or a CM description file as output should put a specification for this file into the specification list of a partial expansion, letting the rules for classes `sml` and `cm` take care of the rest. At this point we will therefore not dwell on explanations for the types of these two fields.

The `sources` field is used to implement `CM.sources` (see section 5.1.11). Therefore, the rule should include here every file that it consumes if its implementer wishes to have it reported by `CM.sources`. (Do not include source files that are *produced* by the rule because those will be reported by subsequent rules.)

12.2.3 Reporting errors from tools

When a rule encounters an error, it should raise the following exception, setting `tool` to a string describing the current tool and `msg` to a diagnostic string describing the nature of the error:

```
exception ToolError of { tool: string, msg: string }
```

12.2.4 Adding a classifier

A classifier is a mechanism that enables CM to infer a member's class from its name. Classifiers are invoked if no explicit class was given. CM supports two kinds of classifiers: suffix classifiers and general classifiers.

```
datatype classifier =
    SFX_CLASSIFIER of string -> class option
  | GEN_CLASSIFIER of { name: string, mkfname: unit -> string } ->
                      class option
```

Most of the time classifiers look at the file name suffix as their only clue. This idea is captured by `SFX_CLASSIFIER` which carries a partial function from suffixes to class names. The function should return `NONE` if it does not know about the given argument suffix.

The `GEN_CLASSIFIER` constructor carries a similar function—the difference being that the entire member name is passed to it. Moreover, the function can also invoke the `mkfname` argument to obtain a native filename string. This string can at this point be used to perform actual filesystem operations.

Invocation of `mkfname` may raise exceptions, usually due to syntax errors in `name` that prevent it from being interpreted as a filename. Tools that use `mkfname` should therefore be prepared to handle such exceptions.

Moreover, it is advisable not to over-use this feature, and not to perform extensive filesystem processing in order to perform classification. Otherwise the presence of this classifier might cause considerable overhead.

By the way, suffix classifiers could be implemented as general classifiers, but using `SFX_CLASSIFIER` for them is slightly more efficient. CM extracts the suffix from the name only once and applies all suffix classifier before ever considering any generic classifier. If some suffix classifier succeeds, there will be no overhead caused by any generic classifier.

Function `Tools.stdSfxClassifier` is a simple wrapper around `SFX_CLASSIFIER` and produces a classifier that looks for precisely one suffix string.

```
val stdSfxClassifier : { sfx: string , class: class } -> classifier
```

Classifiers are registered with CM by invoking `Tools.registerClassifier`:

```
val registerClassifier : classifier -> unit
```

12.2.5 Miscellaneous

Structure `Tools` also provides a number of other types and functions with the purpose of making it easier to write rule functions.

Filename extension: Many tools derive the names of their targets from the name of their source. As discussed in section 12.1, CM provides some support for this via values of type `extensionStyle`:

```
type tooloptcvt = toolopts option -> toolopts option
datatype extensionStyle =
    EXTEND of (string * class option * tooloptcvt) list
  | REPLACE of string list * (string * class option * tooloptcvt) list
```

These values can not only be passed to `Tools.registerStdShellCmdTool` but also be used to let CM perform name extension directly. To do so, one must invoke function `Tools.extend`:

```
val extend : extensionStyle ->
             (string * toolopts option) ->
             (string * class option * toolopts option) list
```

Checking time stamps: A tool can check whether a given source file is older than all of its corresponding target files.

```
val outdated : string -> string list * string -> bool
```

In a call `(Tools.outdated` t `(`l`,` s`))`, t is the name of the tool, l is the list of targets (as native file names), and s is the source (also as a native file name).

An alternative way of checking for outdated sources (in the style of the Noweb-tool; see section 7.2.6) is the following:

```
val outdated' : string ->
                { src: string, wtn: string, tgt: string } -> bool
```

The idea here is that if both `tgt` ("target") and `wtn` ("witness") exist, then `tgt` is considered outdated if `wtn` is older than `src`. Otherwise, if `tgt` exists but `wtn` does not, then `tgt` is considered outdated if it is older than `src`. If `tgt` does not exist, then it is always considered outdated.

File- and directory-creation: To open a text file for output in such a way that all directories leading up to it are created when they do not already exist, use `Tools.openTextOut`:

```
val openTextOut : string -> TextIO.outstream
```

To create the same directories without opening the file (and without even creating it if it does not exist), use function `Tools.makeDirs`:

```
val makeDirs : string -> unit
```

Note that the string passed to `makeDirs` is still the name of a file!

Option processing: For simple tools, the following function for "parsing" tool options can be useful:

```
val parseOptions :
    { tool : string, keywords : string list, options : toolopts } ->
    { matches : string -> toolopts option, restoptions : string list }
```

Given a list of accepted keywords, this function scans the tool options and collects occurrences of sub-option lists labelled by one of these keywords. Any sub-option list that is not recognized and any keyword that occurs more than once will be rejected as an error. The result consists of a function `matches` that can be uses to query each of the keywords. The function also collects and returns all the `STRING` options.

Issuing diagnostics: Functions `Tools.say` and `Tools.vsay` both take a list of strings and output the concatenation of these strings to the compiler's standard control output stream (i.e., usually `TextIO.stdOut`). The difference between `say` and `vsay` is that the former works unconditionally while the latter is controlled by `CM.Control.verbose` (see section 5.1.3).

Anchor-configurable strings: Mainly for the purpose of implementing anchor-configurable names for auxiliary shell commands (such as `ml-yacc`), one can invoke `Tools.mkCmdName`:

```
val mkCmdName : string -> string
```

If m is a path anchor that points to d, then `(mkCmdName` m`)` returns d/m; otherwise it returns m.

12.3 Plug-in Tools

12.3.1 Automatically loaded, global plug-in tools

If CM encounters a member class name c that it does not know about, then it tries to load a plugin module named `$/`c`-tool.cm`. If it sees a file whose name ends in suffix s for which no explicit member class has been specified in the CM description file and for which automatic member classification fails, it will try to load a plugin module named `$/`s`-ext.cm`. The so-loaded module can then register the required tool, thereby enabling CM to successfully deal with the previously unknown member.

This mechanism makes it possible for new tools to be added by simply placing appropriately named plug-in libraries in some convenient place and making the corresponding adjustments to the anchor environment. In other words, description files `$/`c`-tool.cm` and `$/`s`-ext.cm` that correspond to general-purpose tools should be registered either by modifying the global or the local path configuration file or by directly invoking functon `CM.Anchor.anchor` (see section 5.1.4). Actual description files for the tools' implementations can then be placed in arbitrary locations.

12.3.2 Explicitly loaded, local plug-in tools

Some projects might want to use their own special-purpose tools for which a global installation is not convenient or not appropriate. In such a case, the project's description file can explicitly demand the tool to be registered temporarily. This is the purpose of the special tool class `tool`. Example:

```
Library
    structure Foo
is
    bar-tool.cm : tool
    foo.b : bar
```

Here, the member whose class is `tool` (i.e, `bar-tool.cm`) must be the CM description file of the tool's implementation. The difference to class `cm` is that the so-specified library does not become part of the current project but is loaded and linked immediately via `CM.load_plugin`, causing one or more new classes and their classifiers to be registered.

