Camlidl user's manual
Version 1.05

Xavier Leroy
INRIA Rocquencourt

July 13, 2004

1 Overview

Camlidl generates stub code for interfacing Caml with C (as described in chapter "Interfacing with C" of the Objective Caml reference manual[1]) from an IDL description of the C functions to be made available in Caml. Thus, Camlidl automates the most tedious task in interfacing C libraries with Caml programs. It can also be used to interface Caml programs with other languages, as long as those languages have a well-defined C interface.

In addition, Camlidl provides basic support for COM interfaces and components. It supports both using COM components (usually written in C++ or C) from Caml programs, and packaging Caml objects as COM components that can then be used from C++ or C.

1.1 What is IDL?

IDL stands for Interface Description Language. This is a generic term for a family of small languages that have been developed to provide type specifications for libraries written in C and C++. Those languages resembles C declarations (as found in C header files), with extra annotations to provide more precise types for the arguments and results of the functions.

The particular IDL used by Camlidl is inspired by Microsoft's IDL, which itself is an extension of the IDL found in DCE (The Open Group's Distributed Common Environment). The initial motivation for those IDLs was to automate the generation of stub code for remote procedure calls and network objects, where the arguments to the function are marshaled at the calling site, then sent across the network or through interprocess communications to a server process, which unmarshals the arguments, compute the function application, marshal the results, sends them back to the calling site, where they are unmarshaled and returned to the caller. IDLs were also found to be very useful for inter-language communications, since the same type information that guides the generation of marshaling stubs can be used to generate stubs to convert between the data representations of several languages.

1.2 What is COM?

COM is Microsoft's Common Object Model. It provides a set of programming conventions as well as system support for packaging C++ objects as executable components that can be used in other

[1] `http://caml.inria.fr/ocaml/htmlman/index.html`

programs, either by dynamic linking of the component inside the program, or through interprocess or internetwork communications between the program and a remote server. COM components implement one or several interfaces, (similar to Caml object types or Java interfaces) identified by unique 128-bit interface identifiers (IIDs). COM specifies a standard protocol for reference counting of components, and for asking a component which interfaces it implements.

While the full range of COM services and third-party components is available only on Microsoft's Windows operating systems, the basic COM conventions can also be used on Unix and other operating systems to exchange objects between Caml and C or C++. Of particular interest is the encapsulation of Caml objects as COM components, which can then be used inside larger C or C++ applications; those applications do not need to know anything about Caml: they just call the component methods as if they were C++ methods or C functions, without knowing that they are actually implemented in Caml.

For more information about COM, see for instance *Inside COM* by Dale Rogerson (Microsoft Press), or the Microsoft developer Web site[2].

2 IDL syntax

This section describes the syntax of IDL files. IDL syntax is very close to that of C declarations, with extra attributes between brackets adding information to the C types. The following example should give the flavor of the syntax:

```
int f([in,string] char * msg);
```

This reads: "`f` is a function taking a character string as input and returning an `int`".

2.1 Lexical conventions

Blanks. Space, newline, horizontal tabulation, carriage return, line feed and form feed are considered as blanks. Blanks are ignored, but they separate adjacent tokens.

Comments. Both C-style comments `/* ... */` and Java-style comments `// ...` are supported. C-style comments are introduced by `/*` and terminated by `*/`. Java-style comments are introduced by `//` and extend to the end of the line. Comments are treated as blank characters. Comments do not occur inside string or character literals. Nested C-style comments are not supported.

Identifiers. Identifiers have the same syntax as in C.

$$\textit{ident} \quad ::= \quad (\texttt{A}\ldots\texttt{Z} \mid \texttt{a}\ldots\texttt{z} \mid \texttt{\_})\ \{\texttt{A}\ldots\texttt{Z} \mid \texttt{a}\ldots\texttt{z} \mid \texttt{0}\ldots\texttt{9} \mid \texttt{\_}\}$$

[2] `http://msdn.microsoft.com`

Literals. Integer literals, character literals and string literals have the same syntax as in C.

$$
\begin{array}{rcl}
\textit{integer} & ::= & [\texttt{-}]\ \{\texttt{0}\ldots\texttt{9}\}\ [\texttt{-}]\ \texttt{0x}\ \{\texttt{0}\ldots\texttt{9} \mid \texttt{a}\ldots\texttt{f} \mid \texttt{A}\ldots\texttt{F}\}\ [\texttt{-}]\ \texttt{0}\ \{\texttt{0}\ldots\texttt{7}\} \\
\textit{character} & ::= & \texttt{'}\ (\textit{regular-char} \mid \textit{escape-char})\ \texttt{'} \\
\textit{string} & ::= & \texttt{"}\ \{\textit{regular-char} \mid \textit{escape-char}\}\ \texttt{"} \\
\textit{escape-char} & ::= & \texttt{\char`\\}\ (\texttt{b} \mid \texttt{n} \mid \texttt{r} \mid \texttt{t}) \\
 & \mid & \texttt{\char`\\}\ (\texttt{0}\ldots\texttt{7})\ [\texttt{0}\ldots\texttt{7}]\ [\texttt{0}\ldots\texttt{7}]
\end{array}
$$

UUID. Unique identifiers are composed of 16 hexadecimal digits, in groups of 8, 4, 4, 4 and 12, separated by dashes.

$$
\begin{array}{rcl}
\textit{uuid} & ::= & \textit{hex}^8\ \texttt{-}\ \textit{hex}^4\ \texttt{-}\ \textit{hex}^4\ \texttt{-}\ \textit{hex}^4\ \texttt{-}\ \textit{hex}^4\ \textit{hex}^4\ \textit{hex}^4 \\
\textit{hex} & ::= & \texttt{0}\ldots\texttt{9} \mid \texttt{a}\ldots\texttt{f} \mid \texttt{A}\ldots\texttt{F}
\end{array}
$$

2.2 Limited expressions

Limited expressions are similar to C expressions, with the omission of assignment operators (`=`, `+=`, etc), and the addition of the unsigned (logical) right shift operator `>>>`. Operators have the same precedences and associativities as in C. They are listed below in decreasing priority order.

