

CIL API Documentation (version 1.3.5)

August 8, 2006

Contents

1	Module Pretty : Utility functions for pretty-printing.	1
2	Module Errormsg : Utility functions for error-reporting	7
3	Module Clist : Utilities for managing "concatenable lists" (clists).	9
4	Module Stats : Utilities for maintaining timing statistics	10
5	Module Cil : CIL API Documentation.	11
6	Module Formatcil : An Interpreter for constructing CIL constructs	57
7	Module Alpha : ALPHA conversion	58

1 Module Pretty : Utility functions for pretty-printing.

The major features provided by this module are

- An `fprintf`-style interface with support for user-defined printers
- The printout is fit to a width by selecting some of the optional newlines
- Constructs for alignment and indentation
- Print ellipsis starting at a certain nesting depth
- Constructs for printing lists and arrays

Pretty-printing occurs in two stages:

- Construct a `Pretty.doc[1]` object that encodes all of the elements to be printed along with alignment specifiers and optional and mandatory newlines
- Format the `Pretty.doc[1]` to a certain width and emit it as a string, to an output stream or pass it to a user-defined function

The formatting algorithm is not optimal but it does a pretty good job while still operating in linear time. The original version was based on a pretty printer by Philip Wadler which turned out to not scale to large jobs.

API

`type doc`

The type of unformatted documents. Elements of this type can be constructed in two ways. Either with a number of constructor shown below, or using the `Pretty.dprintf[1]` function with a `printf`-like interface. The `Pretty.dprintf[1]` method is slightly slower so we do not use it for large jobs such as the output routines for a compiler. But we use it for small jobs such as logging and error messages.

Constructors for the `doc` type.

`val nil : doc`

Constructs an empty document

`val (++) : doc -> doc -> doc`

Concatenates two documents. This is an infix operator that associates to the left.

`val concat : doc -> doc -> doc`

`val text : string -> doc`

A document that prints the given string

`val num : int -> doc`

A document that prints an integer in decimal form

`val real : float -> doc`

A document that prints a real number

`val chr : char -> doc`

A document that prints a character. This is just like `Pretty.text[1]` with a one-character string.

`val line : doc`

A document that consists of a mandatory newline. This is just like `(text "\n")`. The new line will be indented to the current indentation level, unless you use `Pretty.leftflush[1]` right after this.

`val leftflush : doc`

Use after a `Pretty.line[1]` to prevent the indentation. Whatever follows next will be flushed left. Indentation resumes on the next line.

`val break : doc`

A document that consists of either a space or a line break. Also called an optional line break. Such a break will be taken only if necessary to fit the document in a given width. If the break is not taken a space is printed instead.

```

val align : doc
    Mark the current column as the current indentation level. Does not print anything. All
    taken line breaks will align to this column. The previous alignment level is saved on a stack.

val unalign : doc
    Reverts to the last saved indentation level.

val mark : doc
    Mark the beginning of a markup section. The width of a markup section is considered 0 for
    the purpose of computing indentation

val unmark : doc
    The end of a markup section

    Syntactic sugar
val indent : int -> doc -> doc
    Indents the document. Same as ((text " ") ++ align ++ doc ++ unalign), with the
    specified number of spaces.

val markup : doc -> doc
    Prints a document as markup. The marked document cannot contain line breaks or
    alignment constructs.

val seq : sep:doc -> doit:('a -> doc) -> elements:'a list -> doc
    Formats a sequence. sep is a separator, doit is a function that converts an element to a
    document.

val docList : ?sep:doc -> ('a -> doc) -> unit -> 'a list -> doc
    An alternative function for printing a list. The unit argument is there to make this function
    more easily usable with the Pretty.dprintf[1] interface. The first argument is a separator,
    by default a comma.

val d_list : string -> (unit -> 'a -> doc) -> unit -> 'a list -> doc
    sm: Yet another list printer. This one accepts the same kind of printing function that
    Pretty.dprintf[1] does, and itself works in the dprintf context. Also accepts a string as the
    separator since that's by far the most common.

val docArray : ?sep:doc ->
    (int -> 'a -> doc) -> unit -> 'a array -> doc
    Formats an array. A separator and a function that prints an array element. The default
    separator is a comma.

val docOpt : ('a -> doc) -> unit -> 'a option -> doc
    Prints an 'a option with None or Some

val d_int32 : int32 -> doc

```

Print an int32

```
val f_int32 : unit -> int32 -> doc
val d_int64 : int64 -> doc
val f_int64 : unit -> int64 -> doc
module MakeMapPrinter :
  functor (Map : sig
    type key
    type 'a t
    val fold : (key -> 'a -> 'b -> 'b) ->
      'a t -> 'b -> 'b
  end) -> sig

    val docMap :
      ?sep:Pretty.doc ->
      (Map.key -> 'a -> Pretty.doc) -> unit -> 'a Map.t -> Pretty.doc

      Format a map, analogous to docList.

    val d_map :
      ?dmaplet:(Pretty.doc -> Pretty.doc -> Pretty.doc) ->
      string ->
      (unit -> Map.key -> Pretty.doc) ->
      (unit -> 'a -> Pretty.doc) -> unit -> 'a Map.t -> Pretty.doc

      Format a map, analogous to d_list.

  end

  Format maps.

module MakeSetPrinter :
  functor (Set : sig
    type elt
    type t
    val fold : (elt -> 'a -> 'a) ->
      t -> 'a -> 'a
  end) -> sig

    val docSet :
      ?sep:Pretty.doc -> (Set.elt -> Pretty.doc) -> unit -> Set.t -> Pretty.doc

      Format a set, analogous to docList.

    val d_set :
      string -> (unit -> Set.elt -> Pretty.doc) -> unit -> Set.t -> Pretty.doc
```

Format a set, analogous to `d_list`.

end

Format sets.

```
val insert : unit -> doc -> doc
```

A function that is useful with the `printf`-like interface

```
val dprintf : ('a, unit, doc, doc) format4 -> 'a
```

This function provides an alternative method for constructing `doc` objects. The first argument for this function is a format string argument (of type `('a, unit, doc) format`; if you insist on understanding what that means see the module `Printf`). The format string is like that for the `printf` function in C, except that it understands a few more formatting controls, all starting with the `@` character.