If we assume that loading `bar-tool.cm` causes a class `bar` to be registered with its associated rule (e.g., by invoking `Tools.registerStdShellCmdTool`), the class name `bar` will be available for all subsequent members of the current description file. Likewise, classifiers (e.g., filename suffixes) registered by `bar-tool.cm` will also be available.

The effect of registering classes and classifiers using class `tool` lasts until the end of the current description file and is restricted to that file. This means that other description files that also want to use class `bar` will have to have their own `tool` entry.[12]

Local tool classes and suffixes temporarily override any equally-named global classes or suffixes, respectively.

12.3.3 Locally declared suffixes

It is sometimes convenient to locally add another recognized filename suffix to an already registered class. This is the purpose of the special tool class `suffix`. For example, a programmer who has named all ML files in such a way that file names end in `.ml` could write near the beginning of the description file:

```
    ml : suffix (sml)
```

For the remainder of the current description file, all such `.ml`-files will now be treated as members of class `sml`.

13 Parallel and distributed compilation

To speed up recompilation of large projects with many ML source files, CM can exploit parallelism that is inherent in the dependency graph. Currently, the only kind of operating system for which this is implemented is Unix (`OPSYS_UNIX`), where separate processes are used. From there, one can distribute the work across a network of machines by taking advantage of the network file system and the "rsh" facility.

To perform parallel compilations, one must attach *compile servers* to the currently running CM session. This is done using function `CM.Server.start`, which has the following type:

```
  structure Server : sig
      type server
      val start : { name: string,
                    cmd: string * string list,
                    pathtrans: (string -> string) option,
                    pref: int } -> server option
  end
```

[12]Note that CM cannot enforce that the tool library actually register a class or a classifier. Any side-effects other than registering classes or classifiers are beyond CM's control and will not be undone once processing the current description file is complete.

Here, `name` is an arbitrary string that is used by CM when issuing diagnostic messages concerning the server[13] and `cmd` is a value suitable as argument to `Unix.execute`.

The program to be specified by `cmd` should be another instance of CM—running in "slave mode". To start CM in slave mode, start `sml` with a single command-line argument of `@CMslave`. For example, if you have installed in /path/to/smlnj/bin/sml, then a server process on the local machine could be started by

```
CM.Server.start { name = "A", pathtrans = NONE, pref = 0,
                  cmd = ("/path/to/smlnj/bin/sml",
                         ["@CMslave"]) };
```

To run a process on a remote machine, e.g., "thatmachine", as compute server, one can use "rsh".[14] Unfortunately, at the moment it is necessary to specify the full path to "rsh" because `Unix.execute` (and therefore `CM.Server.start`) does not perform a `PATH` search. The remote machine must share the file system with the local machine, for example via NFS.

```
CM.Server.start { name = "thatmachine",
                  pathtrans = NONE, pref = 0,
                  cmd = ("/usr/ucb/rsh",
                         ["thatmachine",
                          "/path/to/smlnj/bin/sml",
                          "@CMslave"]) };
```

You can start as many servers as you want, but they all must have different names. If you attach any servers at all, then you should attach at least two (unless you want to attach one that runs on a machine vastly more powerful than your local one). Local servers make sense on multi-CPU machines: start as many servers as there are CPUs. Parallel make is most effective on multiprocessor machines because network latencies can have a severely limiting effect on what can be gained in the distributed case. (Be careful, though. Since there is no memory-sharing to speak of between separate instances of `sml`, you should be sure to check that your machine has enough main memory.)

If servers on machines of different power are attached, one can give some preference to faster ones by setting the `pref` value higher. (But since the `pref` value is consulted only in the rare case that more than one server is idle, this will rarely lead to vastly better throughput.) All attached servers must use the same architecture-OS combination as the controlling machine.

In parallel mode, the master process itself normally does not compile anything. Therefore, if you want to utilize the master's CPU for compilation, you should start a compile server on the same machine that the master runs on (even if it is a uniprocessor machine).

The `pathtrans` argument is used when connecting to a machine with a different file-system layout. For local servers, it can safely be left at `NONE`. The "path transformation" function is used to translate local path names to their remote counterparts. This can be a bit tricky to get right, especially if the machines use automounters or similar devices. The `pathtrans` functions consumes and produces names in CM's internal "protocol encoding" (see Section 13.1).

Once servers have been attached, one can invoke functions like `CM.recomp`, `CM.make`, and `CM.stabilize`. They should work the way the always do, but during compilation they will take advantage of parallelism.

When CM is interrupted using Control-C (or such), one will sometimes experience a certain delay if servers are currently attached and busy. This is because the interrupt-handling code will wait for the servers to finish what they are currently doing and bring them back to an "idle" state first.

[13] Therefore, it is useful to choose `name` uniquely.

[14] On certain systems it may be necessary to wrap `rsh` into a script that protects rsh from interrupt signals.

13.1 Pathname protocol encoding

A path encoded by CM's master-slave protocol encoding does not only specify which file a path refers to but also, in some sense, specifies why CM constructed this path in the first place. For example, the encoding `a/b/c.cm:d/e.sml` represents the file `a/b/d/e.sml` but also tells us that it was constructed by putting `d/e.sml` into the context of description file `a/b/c.cm`. Thus, an encoded path name consists of one or more colon-separated (**:**) sections, and each section consists of slash-separated (**/**) arcs. To find out what actual file a path refers to, it is necessary to erase all arcs that precede colons.

The first section is special because it also specifies whether the whole path was relative or absolute, or whether it was an anchored path.

Anchored paths start with a dollar-symbol **$**. The name of the anchor is the string between this leading dollar-symbol and the first occurence of a slash **/** within the first section. The remaining arcs of the first section are interpreted relative to the current value of the anchor.

Absolute paths start either with a percent-sign **%** or a slash **/**. The canonical form is the one with the percent-sign: it specifies the volume name between the **%** and the first slash. In the common case where the volume name is empty (i.e, *always* on Unix systems), the path starts with **/**.

Relative paths are all other paths.

Encoded path names never contain white space. Moreover, the encoding for path arcs, volume names, or anchor names does not contain special characters such as **/**, **$**, **%**, **:**, `\`, **(**, and **)**. Instead, should white space or special characters occur in the non-encoded name, then they will be encoded using the escape-sequence `\ddd` where `ddd` is the decimal value of the respective character's ordinal number (i.e, the result of applying `Char.ord`).

The so-called *current* arc is encoded as **.**, the *parent* arc uses **..** as its representation. It might be that under some operating systems the names `.` or `..` do not actually refer to the current or the parent arc. In such a case, CM will encode the dots in these names using the `\ddd` method, too.