$$
\begin{array}{rcl}
\textit{lexpr} & ::= & \textit{ident} \\
 & \mid & \textit{integer} \\
 & \mid & \textit{character} \\
 & \mid & \texttt{true} \\
 & \mid & \texttt{false} \\
 & \mid & \textit{string} \\
 & \mid & \textit{sizeof}\ \texttt{(}\ \textit{type-expr}\ \texttt{)} \\
 & \mid & \texttt{(}\ \textit{lexpr}\ \texttt{)} \\
 & \mid & \textit{lexpr}\ (\texttt{.} \mid \texttt{->})\ \textit{ident} \\
 & \mid & \texttt{(}\ \textit{type-expr}\ \texttt{)}\ \textit{lexpr} \\
 & \mid & (\texttt{\&} \mid \texttt{*} \mid \texttt{!} \mid \texttt{\~{}} \mid \texttt{-} \mid \texttt{+})\ \textit{lexpr} \\
 & \mid & \textit{lexpr}\ (\texttt{*} \mid \texttt{/} \mid \texttt{\%})\ \textit{lexpr} \\
 & \mid & \textit{lexpr}\ (\texttt{+} \mid \texttt{-})\ \textit{lexpr} \\
 & \mid & \textit{lexpr}\ (\texttt{<<} \mid \texttt{>>} \\
 & \mid & \texttt{>>>})\ \textit{lexpr} \\
 & \mid & \textit{lexpr}\ (\texttt{==} \mid \texttt{!=} \mid \texttt{>=} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{<}) \\
 & \mid & \textit{lexpr}\ (\texttt{\&} \mid \texttt{\^{}} \mid \texttt{|})\ \textit{lexpr} \\
 & \mid & \textit{lexpr}\ (\texttt{\&\&} \mid \texttt{||})\ \textit{lexpr} \\
 & \mid & \textit{lexpr}\ \texttt{?}\ \textit{lexpr}\ \texttt{:}\ \textit{lexpr}
\end{array}
$$

Constant limited expressions, written *const-lexpr* below, can only reference identifiers that are bound by the IDL `const` declaration.

2.3 Attributes

$$
\begin{array}{rcl}
\mathit{attributes} & ::= & \texttt{[}\ \mathit{attribute}\ \{\texttt{,}\ \mathit{attribute}\}\ \texttt{]} \\
\mathit{attribute} & ::= & \mathit{ident} \\
 & | & \mathit{ident}\ \texttt{(}\ [\mathit{lexpr}]\ \{\texttt{,}\ [\mathit{lexpr}]\}\ \texttt{)} \\
 & | & \mathit{ident}\ \texttt{(}\ \mathit{uuid}\ \texttt{)} \\
 & | & \mathit{attribute}\ \texttt{*} \\
 & | & \texttt{*}\ \mathit{attribute}
\end{array}
$$

Attribute lists are written in brackets `[...]`, and are always optional. Each attribute is identified by a name, and may carry optional arguments. Starred attributes apply to the element type of a pointer or array type, rather than to the pointer or array type itself. The following table summarizes the recognized attributes and their arguments.

<table>
<tr><th>Attribute</th><th>Context where it can appear</th></tr>
<tr><td>abstract</td><td>typedef</td></tr>
<tr><td>bigarray</td><td>array type</td></tr>
<tr><td>blocking</td><td>function declaration</td></tr>
<tr><td>camlint</td><td>int or long integer type</td></tr>
<tr><td>compare (fun-name)</td><td>typedef</td></tr>
<tr><td>c2ml (fun-name)</td><td>typedef</td></tr>
<tr><td>errorcheck (fun-name)</td><td>typedef</td></tr>
<tr><td>errorcode</td><td>typedef</td></tr>
<tr><td>finalize (fun-name)</td><td>typedef</td></tr>
<tr><td>fortran</td><td>array type with bigarray attribute</td></tr>
<tr><td>hash (fun-name)</td><td>typedef</td></tr>
<tr><td>ignore</td><td>any pointer type</td></tr>
<tr><td>in</td><td>function parameter</td></tr>
<tr><td>int_default (camlint | nativeint | int32 | int64)</td><td>interface</td></tr>
<tr><td>int32</td><td>int or long integer type</td></tr>
<tr><td>int64</td><td>int or long integer type</td></tr>
<tr><td>length_is (le_1 , le_2 , ...)</td><td>array type</td></tr>
<tr><td>long_default (camlint | nativeint | int32 | int64)</td><td>interface</td></tr>
<tr><td>managed</td><td>array type with bigarray attribute</td></tr>
<tr><td>ml2c (fun-name)</td><td>typedef</td></tr>
<tr><td>mlname(fun-or-field-name)</td><td>function declaration, struct field</td></tr>
<tr><td>mltype(" caml-type-expr ")</td><td>typedef</td></tr>
<tr><td>nativeint</td><td>int or long integer type</td></tr>
<tr><td>null_terminated</td><td>array of pointers</td></tr>
<tr><td>object</td><td>interface</td></tr>
<tr><td>out</td><td>function parameter</td></tr>
<tr><td>pointer_default (ref | unique | ptr)</td><td>interface</td></tr>
<tr><td>propget</td><td>function declaration</td></tr>
<tr><td>propput</td><td>function declaration</td></tr>
<tr><td>propputref</td><td>function declaration</td></tr>
<tr><td>ptr</td><td>any pointer type</td></tr>
<tr><td>ref</td><td>any pointer type</td></tr>
<tr><td>set</td><td>enum type</td></tr>
<tr><td>size_is (le_1 , le_2 , ...)</td><td>array type</td></tr>
<tr><td>string</td><td>character array or pointer</td></tr>
<tr><td>switch_is (le)</td><td>union type or pointer to union</td></tr>
<tr><td>switch_type (ty)</td><td>union or pointer to union</td></tr>
<tr><td>unique</td><td>any pointer, array, or bigarray type</td></tr>
<tr><td>uuid (uuid)</td><td>interface</td></tr>
</table>

2.4 Types and declarators

The declaration of an identifier along with its type is as in C: a type specification comes first, followed by the identifier possibly decorated with `*` and `[...]` to denote pointers and array types.

For instance, `int x` declares an identifier `x` of type `int`, while `int (*x)[]` declares an identifier `x` that is a pointer to an array of integers.

```
         type-spec ::= [unsigned | signed] (int | short | long | char | hyper | long long | __int64)
                     | byte
                     | float
                     | double
                     | boolean
                     | void
                     | ident
                     | wchar_t
                     | handle_t
                     | struct ident
                     | union ident
                     | enum ident
                     | struct-decl
                     | union-decl
                     | enum-decl
        declarator ::= {*} direct-declarator
 direct-declarator ::= ident
                     | ( declarator )
                     | direct-declarator [ [const-lexpr] ]
```

2.5 Structures, unions and enumerations

```
struct-decl ::= struct [ident] { {field-decl} }
 field-decl ::= attributes type-spec declarator {, declarator} ;
 union-decl ::= union [ident] { {union-case} }
              | union [ident] switch ( type-spec ident ) { {union-case} }
 union-case ::= {case ident :}+ [field-decl] ; default : [field-decl] ;
  enum-decl ::= enum [ident] { enum-case {, enum-case} [,] }
  enum-case ::= ident [= const-lexpr]
```

IDL `struct` declarations are like those of C, with the addition of optional attributes on each field. `union` declarations are also as in C, except that each case of an union must be labeled by one or several `case` *ident* `:`. The first form of union declaration assumes that the discriminant of the union is provided separately via a `switch_is` annotation on the union type, while the second form encapsulates the discriminant along with the union itself (like in Pascal's `record case of` construct).