See the `gprintf` function if you want to pipe the result of `dprintf` into some other functions.

The following special formatting characters are understood (these do not correspond to arguments of the function):

- `@[` Inserts an `Pretty.align[1]`. Every format string must have matching `Pretty.align[1]` and `Pretty.unalign[1]`.
- `@]` Inserts an `Pretty.unalign[1]`.
- `@!` Inserts a `Pretty.line[1]`. Just like `"\n"`
- `@?` Inserts a `Pretty.break[1]`.
- `@<` Inserts a `Pretty.mark[1]`.
- `@>` Inserts a `Pretty.unmark[1]`.
- `@^` Inserts a `Pretty.leftflush[1]` Should be used immediately after `@!` or `"\n"`.
- `@@` : inserts a `@` character

In addition to the usual `printf` `%` formatting characters the following two new characters are supported:

- `%t` Corresponds to an argument of type `unit -> doc`. This argument is invoked to produce a document
- `%a` Corresponds to **two** arguments. The first of type `unit -> 'a -> doc` and the second of type `'a`. (The extra `unit` is do to the peculiarities of the built-in support for format strings in Ocaml. It turns out that it is not a major problem.) Here is an example of how you use this:

```
dprintf "Name=%s, SSN=%7d, Children=@[%a@]\n"
      pers.name pers.ssn (docList (chr ',' ++ break) text)
      pers.children
```

The result of `dprintf` is a `Pretty.doc[1]`. You can format the document and emit it using the functions `Pretty.fprint[1]` and `Pretty.sprint[1]`.

```
val gprintf : (doc -> 'a) -> ('b, unit, doc, 'a) format4 -> 'b
```

Like `Pretty.dprintf[1]` but more general. It also takes a function that is invoked on the constructed document but before any formatting is done. The type of the format argument means that 'a is the type of the parameters of this function, unit is the type of the first argument to %a and %t formats, doc is the type of the intermediate result, and 'b is the type of the result of `gprintf`.

```
val fprintf : Pervasives.out_channel -> width:int -> doc -> unit
    Format the document to the given width and emit it to the given channel
```

```
val sprintf : width:int -> doc -> string
    Format the document to the given width and emit it as a string
```

```
val fprintf :
    Pervasives.out_channel -> ('a, unit, doc) Pervasives.format -> 'a
    Like Pretty.dprintf[1] followed by Pretty.fprint[1]
```

```
val printf : ('a, unit, doc) Pervasives.format -> 'a
    Like Pretty.fprintf[1] applied to stdout
```

```
val eprintf : ('a, unit, doc) Pervasives.format -> 'a
    Like Pretty.fprintf[1] applied to stderr
```

```
val withPrintDepth : int -> (unit -> unit) -> unit
    Invokes a thunk, with printDepth temporarily set to the specified value
```

The following variables can be used to control the operation of the printer

```
val printDepth : int Pervasives.ref
    Specifies the nesting depth of the align/unalign pairs at which everything is replaced with ellipsis
```

```
val printIndent : bool Pervasives.ref
    If false then does not indent
```

```
val fastMode : bool Pervasives.ref
    If set to true then optional breaks are taken only when the document has exceeded the given width. This means that the printout will look more ragged but it will be faster
```

```
val flushOften : bool Pervasives.ref
    If true then it flushes after every print
```

```
val countNewLines : int Pervasives.ref
    Keep a running count of the taken newlines. You can read and write this from the client code if you want
```

```
val auto_printer : string -> 'a
    A function that when used at top-level in a module will direct the pa_prttype module to generate automatically the printing functions for a type
```

2 Module Errormsg : Utility functions for error-reporting

```
val logChannel : Pervasives.out_channel Pervasives.ref
    A channel for printing log messages

val debugFlag : bool Pervasives.ref
    If set then print debugging info

val verboseFlag : bool Pervasives.ref
val warnFlag : bool Pervasives.ref
    Set to true if you want to see all warnings.

exception Error
    Error reporting functions raise this exception

val error : ('a, unit, Pretty.doc, unit) format4 -> 'a
    Prints an error message of the form Error: .... Use in conjunction with s, for example:
    E.s (E.error ... ).

val bug : ('a, unit, Pretty.doc, unit) format4 -> 'a
    Similar to error except that its output has the form Bug: ...

val unimp : ('a, unit, Pretty.doc, unit) format4 -> 'a
    Similar to error except that its output has the form Unimplemented: ...

val s : 'a -> 'b
    Stop the execution by raising an Error.

val hadErrors : bool Pervasives.ref
    This is set whenever one of the above error functions are called. It must be cleared manually

val warn : ('a, unit, Pretty.doc, unit) format4 -> 'a
    Like Errormsg.error[2] but does not raise the Errormsg.Error[2] exception. Return type is
    unit.

val warnOpt : ('a, unit, Pretty.doc, unit) format4 -> 'a
    Like Errormsg.warn[2] but optional. Printed only if the Errormsg.warnFlag[2] is set

val log : ('a, unit, Pretty.doc, unit) format4 -> 'a
    Print something to logChannel

val logg : ('a, unit, Pretty.doc, unit) format4 -> 'a
    same as Errormsg.log[2] but do not wrap lines

val null : ('a, unit, Pretty.doc, unit) format4 -> 'a
```