When issuing progress messages, CM shows path names in a form that is almost the same as the protocol encoding. The only difference is that arcs that precede colon-sign **:** are enclosed within parentheses to emphasize that they are "not really there". The same form is also used by `CM.Library.descr`.

13.2 Parallel bootstrap compilation

The bootstrap compiler[15] with its main function `CMB.make` and the corresponding cross-compilation variants of the bootstrap compiler will also use any attached compile servers. If one intends to exclusively use the bootstrap compiler, one can even attach servers that run on machines with different architecture or operating system.

Since the master-slave protocol is fairly simple, it cannot handle complicated scenarios such as the one necessary for compiling the "init group" (i.e., the small set of files necessary for setting up the "pervasive" environment) during `CMB.make`. Therefore, this will always be done locally by the master process.

14 The `sml` command line

The SML/NJ interactive system—including CM—is started from the operating system shell by invoking the command `sml`. This section describes those arguments accepted by `sml` that are related to (and processed by) CM.

CM accepts *file names*, *mode switching flags*, *preprocessor definitions*, and *control parameters* as arguments. All these arguments are processed one-by-one from left to right.

[15] otherwise not mentioned in this document

14.1 File arguments

Names of ML source files and CM description files can appear as arguments in any order.

ML source files are recognized by their filename extensions (`.sig`, `.sml`, or `.fun`) and cause the named file to be loaded via `use` at the time the argument is being considered. Names of ML source files are specified using the underlying operating system's native pathname syntax.

CM description files are recognized by their extension `.cm`. They must be specified in CM's *standard* pathname syntax. At the time the argument is being considered, the named library (or group) will be loaded by passing the name to either `CM.autoload` or `CM.make`—depending on which *mode switching flag* (`-a` or `-m`) was specified last. The default is `-a` (i.e., `CM.autoload`).

14.2 Mode-switching flags

By default, CM description files are loaded via `CM.autoload`. By specifying `-m` somewhere on the command line one can force the system to use `CM.make` for all following description files up to the next occurence of `-a`. The `-a` flag switches back to the default behavior, using `CM.autoload`, which will then again be in effect up to the next occurrence of another `-m`.

Mode-switching flags can be specified arbitrarily often on the same command line.

14.3 Defining and undefining CM preprocessor symbols

The following options for defining and undefining CM preprocessor symbols can also occur arbitrarily often. Their effects accumulate while processing the command line from left to right. The resulting final state of the internal preprocessor registry becomes observable in the interactive system.

`-D`*v*`=`*n* acts like `(#set (CM.symval "`*v*`") (SOME `*n*`))`.

`-D`*v* is equivalent to `-D`*v*`=1`.

`-U`*v* acts like `(#set (CM.symval "`*v*`") NONE)`.

14.4 Control Parameters

There are three kinds of control parameters:

help A request of help is one of `-h`, `-h`*num*, or `-H`. In general, `-h`*num* produces a help listing with all configurable controls up to obscurity level *num*. If *num* is omitted, it defaults to 0. The `-H` flag requests a full help listing with all controls.

current settings This is similar to requesting help—except `-h` and `-H` are replaced with `-s` and `-S`, respectively. Instead of showing the help message associated with each control, this requests showing the current settings.

set a control A control value can be set using an argument of the form `-C`*control*=*value*, where *control* is the name of a control and `value` is a string that can be parsed and converted to a new value appropriate for the type of the control.

15 Auxiliary scripts

15.1 Building stand-alone programs

The programmer should normally have no need to invoke `CM.mk_standalone` (see Section 5.1.10) directly. Instead, SML/NJ provides a command `ml-build` which does all the work. To be able to use `ml-build`, one must implement a library exporting a structure that has some function suitable to be an argument to `SMLofNJ.exportFn`. Suppose the library is called `myproglib.cm`, the structure is called `MyProg`, and the function is called `MyProg.main`. If one wishes to produce a heap image file `myprog` one simply has to invoke the following command:

```
ml-build myproglib.cm MyProg.main myprog
```

The heap image is written only when needed: if a heap image exists and is newer than all ML sources involved, provided that none of the ML sources have to be recompiled, `ml-build` will just issue a message indicating that everything is up-to-date.

As in the case of `sml`, it is possible to define or undefine preprocessor symbols using `-D` or `-U` options (see Section 14.3). These options must be specified before the three regular arguments. Thus, the full command line syntax is:

```
ml-build [DU-options] myproglib.cm MyProg.main myprog
```

15.1.1 Bootstrapping: How `ml-build` works

Internally, `ml-build` generates a temporary wrapper library containing a single call of `SMLofNJ.exportFn` as part of the library's module-initialization code. Once this is done, CM is started, `CM.mk_standalone` is invoked (with the main project description file, the generated wrapper library file, and the heap image name as arguments), and a *bootlist* file is written. If all these steps were successful, `ml-build` invokes the (bare) SML/NJ runtime with a special option, causing it to *bootstrap* using the *bootlist* file.

Each line of the *bootlist* file specifies one module to be linked into the final stand-alone program. The runtime system reads these lines one-by-one, loads the corresponding modules, and executes their initialization code. Since the last module has been arranged (by way of using the wrapper library from above) to contain a call of `SMLofNJ.exportFn`, initialization of this module causes the program's heap image to be written and the bootstrap procedure to terminate.

15.2 Generating dependencies for `make`

When ML programs are used as parts of larger projects, it can become necessary to use CM (or, e.g., `ml-build` as described in Section 15.1) in a traditional makefile for Unix' **make**. To avoid repeated invocations, the dependency information that CM normally manages internally must be described externally so that **make** can process it.

For this purpose, it is possible to let CM's dependency analyzer generate a list of files that a given ML program depends on (see Section 5.1.11). The `ml-makedepend` scripts conveniently wraps this functionality in such a way that it resembles the familiar **makedepend** facility found on many Unix installations for the use by C projects.

An invocation of `ml-makedepend` takes two mandatory arguments: the root description file of the ML program in question and the name of the target that is to be used by the generated makefile entry. Thus, a typical command line has the form:

```
ml-makedepend project.cm targetname
```

This will cause `ml-makedepend` to first look for a file named `makefile` and if that cannot be found for `Makefile`. (An error message is issued if neither of the two exists.) After deleting any previously generated entry for this description-target combination, the script will invoke CM and add up-to-date dependency information to the file.

Using the `-f` option it is possible to force an arbitrary programmer-specified file to be used in place of `makefile` or `Makefile`.

Some of the files a CM-managed program depends on are stable libraries. Since the file names for stable libraries vary according to current CPU architecture and operating system, writing them directly would require different entries for different systems. To avoid this problem (most of the time[16]), `ml-makedepend` will use **make**-variables `$(ARCH)` and `$(OPSYS)` as placeholders within the information it generates. It is the programmer's responsibility to make sure that these variables are set to meaningful values at the time **make** is eventually being invoked. This feature can be turned off (causing actual file names to be used) by specifying the `-n` option to `ml-makedepend`.