2.6 Function declarations

function-decl ::= *attributes type-spec* {`*`} *ident* `(` *params* `)` {`quote` `(` *ident* `,` *string* `)`}

params ::= ϵ
| `void`
| *param* { `,` *param*}

param ::= *attributes type-spec declarator*

Function declarations are like in ANSI C, with the addition of attributes on each parameter and on the function itself. Parameters must be named. The optional *quote* statements following the declaration are user-provided calling sequences and deallocation sequences that replaces the default sequences in the `camlidl`-generated stub code for the function.

2.7 Constant definitions

constant-decl ::= `const` *attributes type-spec* {`*`} *ident* `=` *const-lexpr* `;`

A constant declaration associates a name to a limited expression. The limited expression can refer to constant names declared earlier, but cannot refer to other kinds of identifiers. The optional attributes influence the interpretation of the type specification, e.g. `const int x = 3` defines `x` with Caml type `int`, but `const [int64] long x = 5` defines `x` with Caml type `int64`.

2.8 IDL files

file ::= {*decl*}

decl ::= *function-decl* `;`
| *constant-decl* `;`
| *struct-decl* `;`
| *union-decl* `;`
| *enum-decl* `;`
| `typedef` *attributes type-spec declarator* { `,` *declarator*} `;`
| *attributes* `interface` *ident* [`:` *ident*] `{` {*decl*} `}`
| `struct` *ident* `;`
| `union` *ident* `;`
| `union switch` `(` *type-spec ident* `)` `;`
| *attributes* `interface` *ident* `;`
| `import` *string* `;`
| `quote` `(` [*ident* `,`] *string* `)`
| `cpp_quote` `(` *string* `)`

An IDL file is a sequence of IDL declarations. Declarations include function declarations, constant declarations, type declarations (structs, unions, enums, as well as a C-style `typedef` declaration to name a type expression), and interfaces.

An interface declaration gives a name and attributes to a collection of declarations. For interfaces with the `object` attribute, an optional super-interface can be provided, as in `interface` *intf* `:` *super-intf*. The name of the interface can be used as a type name in the remainder of the file.

Forward declarations of structs, unions and interfaces are supported in the usual C manner, by just giving the name of the struct, union or interface, but not its actual contents.

The `import` statement reads another IDL file and makes available its type and constant declarations in the remainder of the file. No code is generated for the functions and interfaces declared in the imported file. The same file can be imported several times, but is read in only the first time.

The `quote (` *ident* `,` *str* `)` diversion copies the string *str* verbatim to one of the files generated by the `camlidl` compiler. The *ident* determines the file where *str* is copied: it can be `ml` for the Caml implementation file (`.ml`), `mli` for the Caml interface file (`.mli`), `mlmli` for both Caml files, `h` for the C header file (`.h`), and `c` for the C source file containing the generated stub code (`.c` file). For backward compatibility, `cpp_quote (` *str* `)` is recognized as synonymous for `quote ( h ,` *str* `)`.

3 The Caml-IDL mapping

This section describes how IDL types, function declarations, and interfaces are mapped to Caml types, functions and classes.

3.1 Base types

IDL type *ty*	Caml type $[\![ty]\!]$
`byte`, `short`	`int`
`int`, `long` with `[camlint]` attribute	`int`
`int`, `long` with `[nativeint]` attribute	`nativeint`
`int`, `long` with `[int32]` attribute	`int32`
`int`, `long` with `[int64]` attribute	`int64`
`hyper`, `long long`, `__int64`	`int64`
`char`	`char`
`float`, `double`	`float`
`boolean`	`bool`

(For integer types, `signed` and `unsigned` variants of the same IDL integer type translate to the same Caml type.)

Depending on the attributes, the `int` and `long` integer types are converted to one of the Caml integer types `int`, `nativeint`, `int32`, or `int64`. Values of Caml type `int32` are exactly 32-bit wide and values of type `int64` are exactly 64-bit wide on all platforms. Values of type `nativeint` have the natural word size of the platform, and are large enough to accommodate any C `int` or `long int` without loss of precision. Values of Caml type `int` have the natural word size of the platform minus one bit of tag, hence the conversion from IDL types `int` and `long` loses the most significant bit on 32-bit platforms. On 64-bit platforms, the conversion from `int` is exact, but the conversion from `long` loses the most significant bit.

If no explicit integer attribute is given for an `int` or `long` type, the `int_default` or `long_default` attribute of the enclosing interface, if any, determines the kind of the integer. If no `int_default` or `long_default` attribute is in scope, the kind `camlint` is assumed, which maps IDL `int` and `long` types to the Caml `int` type.

3.2 Pointers

The mapping of IDL pointer types depends on their kinds. Writing $[\![ty]\!]$ for the Caml type corresponding to the IDL type ty, we have:

`[ref]` ty `*`	$\Rightarrow$	$[\![ty]\!]$
`[unique]` ty `*`	$\Rightarrow$	$[\![ty]\!]$ `option`
`[ptr]` ty `*`	$\Rightarrow$	$[\![ty]\!]$ `Com.opaque`

In other terms, IDL pointers of kind `ref` are ignored during the mapping: `[ref]` ty `*` is mapped to the same Caml type as ty. A pointer p to a C value c `=` `*`p is translated to the Caml value corresponding to c.

IDL pointers of kind `unique` are mapped to an `option` type. The option value is `None` for a null pointer, and `Some(`v`)` for a non-null pointer to a C value c that translates to the ML value v.

IDL pointers of kind `ptr` are mapped to a `Com.opaque` type. This is an abstract type that encapsulates the C pointer without attempting to convert it to an ML data structure.

IDL pointers of kind `ignore` denote struct fields and function parameters that need not be exposed in the Caml code. Those pointers are simply set to null when converting from Caml to C, and ignored when converting from C to Caml. They cannot occur elsewhere.

If no explicit pointer kind is given, the `pointer_default` attribute of the enclosing interface, if any, determines the kind of the pointer. If no `pointer_default` attribute is in scope, the kind `unique` is assumed.

3.3 Arrays

IDL arrays of characters that carry the `[string]` attribute are mapped to the Caml `string` type:

IDL type ty	Caml type $[\![ty]\!]$
`[string] char []`	`string`
`[string] unsigned char []`	`string`
`[string] signed char []`	`string`
`[string] byte []`	`string`

Caml string values are translated to standard null-terminated C strings. Be careful about embedded null characters in the Caml string, which will be recognized as end of string by C functions.

IDL arrays carrying the `[bigarray]` attribute are translated to Caml "big arrays", as described in the next section.

All other IDL arrays are translated to ML arrays:

ty `[]` $\Rightarrow$ $[\![ty]\!]$ `array`

For instance, `double []` becomes `float array`. Consequently, multi-dimensional arrays are translated to Caml arrays of arrays. For instance, `int [][]` becomes `int array array`.