Do not actually print (i.e. print to /dev/null)

```
val pushContext : (unit -> Pretty.doc) -> unit
```

Registers a context printing function

```
val popContext : unit -> unit
```

Removes the last registered context printing function

```
val showContext : unit -> unit
```

Show the context stack to stderr

```
val withContext : (unit -> Pretty.doc) -> ('a -> 'b) -> 'a -> 'b
```

To ensure that the context is registered and removed properly, use the function below

```
val newline : unit -> unit
```

```
val newHline : unit -> unit
```

```
val getPosition : unit -> int * string * int
```

```
val getHPosition : unit -> int * string
```

high-level position

```
val setHLine : int -> unit
```

```
val setHFile : string -> unit
```

```
val setCurrentLine : int -> unit
```

```
val setCurrentFile : string -> unit
```

```
type location = {
```

```
  file : string ;
```

The file name

```
  line : int ;
```

The line number

```
  hfile : string ;
```

The high-level file name, or "" if not present

```
  hline : int ;
```

The high-level line number, or 0 if not present

```
}
```

Type for source-file locations

```
val d_loc : unit -> location -> Pretty.doc
```

```
val d_hloc : unit -> location -> Pretty.doc
```

```
val getLocation : unit -> location
```

```
val parse_error : string -> 'a
```

```
val locUnknown : location
```

An unknown location for use when you need one but you don't have one


```

val readingFromStdin : bool Pervasives.ref
    Records whether the stdin is open for reading the goal *

val startParsing : ?useBasename:bool -> string -> Lexing.lexbuf
val startParsingFromString :
    ?file:string -> ?line:int -> string -> Lexing.lexbuf
val finishParsing : unit -> unit

```

3 Module Clist : Utilities for managing "concatenable lists" (clists).

We often need to concatenate sequences, and using lists for this purpose is expensive. This module provides routines to manage such lists more efficiently. In this model, we never do cons or append explicitly. Instead we maintain the elements of the list in a special data structure. Routines are provided to convert to/from ordinary lists, and carry out common list operations.

```

type 'a clist =
  | CList of 'a list
      The only representation for the empty list. Try to use sparingly.
  | CConsL of 'a * 'a clist
      Do not use this a lot because scanning it is not tail recursive
  | CConsR of 'a clist * 'a
  | CSeq of 'a clist * 'a clist
      We concatenate only two of them at this time. Neither is the empty clist. To be sure
      always use append to make these

  The clist datatype. A clist can be an ordinary list, or a clist preceded or followed by an
  element, or two clists implicitly appended together

val toList : 'a clist -> 'a list
    Convert a clist to an ordinary list

val fromList : 'a list -> 'a clist
    Convert an ordinary list to a clist

val single : 'a -> 'a clist
    Create a clist containing one element

val empty : 'a clist
    The empty clist

val append : 'a clist -> 'a clist -> 'a clist
    Append two clists

val checkBeforeAppend : 'a clist -> 'a clist -> bool

```

A useful check to assert before an append. It checks that the two lists are not identically the same (Except if they are both empty)

```
val length : 'a clist -> int
    Find the length of a clist

val map : ('a -> 'b) -> 'a clist -> 'b clist
    Map a function over a clist. Returns another clist

val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b clist -> 'a
    A version of fold_left that works on clists

val iter : ('a -> unit) -> 'a clist -> unit
    A version of iter that works on clists

val rev : ('a -> 'a) -> 'a clist -> 'a clist
    Reverse a clist. The first function reverses an element.

val docCList :
    Pretty.doc -> ('a -> Pretty.doc) -> unit -> 'a clist -> Pretty.doc
    A document for printing a clist (similar to docList)
```

4 Module Stats : Utilities for maintaining timing statistics

```
val reset : bool -> unit
    Resets all the timings. Invoke with "true" if you want to switch to using the hardware
    performance counters from now on. You get an exception if there are not performance
    counters available

exception NoPerfCount

val has_performance_counters : unit -> bool
    Check if we have performance counters

val sample_pentium_perfcount_20 : unit -> int
    Sample the current cycle count, in megacycles.

val sample_pentium_perfcount_10 : unit -> int
    Sample the current cycle count, in kilocycles.

val time : string -> ('a -> 'b) -> 'a -> 'b
    Time a function and associate the time with the given string. If some timing information is
    already associated with that string, then accumulate the times. If this function is invoked
    within another timed function then you can have a hierarchy of timings
```

```

val repeatTime : float -> string -> ('a -> 'b) -> 'a -> 'b
    repeatTime is like time but runs the function several times until the total running time is
    greater or equal to the first argument. The total time is then divided by the number of times
    the function was run.

val print : Pervasives.out_channel -> string -> unit
    Print the current stats preceded by a message

val lookupTime : string -> float
    Return the cumulative time of all calls to Stats.time[4] and Stats.repeatTime[4] with the
    given label.

val lastTime : float Pervasives.ref
    Time a function and set lastTime to the time it took

val timethis : ('a -> 'b) -> 'a -> 'b

```

5 Module Cil : CIL API Documentation.

An html version of this document can be found at <http://manju.cs.berkeley.edu/cil>.

```

val initCIL : unit -> unit
    Call this function to perform some initialization. Call if after you have set Cil.msvcMode[5].

val cilVersion : string
    This are the CIL version numbers. A CIL version is a number of the form M.m.r (major,
    minor and release)

val cilVersionMajor : int
val cilVersionMinor : int
val cilVersionRevision : int
    This module defines the abstract syntax of CIL. It also provides utility functions for traversing
    the CIL data structures, and pretty-printing them. The parser for both the GCC and MSVC front-
    ends can be invoked as Frontc.parse: string -> unit -> Cil.file[5]. This function must be
    given the name of a preprocessed C file and will return the top-level data structure that describes
    a whole source file. By default the parsing and elaboration into CIL is done as for GCC source. If
    you want to use MSVC source you must set the Cil.msvcMode[5] to true and must also invoke the
    function Frontc.setMSVCMode: unit -> unit.