In cases where the programmer prefers other strings to be used in place of `$(ARCH)` or `$(OPSYS)` (or both) one can specify those strings using the `-a` and `-o` options, respectively.

Like `ml-build` (Section 15.1) and `sml` (Section 14.3), the `ml-makedepend` command also accepts `-D` and `-U` command line options.

Thus, the full command line syntax for `ml-makedepend` is:

```
ml-makedepend [DU-options] [-n] [-f makefile] project.cm target
ml-makedepend [DU-options] [-a arch] [-o os] [-f makefile] project.cm target
```

(If `-n` is given, then any additional `-a` or `-o` options—while not illegal—will be ignored.)

16 Example: Dynamic linking

Autoloading is convenient and avoids wasted memory for modules that should be available at the interactive prompt but have not actually been used so far. However, sometimes one wants to be even more aggressive and save the space needed for a function until—at runtime—that function is actually being dynamically invoked.

CM does not provide immediate support for this kind of *dynamic linking*, but it is quite simple to achieve the effect by carefully arranging some helper libraries and associated stub code.

Consider the following module:

```
structure F = struct
    fun f (x: int): int =
        G.g x + H.h (2 * x + 1)
end
```

Let us further assume that the implementations of structures `G` and `H` are rather large so that it would be worthwhile to avoid loading the code for `G` and `H` until `F.f` is called with some actual argument. Of course, if `F` were bigger, then we also want to avoid loading `F` itself.

To achieve this goal, we first define a *hook* module which will be the place where the actual implementation of our function will be registered once it has been loaded. This hook module is then wrapped into a hook library. Thus, we have `f-hook.cm`:

```
Library
    structure F_Hook
is
    f-hook.sml
```

[16]The careful reader may have noticed that because of CM's conditional compilation it is possible that dependency information itself varies between different architectures or operating systems.

and f-hook.sml:

```
structure F_Hook = struct
    local
        fun placeholder (i: int) : int =
            raise Fail "F_Hook.f: unitinialized"
        val r = ref placeholder
    in
        fun init f = r := f
        fun f x = !r x
    end
end
```

The hook module provides a reference cell into which a function of type equal to F.f can be installed. Here we have chosen to hide the actual reference cell behind a **local** construct. Accessor functions are provided to install something into the hook (init) and to invoke the so-installed value (f).

With this preparation we can write the implementation module f-impl.sml in such a way that not only does it provide the actual code but also installs itself into the hook:

```
structure F_Impl = struct
    local
        fun f (x: int): int =
            G.g x + H.h (2 * x + 1)
    in
        val _ = F_Hook.init f
    end
end
```

The implementation module is wrapped into its implementation library f-impl.cm:

```
Library
    structure F_Impl
is
    f-impl.sml
    f-hook.cm
    g.cm       (* imports G *)
    h.cm       (* imports H *)
```

Note that f-impl.cm must mention f-hook.cm for f-impl.sml to be able to access structure F_Hook.

Finally, we replace the original contents of f.sml with a stub module that defines structure F:

```
structure F = struct
    local
        val initialized = ref false
    in
        fun f x =
            (if !initialized then ()
             else if CM.make "f-impl.cm" then initialized := true
             else raise Fail "dynamic linkage for F.f failed";
             F_Hook.f x)
    end
end
```

The trick here is to explicitly invoke `CM.make` the first time `F.f` is called. This will then cause `f-impl.cm` (and therefore `g.cm` and also `h.cm`) to be loaded and the "real" implementation of `F.f` to be registered with the hook module from where it will then be available to this and future calls of `F.f`.

For the new `f.sml` to be compiled successfully it must be placed into a library `f.cm` that mentions `f-hook.cm` and `$smlnj/cm.cm`. As we have seen, `f-hook.cm` exports `F_Hook.f` and `$smlnj/cm.cm` is needed because it exports `CM.make`:

```
Library
    structure F
is
    f.sml
    f-hook.cm
    $smlnj/cm.cm (* or $smlnj/cm/full.cm *)
```

Beware! This solution makes use of `$smlnj/cm.cm` which in turn requires the SML/NJ compiler to be present. Therefore, is worthwhile only for really large program modules where the benefits of their absence are not outweighed be the need for the compiler.

17 Some history

Although its programming model is more general, CM's implementation is closely tied to the Standard ML programming language [MTHM97] and its SML/NJ implementation [AM91].

The current version is preceded by several other compilation managers. Of those, the most recent went by the same name "CM" [Blu95], while earlier ones were known as IRM (*Incremental Recompilation Manager*) [HLPR94b] and SC (for *Separate Compilation*) [HLPR94a]. CM owes many ideas to SC and IRM.

Separate compilation in the SML/NJ system heavily relies on mechanisms for converting static environments (i.e., the compiler's symbol tables) into linear byte stream suitable for storage on disks [AM94]. However, unlike all its predecessors, the current implementation of CM is integrated into the main compiler and no longer relies on the *Visible Compiler* interface.

A CM description file syntax

A.1 Lexical Analysis

The CM parser employs a context-sensitive scanner. In many cases this avoids the need for "escape characters" or other lexical devices that would make writing description files cumbersome. (The downside of this is that it increases the complexity of both documentation and implementation.)

The scanner skips all nestable SML-style comments (enclosed with **(*** and ***)**).

Lines starting with **#line** may list up to three fields separated by white space. The first field is taken as a line number and the last field (if more than one field is present) as a file name. The optional third (middle) field specifies a column number. A line of this form resets the scanner's idea about the name of the file that it is currently processing and about the current position within that file. If no file is specified, the default is the current file. If no column is specified, the default is the first column of the (specified) line. This feature is meant for program-generators or tools such as `noweb` but is not intended for direct use by programmers.

The following lexical classes are recognized:

Namespace specifiers: structure, **signature**, **functor**, or **funsig**. These keywords are recognized everywhere.

CM keywords: group, **Group**, **GROUP**, **library**, **Library**, **LIBRARY**, **source**, **Source**, **SOURCE**, **is**, **IS**, ***, **-**. These keywords are recognized everywhere except within "preprocessor" lines (lines starting with **#**) or following one of the namespace specifiers.

Preprocessor control keywords: #if, **#elif**, **#else**, **#endif**, **#error**. These keywords are recognized only at the beginning of the line and indicate the start of a "preprocessor" line. The initial **#** character may be separated from the rest of the token by white space (but not by comments).

Preprocessor operator keywords: defined, **div**, **mod**, **andalso**, **orelse**, **not**. These keywords are recognized only when they occur within "preprocessor" lines. Even within such lines, they are not recognized as keywords when they directly follow a namespace specifier—in which case they are considered SML identifiers.

SML identifiers (*mlid*): Recognized SML identifiers include all legal identifiers as defined by the SML language definition. (CM also recognizes some tokens as SML identifiers that are really keywords according to the SML language definition. However, this can never cause problems in practice.) SML identifiers are recognized only when they directly follow one of the namespace specifiers.