If the `unique` attribute is given, the IDL array is translated to an ML option type:

`[string,unique] char []`	$\Rightarrow$	`string option`
`[unique]` ty `[]`	$\Rightarrow$	$[\![ty]\!]$ `array option`

As in the case of pointers of kind `unique`, the option value is `None` for a null C pointer, and `Some(`v`)` for a non-null C pointer to a C array that translates to the ML string or array v.

Conversion between a C array and an ML array proceed element by element. For the conversion from C to ML, the number of elements of the ML array is determined as follows (in the order presented):

- By the `length_is` attribute, if present.
- By the `size_is` attribute, if present.
- By the bound written in the array type, if any.
- By searching the first null element of the C array, if the `null_terminated` attribute is present.

For instance, C values of IDL type `[length_is(n)] double[]` are mapped to Caml `float array` of `n` elements. C values of IDL type `double[10]` are mapped to Caml `float array` of 10 elements.

The `length_is` and `size_is` attributes take as argument one or several limited expressions. Each expression applies to one dimension of the array. For instance, `[size_is(*dimx, *dimy)] double d[][]` specifies a matrix of `double` whose first dimension has size `*dimx` and the second has size `*dimy`.

3.4 Big arrays

IDL arrays of integers or floats that carry the `[bigarray]` attribute are mapped to one of the Caml `Bigarray` types: `Array1.t` for one-dimensional arrays, `Array2.t` for 2-dimensional arrays, `Array3.t` for 3-dimensional arrays, and `Genarray.t` for arrays of 4 dimensions or more.

If the `[fortran]` attribute is given, the big array is accessed from Caml using the Fortran conventions (array indices start at 1; column-major memory layout). By default, the big array is accessed from Caml using the C conventions (array indices start at 0; row-major memory layout).

If the `[managed]` attribute is given on a big array type that is result type or out parameter type of a function, Caml assumes that the corresponding C array was allocated using `malloc()`, and is not referenced anywhere else; then, the Caml garbage collector will free the C array when the corresponding Caml big array becomes unreachable. By default, Caml assumes that result or out C arrays are statically or permanently allocated, and keeps a pointer to them during conversion to Caml big arrays, and does not free them when the Caml bigarrays become unreachable.

Finally, the `[unique]` attribute applies to bigarrays as to arrays, that is, it maps a null C pointer to `None`, and a non-null C pointer p to `Some(`v`)` where v is the ML bigarray resulting from the translation of p.

3.5 Structs

IDL structs are mapped to Caml record types. The names and types of the IDL struct fields determine the names and types of the Caml record type:

`struct` s `{ ... ;` ty_i id_i `; ... }` becomes `type` s `= { ... ;` id_i `:` $[\![ty_i]\!]$ `; ... }`

Example: `struct s { int n; double d[4]; }` becomes `type s = {n: int; d: float array}`.

Exceptions to this rule are as follows:

- Fields of the IDL struct that are pointers with the `[ignore]` attribute do not appear in the Caml record type. Example: `struct s { double x,y; [ignore] void * data; }` becomes `type struct_s = {x : float; y: float}`. Those ignored pointer fields are set to `NULL` when converting from a Caml record to a C struct.
- Integer fields of the IDL struct that appear in a `length_is`, `size_is` or `switch_is` attribute of another field also do not appear in the Caml record type. (We call those fields *dependent* fields.) Example: `struct s { int idx; int len; [size_is(len)] double d[]; }` is translated to the Caml record type `type struct_s = {idx: int; d: float array}`. The value of `len` is recovered from the size of the Caml array `d`, and thus doesn't need to be represented explicitly in the Caml record.
- If, after elimination of ignored pointer fields and dependent fields as described above, the IDL struct has only one field *ty id*, we avoid creating a one-field Caml record type and translate the IDL struct type directly to the Caml type $[\![ty]\!]$. Example: `struct s { int len; [size_is(len)] double d[]; }` is translated to the Caml type abbreviation `type struct_s = double array`.
- The names of labels in the Caml record type can be changed by using the `mlname` attribute on struct field declarations. For instance,

  ```
  struct s { int n; [mlname(p)] int q; }
          becomes type s = { n : int; p : int }
  ```

- The Caml type system makes it difficult to use two record types defined in the same module and having some label names in common. Thus, if CamlIDL encounters two or more structs having identically-named fields, it prefixes the Caml label names by the names of the structs in order to distinguish them. For instance:

  ```
  struct s1 { int x; int y; }
  struct s2 { double x; double t; }
  struct s3 { int z; }
          becomes type s1 = { s1_x: int; s1_y: int }
                  and s2 = { s2_x: float; s2_t: float }
                  and s3 = { z: int }
  ```

  The labels for `s1` and `s2` have been prefixed by `s1_` and `s2_` respectively, to avoid ambiguity on the `x` label. However, the label `z` for `s3` is not prefixed, since it is not used elsewhere.

  The prefix added in front of multiply-defined labels is taken from the struct name, if any, and otherwise from the name of the nearest enclosing struct, union or typedef. For instance:

  ```
  typedef struct { int x; } t;
  struct s4 { struct { int x; } z; };
          becomes type t = { t_x: int }
                  and s4 = { z: struct_1 }
                  and struct_1 = { s4_x: int }
  ```

The "minimal prefixing" strategy described above is the default behavior of `camlidl`. If the `-prefix-all-labels` option is given, all record labels are prefixed, whether they occur several times or not. If the `-keep-labels` option is given, no automatic prefixing takes place; the naming of record labels is left entirely under the user's control, via `mlname` annotations.

3.6 Unions

IDL discriminated unions are translated to Caml sum types. Each case of the union corresponds to a constructor of the sum type. The constructor is constant if the union case has no associated field, otherwise has one argument corresponding to the union case field. If the union has a `default` case, an extra constructor `Default_`*unionname* is added to the Caml sum type, carrying an `int` argument (the value of the discriminating field), and possibly another argument corresponding to the default field. Examples:

```
union u1 { case A: int x; case B: case C: double d; case D: ; }
        becomes type u1 = A of int | B of float | C of float | D
union u2 { case A: int x; case B: double d; default: ; }
        becomes type u2 = A of int | B of float | Default_u of int
union u3 { case A: int x; default: double d; }
        becomes type u3 = A of int | Default_v of int * double
```

All IDL unions must be discriminated, either via the special syntax `union` *name* `switch(int` *discr*)..., or via the attribute `switch_is(`*discr*`)`, where *discr* is a C l-value built from other parameters of the current function, or other fields of the current `struct`. Both the discriminant and the case labels must be of an integer type. Unless a `default` case is given, the value of the discriminant must be one of the cases of the union.