```

The Abstract Syntax of CIL

The top-level representation of a CIL source file (and the result of the parsing and elaboration). Its main contents is the list of global declarations and definitions. You can iterate over the globals in a Cil.file[5] using the following iterators: Cil.mapGlobals[5], Cil.iterGlobals[5] and Cil.foldGlobals[5]. You can also use the Cil.dummyFile[5] when you need a Cil.file[5] as a placeholder. For each global item CIL stores the source location where it appears (using the type Cil.location[5])

```

type file = {
    mutable fileName : string ;

```

The complete file name

```
mutable globals : global list ;
```

List of globals as they will appear in the printed file

```
mutable globinit : fundec option ;
```

An optional global initializer function. This is a function where you can put stuff that must be executed before the program is started. This function, is conceptually at the end of the file, although it is not part of the globals list. Use `Cil.getGlobInit[5]` to create/get one.

```
mutable globinitcalled : bool ;
```

Whether the global initialization function is called in main. This should always be false if there is no global initializer. When you create a global initialization CIL will try to insert code in main to call it. This will not happen if your file does not contain a function called "main"

```
}
```

Top-level representation of a C source file

```
type comment = location * string
```

Globals. The main type for representing global declarations and definitions. A list of these form a CIL file. The order of globals in the file is generally important.

```
type global =
```

```
| GType of typeinfo * location
```

A typedef. All uses of type names (through the `TNamed` constructor) must be preceded in the file by a definition of the name. The string is the defined name and always not-empty.

```
| GCompTag of compinfo * location
```

Defines a struct/union tag with some fields. There must be one of these for each struct/union tag that you use (through the `TComp` constructor) since this is the only context in which the fields are printed. Consequently nested structure tag definitions must be broken into individual definitions with the innermost structure defined first.

```
| GCompTagDecl of compinfo * location
```

Declares a struct/union tag. Use as a forward declaration. This is printed without the fields.

```
| GEnumTag of enuminfo * location
```

Declares an enumeration tag with some fields. There must be one of these for each enumeration tag that you use (through the `TEnum` constructor) since this is the only context in which the items are printed.

```
| GEnumTagDecl of enuminfo * location
```

Declares an enumeration tag. Use as a forward declaration. This is printed without the items.

```
| GVarDecl of varinfo * location
```

A variable declaration (not a definition). If the variable has a function type then this is a prototype. There can be several declarations and at most one definition for a given variable. If both forms appear then they must share the same varinfo structure. A prototype shares the varinfo with the fundec of the definition. Either has storage Extern or there must be a definition in this file

| GVar of varinfo * initinfo * location

A variable definition. Can have an initializer. The initializer is updateable so that you can change it without requiring to recreate the list of globals. There can be at most one definition for a variable in an entire program. Cannot have storage Extern or function type.

| GFun of fundec * location

A function definition.

| GAsm of string * location

Global asm statement. These ones can contain only a template

| GPragma of attribute * location

Pragmas at top level. Use the same syntax as attributes

| GText of string

Some text (printed verbatim) at top level. E.g., this way you can put comments in the output.

A global declaration or definition

Types. A C type is represented in CIL using the type `Cil.typ[5]`. Among types we differentiate the integral types (with different kinds denoting the sign and precision), floating point types, enumeration types, array and pointer types, and function types. Every type is associated with a list of attributes, which are always kept in sorted order. Use `Cil.addAttribute[5]` and `Cil.addAttributes[5]` to construct list of attributes. If you want to inspect a type, you should use `Cil.unrollType[5]` or `Cil.unrollTypeDeep[5]` to see through the uses of named types.

CIL is configured at build-time with the sizes and alignments of the underlying compiler (GCC or MSVC). CIL contains functions that can compute the size of a type (in bits) `Cil.bitsSizeOf[5]`, the alignment of a type (in bytes) `Cil.alignOf_int[5]`, and can convert an offset into a start and width (both in bits) using the function `Cil.bitsOffset[5]`. At the moment these functions do not take into account the `packed` attributes and pragmas.

type typ =

| TVoid of attributes

Void type. Also predefined as `Cil.voidType[5]`

| TInt of ikind * attributes

An integer type. The kind specifies the sign and width. Several useful variants are predefined as `Cil.intType[5]`, `Cil.uintType[5]`, `Cil.longType[5]`, `Cil.charType[5]`.

| TFloat of fkind * attributes

A floating-point type. The kind specifies the precision. You can also use the predefined constant `Cil.doubleType[5]`.

| TPtr of typ * attributes

Pointer type. Several useful variants are predefined as `Cil.charPtrType[5]`, `Cil.charConstPtrType[5]` (pointer to a constant character), `Cil.voidPtrType[5]`, `Cil.intPtrType[5]`

| TArray of `typ * exp option * attributes`

Array type. It indicates the base type and the array length.

| TFun of `typ * (string * typ * attributes) list option * bool`
* `attributes`

Function type. Indicates the type of the result, the name, type and name attributes of the formal arguments (`None` if no arguments were specified, as in a function whose definition or prototype we have not seen; `Some []` means void). Use `Cil.argsToList[5]` to obtain a list of arguments. The boolean indicates if it is a variable-argument function. If this is the type of a varinfo for which we have a function declaration then the information for the formals must match that in the function's sformals. Use `Cil.setFormals[5]`, or `Cil.setFunctionType[5]`, or `Cil.makeFormalVar[5]` for this purpose.

| TNamed of `typeinfo * attributes`

The use of a named type. Each such type name must be preceded in the file by a `GType` global. This is printed as just the type name. The actual referred type is not printed here and is carried only to simplify processing. To see through a sequence of named type references, use `Cil.unrollType[5]` or `Cil.unrollTypeDeep[5]`. The attributes are in addition to those given when the type name was defined.