CM identifiers (*cmid*): CM identifiers have the same form as those ML identifiers that are made up solely of letters, decimal digits, apostrophes, and underscores. CM identifiers are recognized when they occur within "preprocessor" lines, but not when they directly follow some namespace specifier.

Numbers (*number*): Numbers are non-empty sequences of decimal digits. Numbers are recognized only within "preprocessor" lines.

Preprocessor operators: The following unary and binary operators are recognized when they occur within "preprocessor" lines: `+`, `-`, `*`, `/`, `%`, `<>`, `!=`, `<=`, `<`, `>=`, `>`, `==`, `=`, `~`, `&&`, `||`, `!`. Of these, the following ("C-style") operators are considered obsolete and trigger a warning message[17] as long as `CM.Control.warn_obsolete` is set to `true`: `/`, `%`, `!=`, `==`, `&&`, `||`, `!`.

Standard path names (*stdpn*): Any non-empty sequence of upper- and lower-case letters, decimal digits, and characters drawn from `'_.;,!%&$+/<=>?@~|#*-^` that occurs outside of "preprocessor" lines and is neither a namespace specifier nor a CM keyword will be recognized as a stardard path name. Strings that lexically constitute standard path names are usually—but not always—interpreted as file names. Sometimes they are simply taken as literal strings. When they act as file names, they will be interpreted according to CM's *standard syntax* (see Section 3.2). (Member class names, names of privileges, and many tool optios are also specified as standard path names even though in these cases no actual file is being named.)

[17] The use of `-` as a unary minus also triggers this warning.

Native path names (*ntvpn*): A token that has the form of an SML string is considered a native path name. The same rules as in SML regarding escape characters apply. Like their "standard" counterparts, native path names are not always used to actually name files, but when they are, they use the native file name syntax of the underlying operating system.

Punctuation: A colon **:** is recognized as a token everywhere except within "preprocessor" lines. Parentheses **()** are recognized everywhere.

A.2 EBNF for preprocessor expressions

Lexical conventions: Syntax definitions use *Extended Backus-Naur Form* (EBNF). This means that vertical bars | separate two or more alternatives, curly braces {} indicate zero or more copies of what they enclose ("Kleene-closure"), and square brackets [] specify zero or one instances of their enclosed contents. Round parentheses () are used for grouping. Non-terminal symbols appear in *this* typeface; terminal symbols are **underlined**.

The following set of rules defines the syntax for CM's preprocessor expressions (*ppexp*):

$$
\begin{array}{rcl}
\mathit{aatom} & \rightarrow & \mathit{number} \mid \mathit{cmid} \mid \underline{(}\ \mathit{asum}\ \underline{)} \mid (\underline{\texttt{\~{}}} \mid \underline{\texttt{-}})\ \mathit{aatom} \\
\mathit{aprod} & \rightarrow & \{\mathit{aatom}\ (\underline{\texttt{*}} \mid \underline{\textbf{div}} \mid \underline{\textbf{mod}}) \mid \underline{\texttt{/}} \mid \underline{\texttt{\%}}\ \}\ \mathit{aatom} \\
\mathit{asum} & \rightarrow & \{\mathit{aprod}\ (\underline{\texttt{+}} \mid \underline{\texttt{-}})\}\ \mathit{aprod} \\
 & & \\
\mathit{ns} & \rightarrow & \underline{\textbf{structure}} \mid \underline{\textbf{signature}} \mid \underline{\textbf{functor}} \mid \underline{\textbf{funsig}} \\
\mathit{mlsym} & \rightarrow & \mathit{ns}\ \mathit{mlid} \\
\mathit{query} & \rightarrow & \underline{\textbf{defined}}\ \underline{(}\ \mathit{cmid}\ \underline{)} \mid \underline{\textbf{defined}}\ \underline{(}\ \mathit{mlsym}\ \underline{)} \\
 & & \\
\mathit{acmp} & \rightarrow & \mathit{asum}\ (\underline{\texttt{<}} \mid \underline{\texttt{<=}} \mid \underline{\texttt{>}} \mid \underline{\texttt{>=}} \mid \underline{\texttt{=}} \mid \underline{\texttt{==}} \mid \underline{\texttt{<>}} \mid \underline{\texttt{!=}})\ \mathit{asum} \\
 & & \\
\mathit{batom} & \rightarrow & \mathit{query} \mid \mathit{acmp} \mid (\underline{\textbf{not}} \mid \underline{\texttt{!}})\ \mathit{batom} \mid \underline{(}\ \mathit{bdisj}\ \underline{)} \\
\mathit{bcmp} & \rightarrow & \mathit{batom}\ [(\underline{\texttt{=}} \mid \underline{\texttt{==}} \mid \underline{\texttt{<>}} \mid \underline{\texttt{!=}})\ \mathit{batom}] \\
\mathit{bconj} & \rightarrow & \{\mathit{bcmp}\ (\underline{\textbf{andalso}} \mid \underline{\texttt{\&\&}})\}\ \mathit{bcmp} \\
\mathit{bdisj} & \rightarrow & \{\mathit{bconj}\ (\underline{\textbf{orelse}} \mid \underline{\texttt{||}})\}\ \mathit{bconj} \\
 & & \\
\mathit{ppexp} & \rightarrow & \mathit{bdisj}
\end{array}
$$

A.3 EBNF for tool options

The following set of rules defines the syntax for tool options (*toolopts*):

$$
\begin{array}{rcl}
\mathit{pathname} & \rightarrow & \mathit{stdpn} \mid \mathit{ntvpn} \\
\mathit{toolopts} & \rightarrow & \{\ \mathit{pathname}\ [\underline{\textbf{:}}\ (\underline{(}\ \mathit{toolopts}\ \underline{)} \mid \mathit{pathname})]\ \}
\end{array}
$$

A.4 EBNF for export lists

The following set of rules defines the syntax for export lists (*elst*):

libkw → **library** | **Library** | **LIBRARY**
groupkw → **group** | **Group** | **GROUP**
sourcekw → **source** | **Source** | **SOURCE**
guardedexports → { *export* } (**#endif** | **#else** { *export* } **#endif** | **#elif** *ppexp guardedexports*)
restline → rest of current line up to next newline character
export → *difference* | **#if** *ppexp guardedexports* | **#error** *restline*
difference → *intersection* | *difference* - *intersection*
intersection → *atomicset* | *intersection* * *atomicset*
atomicset → *mlsym* | *union* | *implicitset*
union → ([*elst*])
implicitset → *sourceset* | *subgroupset* | *libraryset*
sourceset → *sourcekw* ((- | *pathname*))
subgroupset → *groupkw* ((- | *pathname*))
libraryset → *libkw* ((*pathname*) [(*toolopts*)])
elst → *export* { *export* }

A.5 EBNF for member lists

The following set of rules defines the syntax for member lists (*members*):

class → *stdpn*
member → *pathname* [: *class*] [(*toolopts*)]
guardedmembers → *members* (**#endif** | **#else** *members* **#endif** | **#elif** *ppexp guardedmembers*)
members → { (*member* | **#if** *ppexp guardedmembers* | **#error** *restline*) }

A.6 EBNF for library descriptions

The following set of rules defines the syntax for library descriptions (*library*). Notice that although the syntax used for *version* is the same as that for *stdpn*, actual version strings will undergo further analysis according to the rules given in section 6:

version → *stdpn*
privilege → *stdpn*
lprivspec → { *privilege* | ({ *privilege* }) }
library → [*lprivspec*] *libkw* [[(*version*)] *elst*] (**is** | **IS**) *members*

A.7 EBNF for library component descriptions (group descriptions)

The main differences between group- and library-syntax can be summarized as follows:

- Groups use keyword **group** instead of **library**.
- Groups may have an empty export list.
- Groups cannot wrap privileges, i.e., names of privileges (in front of the **group** keyword) never appear within parentheses.
- Groups have no version.
- Groups have an optional owner.