3.7 Enums

IDL enums are translated to Caml enumerated types (sum types with only constant constructors). The names of the constructors are determined by the names of the enum labels. The values attached to the enum labels are ignored. Example: `enum e { A, B = 2, C = 4 }` becomes `type enum_e = A | B | C`.

The `set` attribute can be applied to a named enum to denote a bitfield obtained by logical "or" of zero, one or several labels of the enum. The corresponding ML value is a list of zero, one or several constructors of the Caml enumerated type. Consider for instance:

```
enum e { A = 1, B = 2, C = 4 };
typedef [set] enum e eset;
```

The Caml type `eset` is equal to `enum_e list`. The C integer 6 (= `B | C`) is translated to the ML list `[B; C]`. The ML list `[A; C]` is translated to the C integer `A | C`, that is `5`.

3.8 Type definitions

An IDL `typedef` statement is normally translated to a Caml type abbreviation. For instance, `typedef [string] char * str` becomes `type str = string`.

If the `abstract` attribute is given, a Caml abstract type is generated instead of a type abbreviation, thus hinding from Caml the representation of the type in question. For instance, `typedef [abstract] void * handle` becomes `type handle`. In this case, the IDL type in the `typedef` is ignored.

If the `mltype ( "` *caml-type-expr* `" )` attribute is given, the Caml type is made equal to *caml-type-expr*. This is often used in conjunction with the `ml2c` and `c2ml` attributes to implement custom translation of data structures between C and ML. For instance, `typedef [mltype("int list")] struct mylist_struct * mylist` becomes `type mylist = int list`.

If the `c2ml(` *funct-name* `)` *and*`ml2c`(funct-name) attributes are given, the user-provided C functions given as attributes will be called to perform Caml to C and C to Caml conversions for values of the typedef-ed type, instead of using the `camlidl`-generated conversion functions. This allows user-controlled translation of data structures. The prototypes of the conversion functions must be

```
value c2ml(ty * input);
void ml2c(value input, ty * output);
```

where *ty* is the name of the type defined by `typedef`. In other terms, the `c2ml` function is passed a reference to a *ty* and returns the corresponding Caml value, while the `ml2c` function is passed a Caml value as first argument and stores the corresponding C value in the *ty* reference passed as second argument.

If the `finalize(` *final-fn* `)` attribute is given in combination with the `abstract` attribute, the function *final-fn* is called when the Caml block representing a value of this typedef becomes unreachable from Caml and is reclaimed by the Caml garbage collector. Similarly, `compare(` *compare-fn* `)` and `hash(` *hash-fn* `)` attach a comparison function and a hashing function (respectively) to Caml values for this typedef. The comparison function is called when two Caml values of this typedef are compared using the generic comparisons `compare`, `=`, `<`, etc. The hashing function is called when `Hashtbl.hash` is applied to a Caml value of this typedef. The prototype of the finalization, comparison and hashing functions are:

```
value final-fn(ty * x);
int compare-fn(ty * x, ty * y);
long hash-fn(ty * x);
```

That is, their arguments are passed by reference. The comparison function must return an integer that is negative, zero, or positive depending on whether its first argument is smaller, equal or greater than its second argument. The hashing function returns a suitable hash value for its argument.

If the `errorcheck(` *fn* `)` attribute is provided for the `typedef` *ty*, the error checking function *fn* is called each time a function result of type *ty* is converted from C to Caml. The function can then check the *ty* value for values indicating an error condition, and raise the appropriate exception. If in addition the `errorcode` attribute is provided, the conversion from C to Caml is suppressed: values of type *ty* are only passed to *fn* for error checking, then discarded.

3.9 Functions

IDL function declarations are translated to Caml functions. The parameters and results of the Caml function are determined from those of the IDL function according to the following rules:

- First, dependent parameters (parameters that are `size_is`, `length_is` or `switch_is` of other parameters) as well as parameters that are ignored pointers are removed.
- The remaining parameters are split into Caml function inputs and Caml function outputs. Parameters with the `[in]` attribute are added to the inputs of the function. Parameters with the `[out]` attribute are added to the outputs of the function. Parameters with the `[in,out]` attribute are added both to the inputs and to the outputs of the function, unless they are of type string or big array, in which case they are added to the inputs of the function only. (The reason for this exception is that strings and big arrays are shared between Caml and C, thus allowing true `in,out` behavior on the Caml function parameter, while other data types are copied during Caml/C conversion, thus turning a C `in,out` parameter into a Caml `copy in, copy out` parameter, that is, one parameter and one result.)
- The return value of the IDL function is added to the outputs of the Caml function (in first position), unless it is of type `void` or of a type name that carries the `errorcode` attribute. In the latter two cases, the return value of the IDL function is not transmitted to Caml.
- The Caml function is then given type in_1 `->` ... `->` in_p `->` out_1 `*` ... `*` out_q where $in_1 \ldots in_p$ are the types of its inputs and $out_1 \ldots out_q$ are the types of its outputs. If there are no inputs, a `unit` parameter is added. If there are no outputs, a `unit` result is added.

Examples:

```
int f([in] double x, [in] double y)           f : float -> float -> int
```

Two `double` input, one `int` output

```
void g([in] int x)                            g : int -> unit
```

One `int` input, no output

```
int h()                                       h : unit -> int
```

No input, one `int` result

```
void i([in] int x, [out] double * y)          i : int -> double
```

One `int` input, one `double` output (as an `out` parameter)

```
int j([in] int x, [out] double * y)           j : int -> int * double
```

One `int` input, one `int` output (in the result), one `double` output (as an `out` parameter)

```
void k([in,out,ref] int * x)                  k : int -> int
```

The `in,out` parameter is both one `int` input and one `int` output.

```
HRESULT l([in] int x, [out] int * res1, [out] int * res2)
                                              l : int -> int * int
```

HRESULT is a predefined type with the errorcode attribute, hence it is ignored. It remains one int input and two int outputs (out parameters)

```
void m([in] int len, [in,size_is(len)] double d[])
                                                 m : float array -> int
```

len is a dependent parameter, hence is ignored. The only input is the double array

```
void n([in] int inputlen, [out] int * outputlen,
       [in,out,size_is(inputlen),length_is(*outputlen)] double d[])
                                                 n : float array -> float array
```

The two parameters inputlen and outputlen are dependent, hence ignored. The double array is both an input and an output.

```
void p([in] int dimx, [in] int dimy,
       [in,out,bigarray,size_is(dimx,dimy)] double d[][])
p : (float, Bigarray.float64_elt, Bigarray.c_layout) Bigarray.Array2.t -> unit
```

The two parameters dimx and dimy are dependent (determined from the dimensions of the big array argument), hence ignored. The two-dimensional array d, although marked [in,out], is a big array, hence passed as an input that will be modified in place by the C function p. The Caml function has no outputs.