| TComp of `compinfo * attributes`

The most delicate issue for C types is that recursion that is possible by using structures and pointers. To address this issue we have a more complex representation for structured types (struct and union). Each such type is represented using the `Cil.compinfo[5]` type. For each composite type the `Cil.compinfo[5]` structure must be declared at top level using `GCompTag` and all references to it must share the same copy of the structure. The attributes given are those pertaining to this use of the type and are in addition to the attributes that were given at the definition of the type and which are stored in the `Cil.compinfo[5]`.

| TEnum of `enuminfo * attributes`

A reference to an enumeration type. All such references must share the `enuminfo` among them and with a `GEnumTag` global that precedes all uses. The attributes refer to this use of the enumeration and are in addition to the attributes of the enumeration itself, which are stored inside the `enuminfo`

| TBuiltin_va_list of `attributes`

This is the same as the gcc's type with the same name

There are a number of functions for querying the kind of a type. These are `Cil.isIntegralType[5]`, `Cil.isArithmeticType[5]`, `Cil.isPointerType[5]`, `Cil.isFunctionType[5]`, `Cil.isArrayType[5]`.

There are two easy ways to scan a type. First, you can use the `Cil.existsType[5]` to return a boolean answer about a type. This function is controlled by a user-provided function that is queried for each type that is used to construct the current type. The function can specify whether to terminate the scan with a boolean result or to continue the scan for the nested types.

The other method for scanning types is provided by the visitor interface (see `Cil.cilVisitor[5]`).

If you want to compare types (or to use them as hash-values) then you should use instead type signatures (represented as `Cil.typsig[5]`). These contain the same information as types but canonicalized such that simple Ocaml structural equality will tell whether two types are equal. Use `Cil.typeSig[5]` to compute the signature of a type. If you want to ignore certain type attributes then use `Cil.typeSigWithAttrs[5]`.

```
type ikind =
  | IChar
      char
  | ISChar
      signed char
  | IUChar
      unsigned char
  | IInt
      int
  | IUInt
      unsigned int
  | IShort
      short
  | IUShort
      unsigned short
  | ILong
      long
  | IULong
      unsigned long
  | ILongLong
      long long (or _int64 on Microsoft Visual C)
  | IULongLong
      unsigned long long (or unsigned _int64 on Microsoft Visual C)
  Various kinds of integers

type fkind =
  | FFloat
      float
  | FDouble
      double
  | FLongDouble
      long double
  Various kinds of floating-point numbers
```

Attributes.

type attribute =

| Attr of string * attrparam list

An attribute has a name and some optional parameters. The name should not start or end with underscore. When CIL parses attribute names it will strip leading and ending underscores (to ensure that the multitude of GCC attributes such as `const`, `__const` and `__const__` all mean the same thing.)

type attributes = attribute list

Attributes are lists sorted by the attribute name. Use the functions `Cil.addAttribute[5]` and `Cil.addAttributes[5]` to insert attributes in an attribute list and maintain the sortedness.

type attrparam =

| AInt of int

An integer constant

| AStr of string

A string constant

| ACons of string * attrparam list

Constructed attributes. These are printed `foo(a1,a2,...,an)`. The list of parameters can be empty and in that case the parentheses are not printed.

| ASizeOf of typ

A way to talk about types

| ASizeOfE of attrparam

| ASizeOfS of typsig

Replacement for ASizeOf in type signatures. Only used for attributes inside typsig.

| AAlignOf of typ

| AAlignOfE of attrparam

| AAlignOfS of typsig

| AUnOp of unop * attrparam

| ABinOp of binop * attrparam * attrparam

| ADot of attrparam * string

a.foo *

| AStar of attrparam

a

| AAddrOf of attrparam

&a *

| AIndex of attrparam * attrparam

a1a2

| AQuestion of attrparam * attrparam * attrparam

a1 ? a2 : a3 *

The type of parameters of attributes

Structures. The `Cil.compinfo[5]` describes the definition of a structure or union type. Each such `Cil.compinfo[5]` must be defined at the top-level using the `GCompTag` constructor and must be shared by all references to this type (using either the `TComp` type constructor or from the definition of the fields).

If all you need is to scan the definition of each composite type once, you can do that by scanning all top-level `GCompTag`.

Constructing a `Cil.compinfo[5]` can be tricky since it must contain fields that might refer to the host `Cil.compinfo[5]` and furthermore the type of the field might need to refer to the `Cil.compinfo[5]` for recursive types. Use the `Cil.mkCompInfo[5]` function to create a `Cil.compinfo[5]`. You can easily fetch the `Cil.fieldinfo[5]` for a given field in a structure with `Cil.getCompField[5]`.

```
type compinfo = {
  mutable cstruct : bool ;
    True if struct, False if union

  mutable cname : string ;
    The name. Always non-empty. Use Cil.compFullName[5] to get the full name of a
    comp (along with the struct or union)

  mutable ckey : int ;
    A unique integer. This is assigned by Cil.mkCompInfo[5] using a global variable in the
    Cil module. Thus two identical structs in two different files might have different keys.
    Use Cil.copyCompInfo[5] to copy structures so that a new key is assigned.

  mutable cfields : fieldinfo list ;
    Information about the fields. Notice that each fieldinfo has a pointer back to the host
    compinfo. This means that you should not share fieldinfo's between two compinfo's

  mutable cattr : attributes ;
    The attributes that are defined at the same time as the composite type. These
    attributes can be supplemented individually at each reference to this compinfo using
    the TComp type constructor.

  mutable cdefined : bool ;
    This boolean flag can be used to distinguish between structures that have not been
    defined and those that have been defined but have no fields (such things are allowed in
    gcc).

  mutable creferenced : bool ;
    True if used. Initially set to false.
}
```

The definition of a structure or union type. Use `Cil.mkCompInfo[5]` to make one and use `Cil.copyCompInfo[5]` to copy one (this ensures that a new key is assigned and that the fields have the right pointers to parents.).