The following set of rules defines the syntax for library component (group) descriptions (*group*):

owner → *pathname*
gprivspec → { *privilege* }
group → [*gprivspec*] *groupkw* [(*owner*)] [*elst*] (**is** | **IS**) *members*

B Full signature of `structure CM`

Structure `CM` serves as the compilation manager's user interface and also constitutes the major part of the API. The structure is the (only) export of library `$smlnj/cm.cm`. The standard installation procedure of SML/NJ registers this library for autoloading at the interactive top level.

```
signature CM = sig

    val autoload : string -> bool
    val make : string -> bool
    val recomp : string -> bool
    val stabilize : bool -> string -> bool

    type 'a controller = { get : unit -> 'a, set : 'a -> unit }

    structure Anchor : sig
        val anchor : string -> string option controller
        val reset : unit -> unit
    end

    structure Control : sig
        val keep_going : bool controller
        val verbose : bool controller
        val parse_caching : int controller
        val warn_obsolete : bool controller
        val debug : bool controller
        val conserve_memory : bool controller
        val generate_index : bool controller
    end

    structure Library : sig
        type lib
        val known : unit -> lib list
        val descr : lib -> string
        val osstring : lib -> string
        val dismiss : lib -> unit
        val unshare : lib -> unit
    end

    structure State : sig
        val synchronize : unit -> unit
        val reset : unit -> unit
        val pending : unit -> string list
    end

    structure Server : sig
        type server
        val start : { cmd : string * string list,
                      name : string,
                      pathtrans : (string -> string) option,
                      pref : int } -> server option
        val stop : server -> unit
        val kill : server -> unit
        val name : server -> string
    end

    val sources :
        { arch: string, os: string } option ->
        string ->
```

```
        { file: string, class: string, derived: bool } list option

    val symval : string -> int option controller
    val load_plugin : string -> bool

    val mk_standalone :
        bool option ->
        { project: string, wrapper: string, target: string } ->
        string list option
end

structure CM : CM
```

C Listing of all pre-defined CM identifiers

	Alpha32 Unix	HP-PA Unix	PowerPC MACOS	PowerPC Unix	Sparc Unix	IA32 Unix	IA32 Win32
ARCH_ALPHA	1						
ARCH_HPPA		1					
ARCH_PPC			1	1			
ARCH_SPARC					1		
ARCH_X86						1	1
OPSYS_UNIX	1	1		1	1	1	
OPSYS_MACOS			1				
OPSYS_WIN32							1
BIG_ENDIAN					1		
LITTLE_ENDIAN	1	1	1	1		1	1
SIZE_32	1	1	1	1	1	1	1
NEW_CM	1	1	1	1	1	1	1
SMLNJ_VERSION	110	110	110	110	110	110	110
SMLNJ_MINOR_VERSION	40	40	40	40	40	40	40

D Listing of all CM-specific environment variables

Most control parameters that affect CM's operation can be adjusted using environment variables v_s at startup time, i.e, when the `sml` command is invoked. Each such parameter has a default setting. Default settings are determined at bootstrap time, i.e., the time when the heap image for SML/NJ's interactive system is built.[18] At bootstrap time, it is possible to adjust defaults by using a set of environment variables v_b. In the current version of CM it is always the case that $v_s = v_b$. (In older versions this was not the case.) If neither v_s nor v_b were set, a hard-wired fallback value will be used.

The rule for constructing the name v_s (and v_b) is the following: For each adjustable parameter x there is a *name stem*. If the stem for x is s, then $v_s = v_b =$ `CM_`s.

Since the normal installation procedure for SML/NJ sets some of the v_b variables at bootstrap time, there are two columns with default values in the following table. The value labeled *fallback* is the one that would have been used had there been no environment variable at bootrap time, the one labeled *default* is the one the user will actually see.

To save space, the table lists the stem but not the names for its associated (longer) v_s and v_b. For example, since the the table shows `VERBOSE` in the row for `CM.Control.verbose`, CM's per-session verbosity can be adjusted using `CM_VERBOSE`.

[18]Normally this is the same as installation time, but for SML/NJ compiler hackers there is also a `makeml` script for the purpose of bootstrapping.

CM.Control.*c*	stem	type	fallback	default	default's meaning
verbose	VERBOSE	bool	true	same	issue progess messages
debug	DEBUG	bool	false	same	do not issue debug messages
keep_going	KEEP_GOING	bool	false	same	quit on first error
(none)	PATHCONFIG	string	see below	see below	central path configuration file
parse_caching	PARSE_CACHING	int	100	same	at most 100 parse trees will be cached in main memory
(none)	LOCAL_PATHCONFIG	string	see below	same	user-specific path configuration file
warn_obsolete	WARN_OBSOLETE	bool	true	same	issue warnings about obsolete C-style operators in description files
conserve_memory	CONSERVE_MEMORY	bool	false	same	avoid repeated I/O operations by keeping certain information in main memory

The fallback for `PATHCONFIG` is `/usr/lib/smlnj-pathconfig`, but the standard installation overrides this and uses `$INSTALLDIR/lib/pathconfig` (where `$INSTALLDIR` is the SML/NJ installation directory) instead.

The default for the "local" path configuration file is `.smlnj-pathconfig`. This file is located in the user's home directory (given by the environment variable `$HOME`).

Control parameters can also be set using command-line parameters (see section 14).

E Listing of all class names and their tools

class	file contents	tool	file name suffixes
sml	ML source code	built-in	`.sig`, `.sml`, `.fun`
cm	CM description file	built-in	`.cm`
mlyacc	ML-Yacc grammar	ml-yacc	`.grm`, `.y`
mllex	ML-Lex specification	ml-lex	`.lex`, `.l`
mlburg	ML-Burg specification	ml-burg	`.burg`
noweb	literate program	noweb	`.nw`
make	makefile	make	
shell	arbitrary	shell command	
tool	CM library description file	built-in	
suffix	*(does not name a file)*	built-in	
dir	directory	built-in	*(directories recognized automatically)*

F Available libraries

Compiler and interactive system of SML/NJ consist of several hundred individual compilation units. Like modules of application programs, these compilation units are also organized using CM libraries.