Error checking: For every output that is of a named type with the errorcheck(*fn*) attribute, the error checking function *fn* is called after the C function returns. That function is assumed to raise a Caml exception if it finds an output denoting an error.

Custom calling and deallocation sequences: The IDL declaration for a function can optionally specify a custom calling sequence and/or a custom deallocation sequence, via *quote* clauses following the function declaration:

function-decl ::= *attributes type-spec* {*} *ident* (*params*) {quote (*ident* , *string*)}

The general shape of a camlidl-generated stub function is as follows:

```
value caml_wrapper(value camlparam1, ..., value camlparamK)

  /* Convert the function parameters from Caml to C */
  param1 = ...;
  ...
  paramN = ...;
  /* Call the C function 'ident' */
  _res = ident(param1, ..., paramN);
  /* Convert the function result and out parameters to Caml values */
```

```
camlres = ...;
/* Return result to Caml */
return camlres;
```

A `quote(call, `*string*` )` clause causes the C statements in *string* to be inserted in the generated stub code instead of the default calling sequence `_res = ident(param1, ..., paramN)`. Thus, the statements in *string* find the converted parameters in local variables that have the same names as the parameters in the IDL declaration, and should leave the result of the function, if any, in the local variable named `_res`.

A `quote(dealloc, `*string*` )` clause causes the C statements in *string* to be inserted in the generated stub code just before the stub function returns, hence after the conversion of the C function results to Caml values. Again, the statements in *string* have access to the function result in the local variable named `_res`, and to out parameters in local variables having the same names as the parameters. Since the function results and out parameters have already been converted to Caml values, the code in *string* can safely deallocate the data structures they point to.

Custom calling sequences are typically used to rearrange or combine function parameters, and to perform extra error checks on the arguments and results. For instance, the Unix `write` system call can be specified in IDL as follows:

```
int write([in] int fd,
          [in,string,length_is(len)] char * data,
          [in] int len,
          [in] int ofs,
          [in] int towrite)
  quote(call,
    " /* Validate the arguments */
      if (ofs < 0 || ofs + towrite >= len) failwith(\"write\");
      /* Perform the write */
      _res = write(fd, data + ofs, towrite);
      /* Validate the result */
      if (_res == -1) failwith(\"write\"); ");
```

Custom deallocation sequences are useful to free data structures dynamically allocated and returned by the C function. For instance, a C function `f` that returns a `malloc`-ed string can be specified in IDL as follows:

```
[string] char * f([in] int x)
  quote(dealloc, "free(_res); ");
```

If the string is returned as an `out` parameter instead, we would write:

```
void f ([in] int x, [out, string*] char ** str)
  quote(dealloc, "free(*str); ");
```

Blocking functions: A function can be given the attribute `blocking` to indicate that it may block on an input/output operation. The generated code will then allow other Caml threads to execute concurrently with the operation.

3.10 Interfaces

IDL interfaces that do not have the `object` attribute are essentially ignored. That is, the declarations contained in the interface are processed as if they occurred at the top-level of the IDL file. The `pointer_default`, `int_default` and `long_default` attributes to the interface can be used to specify the default pointer kind and integer mappings for the declarations contained in the interface. Other attributes, as well as the name of the super-interface if any, are ignored.

IDL interfaces having the `object` attribute specify COM-style object interfaces. The function declarations contained in the interface specify the methods of the COM interface. Other kinds of declarations (type declarations, `import` statements, etc) are treated as if they occurred at the top-level of the IDL file. An optional super-interface can be given, in which case the COM interface implements the methods of the super-interface in addition to those specified in the IDL interface. Example:

```
[object, uuid(...)] interface IA { typedef int t; int f(int x); }
[object] interface IB : IA { import "foo.idl"; void g([string] char * s); }
```

This defines a type `t` and imports the file `foo.idl` as usual. In addition, two interfaces are declared: `IA`, containing one method `f` from `int` to `int`, and `IB`, containing two methods, `f` from `int` to `int` and `g` from `string` to `unit`.

The definition of an object interface *i* generates the following Caml definitions:

- An abstract type *i* identifying the interface. COM interfaces of type *i* are represented in Caml with type *i* `Com.interface`.
- If a super-interface *s* is given, a conversion function *s*`_of_`*i* of type *i* `Com.interface ->` *s* `Com.interface`.
- If the `uuid(`*iid*`)` attribute is given, a value `iid_`*i* of type *i* `Com.iid` holding the given interface identifier.
- A Caml class *i*`_class`, with the same methods as the COM interface.
- A function `use_`*i* of type *i* `Com.interface ->` *i*`_class`, to transform a COM object into a Caml object. This allows the methods of the COM object to be invoked from Caml.
- A function `make_`*i* of type `#`*i*`_class ->` *i* `Com.interface`, to transform a Caml object into a COM object with interface *i*. This allows the methods of the Caml object to be invoked from any COM client.

Example: in the `IA` and `IB` example above, the following Caml definitions are generated for `IA`:

```
type iA
val iid_iA : iA Com.iid
class iA_class : iA Com.interface -> object method f : int -> int end
val use_iA : iA Com.interface -> iA_class
val make_iA : #iA_class -> iA Com.interface
```

For `IB`, we get:

```
type iB
val iA_of_iB : iB Com.interface -> iA Com.interface
class iB_class :
  iB Com.interface -> object inherit iA_class method g : string -> unit end
val use_iB : iB Com.interface -> iB_class
val make_iB : #iB_class -> iB Com.interface
```

Error handling in interfaces: Conventionally, methods of COM interfaces always return a result of type `HRESULT` that says whether the method succeeded or failed, and in the latter case returns an error code to its caller.

When calling an interface method from Caml, if the method returns an `HRESULT` denoting failure, the exception `Com.Error` is raised with a message describing the error. Successful `HRESULT` return values are ignored. To make them available to Caml, `camlidl` defines the types `HRESULT_bool` and `HRESULT_int`. If those types are used as return types instead of `HRESULT`, failure results are mapped to `Com.Error` exceptions as before, but successful results are mapped to the Caml types `bool` and `int` respectively. (For `HRESULT_bool`, the `S_OK` result is mapped to `true` and other successful results are mapped to `false`. For `HRESULT_int`, the low 16 bits of the result code are returned as a Caml `int`.)

When calling a Caml method from a COM client, any exception that escapes the Caml method is mapped back to a failure `HRESULT`. A textual description of the uncaught exception is saved using `SetLastError`, and can be consulted by the COM client using `GetLastError` (this is the standard convention for passing extended error information in COM).

If the IDL return type of the method is not one of the `HRESULT` types, any exception escaping the Caml method aborts the whole program after printing a description of the exception. Hence, programmers of Caml components should either use `HRESULT` as result type, or make very sure that all exceptions are properly caught by the method.