Structure fields. The `Cil.fieldinfo[5]` structure is used to describe a structure or union field. Fields, just like variables, can have attributes associated with the field itself or associated with the type of the field (stored along with the type of the field).

```

type fieldinfo = {
  mutable fcomp : compinfo ;
    The host structure that contains this field. There can be only one compinfo that
    contains the field.

  mutable fname : string ;
    The name of the field. Might be the value of Cil.missingFieldName[5] in which case
    it must be a bitfield and is not printed and it does not participate in initialization

  mutable ftype : typ ;
    The type

  mutable fbitfield : int option ;
    If a bitfield then ftype should be an integer type and the width of the bitfield must be
    0 or a positive integer smaller or equal to the width of the integer type. A field of
    width 0 is used in C to control the alignment of fields.

  mutable fattr : attributes ;
    The attributes for this field (not for its type)

  mutable flocc : location ;
    The location where this field is defined
}

```

Information about a struct/union field

Enumerations. Information about an enumeration. This is shared by all references to an enumeration. Make sure you have a `GEnumTag` for each of these.

```

type enuminfo = {
  mutable ename : string ;
    The name. Always non-empty.

  mutable eitems : (string * exp * location) list ;
    Items with names and values. This list should be non-empty. The item values must be
    compile-time constants.

  mutable eattr : attributes ;
    The attributes that are defined at the same time as the enumeration type. These
    attributes can be supplemented individually at each reference to this enuminfo using
    the TEnum type constructor.

  mutable ereferenced : bool ;
    True if used. Initially set to false
}

```

Information about an enumeration

Enumerations. Information about an enumeration. This is shared by all references to an enumeration. Make sure you have a `GEnumTag` for each of these.

```

type typeinfo = {
  mutable tname : string ;

```

The name. Can be empty only in a `GType` when introducing a composite or enumeration tag. If empty cannot be referred to from the file

```
mutable ttype : typ ;
```

The actual type. This includes the attributes that were present in the typedef

```
mutable treferenced : bool ;
```

True if used. Initially set to false

```
}
```

Information about a defined type

Variables. Each local or global variable is represented by a unique `Cil.varinfo[5]` structure. A global `Cil.varinfo[5]` can be introduced with the `GVarDecl` or `GVar` or `GFun` globals. A local `varinfo` can be introduced as part of a function definition `Cil.fundec[5]`.

All references to a given global or local variable must refer to the same copy of the `varinfo`. Each `varinfo` has a globally unique identifier that can be used to index maps and hashtables (the name can also be used for this purpose, except for locals from different functions). This identifier is constructor using a global counter.

It is very important that you construct `varinfo` structures using only one of the following functions:

- `Cil.makeGlobalVar[5]` : to make a global variable
- `Cil.makeTempVar[5]` : to make a temporary local variable whose name will be generated so that to avoid conflict with other locals.
- `Cil.makeLocalVar[5]` : like `Cil.makeTempVar[5]` but you can specify the exact name to be used.
- `Cil.copyVarinfo[5]`: make a shallow copy of a `varinfo` assigning a new name and a new unique identifier

A `varinfo` is also used in a function type to denote the list of formals.

```
type varinfo = {
```

```
  mutable vname : string ;
```

The name of the variable. Cannot be empty. It is primarily your responsibility to ensure the uniqueness of a variable name. For local variables `Cil.makeTempVar[5]` helps you ensure that the name is unique.

```
  mutable vtype : typ ;
```

The declared type of the variable.

```
  mutable vattr : attributes ;
```

A list of attributes associated with the variable.

```
  mutable vstorage : storage ;
```

The storage-class

```
  mutable vglob : bool ;
```

True if this is a global variable

```

mutable vinline : bool ;
    Whether this varinfo is for an inline function.

mutable vdecl : location ;
    Location of variable declaration.

mutable vid : int ;
    A unique integer identifier. This field will be set for you if you use one of the
    Cil.makeFormalVar[5], Cil.makeLocalVar[5], Cil.makeTempVar[5],
    Cil.makeGlobalVar[5], or Cil.copyVarinfo[5].

mutable vaddrof : bool ;
    True if the address of this variable is taken. CIL will set these flags when it parses C,
    but you should make sure to set the flag whenever your transformation create AddrOf
    expression.

mutable vreferenced : bool ;
    True if this variable is ever referenced. This is computed by removeUnusedVars. It is
    safe to just initialize this to False
}
    Information about a variable.

type storage =
| NoStorage
    The default storage. Nothing is printed

| Static
| Register
| Extern
    Storage-class information

```

Expressions. The CIL expression language contains only the side-effect free expressions of C. They are represented as the type `Cil.exp[5]`. There are several interesting aspects of CIL expressions:

Integer and floating point constants can carry their textual representation. This way the integer 15 can be printed as 0xF if that is how it occurred in the source.

CIL uses 64 bits to represent the integer constants and also stores the width of the integer type. Care must be taken to ensure that the constant is representable with the given width. Use the functions `Cil.kinteger[5]`, `Cil.kinteger64[5]` and `Cil.integer[5]` to construct constant expressions. CIL predefines the constants `Cil.zero[5]`, `Cil.one[5]` and `Cil.mone[5]` (for -1).

Use the functions `Cil.isConstant[5]` and `Cil.isInteger[5]` to test if an expression is a constant and a constant integer respectively.

CIL keeps the type of all unary and binary expressions. You can think of that type qualifying the operator. Furthermore there are different operators for arithmetic and comparisons on arithmetic types and on pointers.