Some of the libraries that make up SML/NJ are actually the same ones that application programmers are likely to use, others exist for organizational purposes only. There are "plugin" libraries—mainly for the CM "tools" subsystem—that will be automatically loaded on demand, and libraries such as `$smlnj/cmb.cm` can be used to obtain access to functionality that by default is not present.

F.1 Libraries for general programming

Libraries listed in the following table provide a broad palette of general-purpose programming tools[19]:

[19] Recall that anchored paths of the form $\texttt{\$/}x[\texttt{/}\cdots]$ act as an abbreviation for $\texttt{\$}x\texttt{/}x[\texttt{/}\cdots]$.

name	description	installed	loaded
`$/basis.cm`	Standard Basis Library	always	auto
`$/ml-yacc-lib.cm`	ML-Yacc library	always	no
`$/smlnj-lib.cm`	SML/NJ general-purpose utility library	always	auto
`$/unix-lib.cm`	SML/NJ Unix programming utility library	optional	no
`$/inet-lib.cm`	SML/NJ internet programming utility library	optional	no
`$/regexp-lib.cm`	SML/NJ regular expression library	optional	no
`$/reactive-lib.cm`	SML/NJ reactive programming library	optional	no
`$/pp-lib.cm`	SML/NJ pretty-printing library	always	no
`$/html-lib.cm`	SML/NJ HTML handling library	always	no
`$/controls-lib.cm`	SML/NJ library for configurable "controls"	always	no
`$/hash-cons-lib.cm`	SML/NJ hash-consing library	optional	no

F.2 Libraries for controlling SML/NJ's operation

The following table lists those libraries that provide access to the so-called *visible compiler* infrastructure and to the compilation manager API.

name	description	installed	loaded
`$smlnj/compiler.cm` `$smlnj/compiler/current.cm`	visible compiler for current architecture	always	auto
`$smlnj/compiler/compiler.cm`	structure `Compiler` (the visible compiler in one big structure)	always	no
`$smlnj/cm.cm` `$smlnj/cm/cm.cm`	compilation manager	always	auto
`$smlnj/cm/tools.cm`	API for extending CM with new tools	always	no
`$/mllex-tool.cm`	plugin library for class `mllex`	always	on demand
`$/lex-ext.cm`	plugin library for extension `.lex`	always	on demand
`$/mlyacc-tool.cm`	plugin library for class `mlyacc`	always	on demand
`$/grm-ext.cm`	plugin library for extension `.grm`	always	on demand
`$/mlburg-tool.cm`	plugin library for class `mlburg`	always	on demand
`$/burg-ext.cm`	plugin library for extension `.burg`	always	on demand
`$/noweb-tool.cm`	plugin library for class `noweb`	always	on demand
`$/nw-ext.cm`	plugin library for extension `.nw`	always	on demand
`$/make-tool.cm`	plugin library for class `make`	always	on demand
`$/shell-tool.cm`	plugin library for class `shell`	always	on demand
`$/dir-tool.cm`	plugin library for class `dir`	always	on demand

F.3 Libraries for SML/NJ compiler hackers

The following table lists libraries that provide access to the SML/NJ *bootstrap compiler*. The bootstrap compiler is a derivative of the compilation manager. In addition to being able to recompile SML/NJ for the "host" system there are also cross-compilers that can target all of SML/NJ's supported platforms.

name	description	installed	loaded
`$smlnj/cmb.cm` `$smlnj/cmb/current.cm`	bootstrap compiler for current architecture and OS	always	no
`$smlnj/cmb/alpha32-unix.cm`	bootstrap compiler for Alpha/Unix systems	always	no
`$smlnj/cmb/hppa-unix.cm`	bootstrap compiler for HP-PA/Unix systems	always	no
`$smlnj/cmb/ppc-macos.cm`	bootstrap compiler for PowerPC/Unix systems	always	no
`$smlnj/cmb/ppc-unix.cm`	bootstrap compiler for PowerPC/MacOS systems	always	no
`$smlnj/cmb/sparc-unix.cm`	bootstrap compiler for Sparc/Unix systems	always	no
`$smlnj/cmb/x86-unix.cm`	bootstrap compiler for IA32/Unix systems	always	no
`$smlnj/cmb/x86-win32.cm`	bootstrap compiler for IA32/Win32 systems	always	no
`$smlnj/compiler/alpha32.cm`	visible compiler with backend for Alpha-specific cross-compiler	always	no
`$smlnj/compiler/hppa.cm`	visible compiler with backend for HP-PA-specific cross-compiler	always	no
`$smlnj/compiler/ppc.cm`	visible compiler with backend for PowerPC-specific cross-compiler	always	no
`$smlnj/compiler/sparc.cm`	visible compiler with backend for Sparc-specific cross-compiler	always	no
`$smlnj/compiler/x86.cm`	visible compiler with backend for IA32-specific cross-compiler	always	no
`$smlnj/compiler/all.cm`	visible compiler, backends and bootstrap compilers for all architectures	always	no
`$/pgraph.cm`	definition of “portable dependency graph” data structure	always	no
`$/pgraph-util.cm`	utility routines for use with portable dependency graphs	optional	no

F.4 Internal libraries

For completeness, here is the list of other libraries that are part of SML/NJ’s implementation:

name	description	installed	loaded
`$SMLNJ-MLRISC/Lib.cm`	utility library for MLRISC backend	always	no
`$SMLNJ-MLRISC/Control.cm`	control facilities for MLRISC backend	always	no
`$SMLNJ-MLRISC/Graphs.cm`	control flow graphs for MLRISC backend	always	no
`$SMLNJ-MLRISC/Visual.cm`	visualization for MLRISC	always	no
`$SMLNJ-MLRISC/MLRISC.cm`	architecture-neutral core of MLRISC backend	always	no
`$SMLNJ-MLRISC/MLTREE.cm`	utility routines for dealing with mltree data structures	always	no
`$SMLNJ-MLRISC/ALPHA.cm`	Alpha-specific MLRISC backend	always	no
`$SMLNJ-MLRISC/HPPA.cm`	HP-PA-specific MLRISC backend	always	no
`$SMLNJ-MLRISC/PPC.cm`	PowerPC-specific MLRISC backend	always	no
`$SMLNJ-MLRISC/SPARC.cm`	Sparc-specific MLRISC backend	always	no
`$SMLNJ-MLRISC/IA32.cm`	IA32-specific MLRISC backend	always	no
`$/pickle-lib.cm`	utility library for compiler and CM	always	no
`$smlnj/viscomp/core.cm`	architecture-neutral core of compiler	always	no
`$smlnj/viscomp/alpha32.cm`	Alpha-specific part of compiler	always	no
`$smlnj/viscomp/hppa.cm`	HP-PA-specific part of compiler	always	no
`$smlnj/viscomp/ppc.cm`	PowerPC-specific part of compiler	always	no
`$smlnj/viscomp/sparc.cm`	Sparc-specific part of compiler	always	no
`$smlnj/viscomp/x86.cm`	IA32-specific part of compiler	always	no
`$smlnj/init/init.cmi`	initial "glue"; implementation of pervasive environment	always	no
`$smlnj/internal/cm-sig-lib.cm`	signatures `CM` and `CMB`	always	no
`$smlnj/internal/srcpath-lib.cm`	implementation of an internal "source path" abstraction used by the compilation manager	always	no
`$smlnj/internal/cm-lib.cm`	implementation of CM (not yet specialized to specific backends)	always	no
`$smlnj/internal/cm0.cm`	specialization of compilation manager to host compiler	always	no
`$smlnj/internal/intsys.cm`	root library, containing interactive system and glue for all the other parts	always	no