4 Using `camlidl`

4.1 Overview

The `camlidl` stub generator is invoked as follows:

`camlidl` *options file1*`.idl` *file2*`.idl ...`

For each file *f*`.idl` given on the command line, `camlidl` generates the following files:

- A Caml interface file *f*`.mli` that defines the Caml view of the IDL file. It contains Caml definitions for the types declared in the IDL file, as well as declarations for the functions and the interfaces.
- A Caml implementation file *f*`.ml` that implements the *f*`.mli` file.
- A C source file *f*`_stubs.c` that contains the stub functions for converting between C and Caml data representations.
- If the `-header` option is given, a C header file *f*`.h` containing C declarations for the types declared in the IDL file.

The generated `.ml` and `.c` files must be compiled and linked with the remainder of the Caml program.

4.2 Options

The following command-line options are recognized by `camlidl`.

`-cpp`
: Pre-process the source IDL files with the C preprocessor. This option is set by default.

`-D` *symbol=value*
: Define a preprocessor symbol. The option `-D`*symbol=value* is passed to the C preprocessor. The *value* can be omitted, as in `-D` *symbol*, and defaults to `1`.

`-header`
: Generate a C header file *f*`.h` containing C declarations for the types and functions declared in the IDL file *f*`.c`.

`-I` *dir*
: Add the directory *dir* to the list of directories searched for `.idl` files, as given on the command line or recursively loaded by `import` statements.

`-keep-labels`
: Keep the Caml names of record labels as specified in the IDL file. Do not prefix them with the name of the enclosing struct, even if they appear in several struct definitions.

`-nocpp`
: Suppresses the pre-processing of source IDL files.

`-no-include`
: By default, `camlidl` emits a `#include "`*f*`.h"` statement in the file *f*`.c` containing the generated C code. The *f*`.h` header file being included is either the one generated by `camlidl -header`, or generated by another tool (such as Microsoft's `midl` compiler) from the IDL file, or hand-written. The *f*`.h` file is assumed to provide all C type declarations needed for compiling the stub code.

  The `-no-include` option suppresses the automatic inclusion of the *f*`.h` file. The IDL file should then include the right header files and provide the right type declarations via `quote` statements.

`-prefix-all-labels`
: Prefix all Caml names of record labels with the name of the enclosing struct. The default is to prefix only those labels that could cause ambiguity because they appear in several struct definitions.

`-prepro` *preprocessing-command*
: Set the command that is executed to pre-process the source IDL files. The default is the C preprocessor.

4.3 The `camlidldll` script

Under Windows, a `bash` script called `camlidldll` is provided to automate the construction of a DLL containing a COM component written in Caml.

The script `camlidldll` accepts essentially the same command-line arguments and options as the `ocamlc` compiler. (It also accepts `.tlb` type library files on the command-line; see section 6.3, "Dispatch interfaces", for more information on type libraries.) It produces a DLL file that encapsulates the Caml and C object files given on the command line.

Use `regsvr32 /s` *file*`.dll` to record the components in the system registry once it is compiled to a DLL.

5 Module `Com`: run-time library for COM components

`type 'a interface`

The type of COM components implementing interface `'a`

`type 'a iid`

The type of the interface identifier for interface `'a`

`type clsid`

The type of component identifiers

`type 'a opaque`

The type representing opaque pointers to values of type `'a`. Opaque pointers are pointers with attribute `ptr` in IDL files.

`exception Error of int * string * string`

Exception raised to report Com errors. The arguments are `Error(errcode, who, what)`. `errcode` is the Com error code (`HRESULT` code) with the high bit clear. `who` identifies the function or method that raised the exception. `what` is a message explaining the cause of the error.

`val initialize : unit -> unit`

Initialize the COM library. Must be called once before using any function in this module. `Com.initialize` can be called several times, provided that `Com.uninitialize` is called an equal number of times before the program exits.

`val uninitialize : unit -> unit`

Terminate the COM library.

`val query_interface : 'a interface -> 'b iid -> 'b interface`

`Com.query_interface comp iid` asks the component `comp` whether it supports the interface identified by `iid`. If yes, it returns the corresponding interface of the component. If not, it raises `Com.Error`.

`type iUnknown`

The type of the interface `IUnknown`, from which all other interfaces derive.

`type iDispatch`

The type of the interface `IDispatch`, from which all dispatch interfaces derive.

`val iUnknown_of : 'a interface -> iUnknown interface`

Return the `IUnknown` interface of the given component. This operation never fails, since all components support the `IUnknown` interface.

`val combine : 'a interface -> 'b interface -> 'a interface`

Combine the interfaces of two components. `Com.combine c1 c2` returns a component that supports the union of the interfaces supported by `c1` and `c2`. When queried for an interface, the resulting component delegates its implementation to `c1` if `c1` implements that interface, and otherwise delegates its implementation to `c2`.

```
val clsid : string -> clsid
```

Parse the string representation of a component identifier (`hex8-hex4-hex4-hex4-hex12`, where `hexN` represents `N` hexadecimal digits).

```
val create_instance : clsid -> 'a iid -> 'a interface
```

`Com.create_instance clsid iid` creates an instance of the component identified by `clsid`, and returns its `iid` interface. The implementation of the component is searched in the registry; if the component is implemented in a DLL, the DLL is loaded in memory if necessary; if the component is implemented in a separate server process, the server is started if necessary. Raise `Com.Error` if the component `clsid` cannot be found, or if it does not support interface `iid`.

```
type 'a component_factory =
  { create : unit -> 'a interface;
    clsid : clsid;
    friendly_name : string;
    ver_ind_prog_id : string;
    prog_id : string }
```

Informations required for registering a Caml implementation of a component. `create` is a function that returns a fresh instance of the component. `clsid` is the component identifier. `friendly_name` is a short description of the component (for information only). `ver_ind_prog_id` and `prog_id` are symbolic names for the component. By convention, `prog_id` is `ver_ind_prog_id` plus a version number at the end, i.e. `ver_ind_prog_id` is `"MyCamlComponent"` while `prog_id` is `"MyCamlComponent.3"`.

```
val register_factory : 'a component_factory -> unit
```

Register a Caml implementation of a component. `Com.register_factory f` stores the component factory `f` in the registry. Other programs can then create instances of the component by calling `CreateInstance` from C and C++ or `Com.create_instance` from Caml.

```
type hRESULT_int = int
type hRESULT_bool = bool
type bSTR = string
```

The Caml types corresponding to the IDL types `HRESULT_int`, `HRESULT_bool` and `BSTR`, respectively.

6 Hints on writing IDL files

6.1 Writing an IDL file for a C library

When writing an IDL file for a C library that doesn't have an IDL interface already, the include files for that library are a good starting point: just copy the relevant type and functin declarations to the IDL file, then annotate them with IDL attributes to describe more precisely their actual behavior. The documentation of the library must be read carefully to determine the mode of function parameters (`in`, `out`, `inout`), the actual sizes of arrays, etc.