Another unusual aspect of CIL is that the implicit conversion between an expression of array type and one of pointer type is made explicit, using the `StartOf` expression constructor (which is not printed). If you apply the `AddrOf` constructor to an lvalue of type T then you will be getting an expression of type `TPtr(T)`.

You can find the type of an expression with `Cil.typeOf[5]`.

You can perform constant folding on expressions using the function `Cil.constFold[5]`.

```
type exp =
| Const of constant
    Constant

| Lval of lval
    Lvalue

| SizeOf of typ
    sizeof(<type>). Has unsigned int type (ISO 6.5.3.4). This is not turned into a
    constant because some transformations might want to change types

| SizeOfE of exp
    sizeof(<expression>)

| SizeOfStr of string
    sizeof(string_literal). We separate this case out because this is the only instance in
    which a string literal should not be treated as having type pointer to character.

| AlignOf of typ
    This corresponds to the GCC __alignof_. Has unsigned int type

| AlignOfE of exp

| UnOp of unop * exp * typ
    Unary operation. Includes the type of the result.

| BinOp of binop * exp * exp * typ
    Binary operation. Includes the type of the result. The arithmetic conversions are
    made explicit for the arguments.

| CastE of typ * exp
    Use Cil.mkCast[5] to make casts.

| AddrOf of lval
    Always use Cil.mkAddrOf[5] to construct one of these. Apply to an lvalue of type T
    yields an expression of type TPtr(T). Use Cil.mkAddrOrStartOf[5] to make one of
    these if you are not sure which one to use.

| StartOf of lval
    Conversion from an array to a pointer to the beginning of the array. Given an lval of
    type TArray(T) produces an expression of type TPtr(T). Use Cil.mkAddrOrStartOf[5]
    to make one of these if you are not sure which one to use. In C this operation is
    implicit, the StartOf operator is not printed. We have it in CIL because it makes the
    typing rules simpler.

    Expressions (Side-effect free)
```

Constants.

```
type constant =
| CInt64 of int64 * ikind * string option
```

Integer constant. Give the `ikind` (see ISO9899 6.1.3.2) and the textual representation, if available. (This allows us to print a constant as, for example, `0xF` instead of `15`.) Use `Cil.integer[5]` or `Cil.kinteger[5]` to create these. Watch out for integers that cannot be represented on 64 bits. OCAML does not give Overflow exceptions.

| **CStr of string**

String constant. The escape characters inside the string have been already interpreted. This constant has pointer to character type! The only case when you would like a string literal to have an array type is when it is an argument to `sizeof`. In that case you should use `SizeOfStr`.

| **CWStr of int64 list**

Wide character string constant. Note that the local interpretation of such a literal depends on `Cil.wcharType[5]` and `Cil.wcharKind[5]`. Such a constant has type pointer to `Cil.wcharType[5]`. The escape characters in the string have not been "interpreted" in the sense that `L"A\xabcd"` remains `"A\xabcd"` rather than being represented as the wide character list with two elements: 65 and 43981. That "interpretation" depends on the underlying wide character type.

| **CChr of char**

Character constant. This has type `int`, so use `charConstToInt` to read the value in case sign-extension is needed.

| **CReal of float * fkind * string option**

Floating point constant. Give the `fkind` (see ISO 6.4.4.2) and also the textual representation, if available.

| **CEnum of exp * string * enuminfo**

An enumeration constant with the given value, name, from the given `enuminfo`. This is used only if `Cil.lowerConstants[5]` is true (default). Use `Cil.constFoldVisitor[5]` to replace these with integer constants.

Literal constants

`type unop =`

| **Neg**

Unary minus

| **BNot**

Bitwise complement (~)

| **LNot**

Logical Not (!)

Unary operators

`type binop =`

| **PlusA**

arithmetic +

| **PlusPI**

pointer + integer

IndexPI	pointer + integer but only when it arises from an expression e[i] when e is a pointer and not an array. This is semantically the same as PlusPI but CCured uses this as a hint that the integer is probably positive.
MinusA	arithmetic -
MinusPI	pointer - integer
MinusPP	pointer - pointer
Mult	
Div	/
Mod	%
Shiftlt	shift left
Shiftrt	shift right
Lt	< (arithmetic comparison)
Gt	> (arithmetic comparison)
Le	\leq (arithmetic comparison)
Ge	> (arithmetic comparison)
Eq	== (arithmetic comparison)
Ne	!= (arithmetic comparison)
BAnd	bitwise and
BXor	exclusive-or
BOr	inclusive-or

| LAnd

logical and. Unlike other expressions this one does not always evaluate both operands. If you want to use these, you must set `Cil.useLogicalOperators[5]`.

| LOr

logical or. Unlike other expressions this one does not always evaluate both operands. If you want to use these, you must set `Cil.useLogicalOperators[5]`.

Binary operations

Lvalues. Lvalues are the sublanguage of expressions that can appear at the left of an assignment or as operand to the address-of operator. In C the syntax for lvalues is not always a good indication of the meaning of the lvalue. For example the C value

`a[0][1][2]`

might involve 1, 2 or 3 memory reads when used in an expression context, depending on the declared type of the variable `a`. If `a` has type `int [4][4][4]` then we have one memory read from somewhere inside the area that stores the array `a`. On the other hand if `a` has type `int ***` then the expression really means `* (* (* (a + 0) + 1) + 2)`, in which case it is clear that it involves three separate memory operations.

An lvalue denotes the contents of a range of memory addresses. This range is denoted as a host object along with an offset within the object. The host object can be of two kinds: a local or global variable, or an object whose address is in a pointer expression. We distinguish the two cases so that we can tell quickly whether we are accessing some component of a variable directly or we are accessing a memory location through a pointer. To make it easy to tell what an lvalue means CIL represents lvalues as a host object and an offset (see `Cil.lval[5]`). The host object (represented as `Cil.lhost[5]`) can be a local or global variable or can be the object pointed-to by a pointer expression. The offset (represented as `Cil.offset[5]`) is a sequence of field or array index designators.