Libraries for the MLRISC backend have internal dependencies. However, when referring to a `$SMLNJ-MLRISC/`*lib*`.cm` library, all anchor names that correspond to these dependencies are still unbound. Thus, a client would have to provide its own bindings for them.[20] There is an analogous set of library names of the form `$smlnj/MLRISC/`*lib*`.cm` which provides the SML/NJ-specific bindings for all the anchors. In practice this means that if one refers to, e.g., `$smlnj/MLRISC/MLRISC.cm`, one will implicitly pick up `$smlnj/MLRISC/Graphs.cm`, and the `$Graphs.cm` anchor within the former will be resolved to the latter. On the other hand, when referring directly to the "raw" library `$SMLNJ-MLRISC/MLRISC.cm`, one still can (and must) provide one's own binding for `$Graphs.cm`.

G Exports of library `$smlnj/cm/tools.cm`

As described in section 12, it is possible to extend CM's set of available tools using the programming interface provided by `structure Tools`. This structure—together with its corresponding `signature TOOLS`—is exported by library `$smlnj/cm/tools.cm`. The same library also exports `structure Sharing`, `structure Version`, and a corresponding `signature VERSION`.

G.1 The public signature of `structure Tools`

```
signature TOOLS = sig

    type class = string
```

[20] The anchor names coincide with the *lib*`.cm` components of the above names.

```
type srcpath
type presrcpath

type rebindings = { anchor: string, value: presrcpath } list

val nativeSpec : srcpath -> string
val nativePreSpec : presrcpath -> string
val srcpath : presrcpath -> srcpath
val augment : presrcpath -> string list -> presrcpath

exception ToolError of { tool: string, msg: string }

type pathmaker = unit -> presrcpath

type fnspec = { name: string, mkpath: pathmaker }

datatype toolopt =
    STRING of fnspec
  | SUBOPTS of { name: string, opts: toolopts }
withtype toolopts = toolopt list

type tooloptcvt = toolopts option -> toolopts option

type spec = { name: string, mkpath: pathmaker,
              class: class option, opts: toolopts option, derived: bool }

type setup = string option * string option (* (pre, post) *)

type expansion =
     { smlfiles: (srcpath * Sharing.request * setup) list,
       cmfiles: (srcpath * Version.t option * rebindings) list,
       sources: (srcpath * { class: class, derived: bool}) list }

type partial_expansion = expansion * spec list

type rulefn = unit -> partial_expansion
type rulecontext = rulefn -> partial_expansion
type rule = { spec: spec,
              native2pathmaker: string -> pathmaker,
              context: rulecontext,
              defaultClassOf: fnspec -> class option }
            -> partial_expansion

val registerClass : class * rule -> unit

datatype classifier =
    SFX_CLASSIFIER of string -> class option
  | GEN_CLASSIFIER of { name: string, mkfname: unit -> string } ->
                      class option

val stdSfxClassifier : { sfx: string , class: class } -> classifier

datatype extensionStyle =
    EXTEND of (string * class option * tooloptcvt) list
  | REPLACE of string list * (string * class option * tooloptcvt) list

val extend : extensionStyle ->
             (string * toolopts option) ->
```

```
                 (string * class option * toolopts option) list

    val outdated : string -> string list * string -> bool

    val outdated' :
        string -> { src: string, wtn: string, tgt: string } -> bool

    val openTextOut : string -> TextIO.outstream

    val makeDirs : string -> unit

    val registerClassifier : classifier -> unit

    val parseOptions :
        { tool : string, keywords : string list, options : toolopts } ->
        { matches : string -> toolopts option, restoptions : string list }

    val say : string list -> unit
    val vsay : string list -> unit

    val mkCmdName : string -> string

    val registerStdShellCmdTool :
        { tool: string, class: string, suffixes: string list,
          cmdStdPath: string, extensionStyle: extensionStyle,
          template: string option, dflopts: toolopts }
        -> unit
end

structure Tools :> TOOLS
```

G.2 The public signature of `structure Version`

```
signature VERSION = sig

    type t

    val fromString : string -> t option
    val toString: t -> string
    val compare : t * t -> order

    val nextMajor : t -> t

    val zero: t
end

structure Version :> VERSION
```

G.3 The public signature of `structure Sharing`

```
structure Sharing : sig
    datatype request = PRIVATE | SHARED | DONTCARE
    datatype mode = SHARE of bool | DONTSHARE
end
```

References

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *3rd International Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[AM94] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Proc. SIGPLAN '94 Symp. on Prog. Language Design and Implementation*, volume 29, pages 13–23. ACM Press, June 1994.

[AMT89] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey, December 1989.

[ATW94] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM TOSEM*, 3(1):3–28, January 1994.

[BA97] Matthias Blume and Andrew W. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 112–124. ACM Press, June 1997.

[BA99] Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Trans. on Programming Languages and Systems*, 21(4):790–812, July 1999.

[Blu95] Matthias Blume. Standard ML of New Jersey compilation manager. Manual accompanying SML/NJ software, 1995.

[Blu99] Matthias Blume. Dependency analysis for Standard ML. *ACM Trans. on Programming Languages and Systems*, 21(4):813–846, July 1999.

[Fel79] S. I. Feldman. Make – a program for maintaining computer programs. In *Unix Programmer's Manual, Seventh Edition, Volume 2A*. Bell Laboratories, 1979.

[GG93] Florent Guillaume and Lal George. *ML-Burg Documentation*. AT&T Bell Laboratories, 1993. distributed with SML/NJ software.

[HLPR94a] Robert Harper, Peter Lee, Frank Pfenning, and Eugene Rollins. A Compilation Manager for Standard ML of New Jersey. In *1994 ACM SIGPLAN Workshop on ML and its Applications*, pages 136–147, 1994.

[HLPR94b] Robert Harper, Peter Lee, Frank Pfenning, and Eugene Rollins. Incremental recompilation for Standard ML of New Jersey. Technical Report CMU-CS-94-116, Department of Computer Science, Carnegie-Mellon University, February 1994.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1988.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994.

[RG99] John Reppy and Emden Gansner. Standard ML Basis Library. (not yet in paper form), 1999.

[TA90] David R. Tarditi and Andrew W. Appel. ML-Yacc, version 2.0. Distributed with Standard ML of New Jersey, April 1990.