The type definitions in the IDL file need not correspond exactly with those in the include files. Often, a cleaner Caml interface can be obtained by omitting irrelevant struct fields, or changing their types. For instance, the Unix library functions for reading library entries may use the following structure:

```
struct dirent {
    long int d_ino;
    __off_t d_off;
    unsigned short int d_reclen;
    unsigned char d_type;
    char d_name[256];
};
```

Of those fields, only `d_name` and `d_ino` are of interest to the user; the other fields are internal information for the library functions, are not specified in the POSIX specs, and therefore must not be used. Thus, in the IDL file, you should declare:

```
struct dirent {
    long int d_ino;
    char d_name[256];
};
```

Thus, the Caml code will have `type struct_dirent = {d_ino: int; d_name: string}` as desired. However, the generated stub code, being compiled against the "true" definition of `struct dirent`, will find those two fields at the correct offsets in the actual struct.

Special attention must be paid to integer fields or variables. By default, integer IDL types are mapped to the Caml type `int`, which is convenient to use in Caml code, but loses one bit when converting from a C `long` integer, and may lose one bit (on 32-bit platforms) when converting from a C `int` integer. When the range of values represented by the C integer is small enough, this loss is acceptable. Otherwise, you should use the attributes `nativeint`, `int32` or `int64` so that integer IDL types are mapped to one of the Caml boxed integer types. (We recommend that you use `int32` or `int64` for integers that are specified as being exactly 32 bit wide or 64 bit wide, and `nativeint` for unspecified `int` or `long` integers.)

Yet another possibility is to declare certain integer fields or variables as `double` in the IDL file, so that they are represented by `float` in Caml, and all 32 bits of the integer are preserved in Caml. For instance, the Unix function to get the current type is declared as

```
time_t time(time_t * t);
```

where `time_t` is usually defined as `long`. We can nonetheless pretend (in the IDL file) that `time` returns a double:

```
double time() quote(" _res = time(NULL); ");
```

This way, `time` will have the Caml type `unit -> float`. Again, the stub code "knows" that `time` actually returns an integer, and therefore will insert the right integer-float coercions.

6.2 Sharing IDL files between MIDL and CamlIDL

The Microsoft software development kit provides a number of IDL files describing various libraries and components. In its current state, `camlidl` cannot exploit those files directly: they use many (often poorly documented) Microsoft IDL features that are not implemented yet in `camlidl`; symmetrically, `camlidl` introduces several new annotations that are not recognized by Microsoft's `midl` compiler. So, significant editing work on the IDL files is required.

The C preprocessor can be used to alleviate the `camlidl`-`midl` incompatibilities: `camlidl` defines the preprocessor symbol `CAMLIDL` when preprocessing its input files, while `midl` does not. Hence, one can bracket incompatible definitions in `#ifdef CAMLIDL ... #else ... #endif`. Along these lines, a C preprocessor header file, `camlidlcompat.h`, is provided: it uses `#define` to remove `camlidl`-specific attributes when compiling with `midl`, and to remove `midl`-specific attributes when compiling with `camlidl`. Thus, an IDL file compatible with both `midl` and `camlidl` would look like this:

```
#include <camlidlcompat.h>

#ifndef CAMLIDL
import "unknwn.idl";    // imports specific to MIDL
import "oaidl.idl";
#endif
import "mymodule.idl";  // imports common to MIDL and CamlIDL

typedef [abstract,marshal_as(int)] void * ptr;

...

#ifndef CAMLIDL
[...] library MyTypeLib {
  importlib("stdole32.tlb");
  [...] coclass MyComponent { [default] interface IX; }
}
#endif
```

Notice that since `camlidl` doesn't handle type libraries, the type library part of an `midl` file must be enclosed in `#ifndef CAMLIDL`.

6.3 Dispatch interfaces and type libraries

A dispatch interface, in COM lingo, is an interface that supports dynamic, interpreted dispatch of method interfaces. This form of interpreted dispatch is used by Visual Basic and other scripting languages to perform calls to methods of COM components.

CamlIDL provides minimal support for dispatch interfaces. To equip a Caml component with a dispatch interface (thus making it callable from Visual Basic), you need to do the following:

1. Use `IDispatch` instead of `IUnknown` as the super-interface of the component's interfaces.

2. Write a type library for your component and compile it using `midl`. A type library is a run-time representation of the interfaces supported by an object. The `midl` compiler can generate a type library from the IDL description of the component, enriched with some special-purpose declarations (the `library` and `coclass` statements). Refer to the documentation of `midl` for more information.

3. Pass the type library files (`.tlb` files) generated by `midl` as extra arguments to `camlidldll` when generating the DLL for your Caml component.

7 Release notes

Here are some caveats and open issues that apply to the current release.

Deallocation of function results and `out` parameters: If a C function dynamically allocates some of its outputs (either returned or stored in `out` parameters), its IDL declaration must contain a `quote(dealloc,` *string* `)` clause to properly free the space occupied by those outputs after they have been converted to Caml. Otherwise, memory leaks will occur. (The only exception is results and output parameters of type `[bigarray,managed]` *ty*`[]`, where the Caml garbage collector takes care of deallocation.)

This does not conform to the MIDL and COM specifications, which say that space for `out` data structures must be allocated with `CoTaskMemAlloc` by the callee, and automatically freed using `CoTaskMemFree` by the generated stub code. (The specs don't say what happens with the return value of the function.) However, there are many functions in Win32 (not to mention the Unix world) that do not follow this convention, and return data structures (e.g. strings) that are statically allocated, or require special deallocation functions. Hence, `camlidl` leaves deallocation of outputs entirely under user control.

Allocation and deallocation of `in,out` parameters: For `in,out` parameters, the MIDL/COM rules are that the caller (the stub code) should allocate the inputs, the callee should free them and allocate again its outputs, and the caller should free the outputs. As explained above, `camlidl`-generated stubs don't automatically free the outputs. Worse, the inputs passed to the functions are allocated partially on the stack and partially in the heap (using `CoTaskMemAlloc`), so the callee may perform an incorrect free on a stack-allocated argument. The best thing to do is avoid `in,out` parameters entirely, and split them into one `in` and one `out` parameter.

Reference-counting of COM interfaces: Caml finalized objects are used to call `Release` automatically on COM interfaces that become unreachable. The reference counting of interfaces passed as `in` and `out` parameters is correctly implemented. However, `in,out` parameters that are interfaces are not correctly handled. Again, avoid `in,out` parameters.

COM support: The support for COM is currently quite small. COM components registered in the system registry can be imported via `Com.create_instance`. Components written in Caml can be exported as DLLs, but not yet as standalone servers. Preliminary support for dispatch interfaces is available, however many of the data types used in the Automation framework are not supported yet (e.g. `SAFEARRAY`).