Both the typing rules and the meaning of an lvalue is very precisely specified in CIL.

The following are a few useful function for operating on lvalues:

- `Cil.mkMem[5]` - makes an lvalue of `Mem` kind. Use this to ensure that certain equivalent forms of lvalues are canonized. For example, `*&x = x`.
- `Cil.typeOfLval[5]` - the type of an lvalue
- `Cil.typeOffset[5]` - the type of an offset, given the type of the host.
- `Cil.addOffset[5]` and `Cil.addOffsetLval[5]` - extend sequences of offsets.
- `Cil.removeOffset[5]` and `Cil.removeOffsetLval[5]` - shrink sequences of offsets.

The following equivalences hold

```
Mem(AddrOf(Mem a, aoff)), off  = Mem a, aoff + off
Mem(AddrOf(Var v, aoff)), off  = Var v, aoff + off
AddrOf (Mem a, NoOffset)      = a
```

```
type lval = lhost * offset
```

An lvalue


```

type lhost =
  | Var of varinfo
      The host is a variable.

  | Mem of exp
      The host is an object of type T when the expression has pointer TPtr(T).
      The host part of an Cil.lval[5].

type offset =
  | NoOffset
      No offset. Can be applied to any lvalue and does not change either the starting
      address or the type. This is used when the lval consists of just a host or as a
      terminator in a list of other kinds of offsets.

  | Field of fieldinfo * offset
      A field offset. Can be applied only to an lvalue that denotes a structure or a union
      that contains the mentioned field. This advances the offset to the beginning of the
      mentioned field and changes the type to the type of the mentioned field.

  | Index of exp * offset
      An array index offset. Can be applied only to an lvalue that denotes an array. This
      advances the starting address of the lval to the beginning of the mentioned array
      element and changes the denoted type to be the type of the array element

      The offset part of an Cil.lval[5]. Each offset can be applied to certain kinds of lvalues and
      its effect is that it advances the starting address of the lvalue and changes the denoted type,
      essentially focusing to some smaller lvalue that is contained in the original one.

```

Initializers. A special kind of expressions are those that can appear as initializers for global variables (initialization of local variables is turned into assignments). The initializers are represented as type `Cil.init[5]`. You can create initializers with `Cil.makeZeroInit[5]` and you can conveniently scan compound initializers them with `Cil.foldLeftCompound[5]` or with `Cil.foldLeftInitializer`.

```

type init =
  | SingleInit of exp
      A single initializer

  | CompoundInit of typ * (offset * init) list
      Used only for initializers of structures, unions and arrays. The offsets are all of the
      form Field(f, NoOffset) or Index(i, NoOffset) and specify the field or the index
      being initialized. For structures all fields must have an initializer (except the unnamed
      bitfields), in the proper order. This is necessary since the offsets are not printed. For
      unions there must be exactly one initializer. If the initializer is not for the first field
      then a field designator is printed, so you better be on GCC since MSVC does not
      understand this. For arrays, however, we allow you to give only a prefix of the
      initializers. You can scan an initializer list with Cil.foldLeftCompound[5] or with
      Cil.foldLeftInitializer.

```

Initializers for global variables.

```

type initinfo = {
  mutable init : init option ;
}

```

We want to be able to update an initializer in a global variable, so we define it as a mutable field

Function definitions. A function definition is always introduced with a `GFun` constructor at the top level. All the information about the function is stored into a `Cil.fundec[5]`. Some of the information (e.g. its name, type, storage, attributes) is stored as a `Cil.varinfo[5]` that is a field of the `fundec`. To refer to the function from the expression language you must use the `varinfo`.

The function definition contains, in addition to the body, a list of all the local variables and separately a list of the formals. Both kind of variables can be referred to in the body of the function. The formals must also be shared with the formals that appear in the function type. For that reason, to manipulate formals you should use the provided functions `Cil.makeFormalVar[5]` and `Cil.setFormals[5]` and `Cil.makeFormalVar[5]`.

```

type fundec = {
  mutable svar : varinfo ;
    Holds the name and type as a variable, so we can refer to it easily from the program.
    All references to this function either in a function call or in a prototype must point to
    the same varinfo.

  mutable sformals : varinfo list ;
    Formals. These must be in the same order and with the same information as the
    formal information in the type of the function. Use Cil.setFormals[5] or
    Cil.setFunctionType[5] or Cil.makeFormalVar[5] to set these formals and ensure
    that they are reflected in the function type. Do not make copies of these because the
    body refers to them.

  mutable slocals : varinfo list ;
    Locals. Does NOT include the sformals. Do not make copies of these because the
    body refers to them.

  mutable smaxid : int ;
    Max local id. Starts at 0. Used for creating the names of new temporary variables.
    Updated by Cil.makeLocalVar[5] and Cil.makeTempVar[5]. You can also use
    Cil.setMaxId[5] to set it after you have added the formals and locals.

  mutable sbody : block ;
    The function body.

  mutable smaxstmtid : int option ;
    max id of a (reachable) statement in this function, if we have computed it. range = 0
    ... (smaxstmtid-1). This is computed by Cil.computeCFGInfo[5].

  mutable sallstmts : stmt list ;
    After you call Cil.computeCFGInfo[5] this field is set to contain all statements in the
    function
}

```

Function definitions.

```

type block = {
  mutable battrs : attributes ;
      Attributes for the block

  mutable bstmts : stmt list ;
      The statements comprising the block
}

```

A block is a sequence of statements with the control falling through from one element to the next

Statements. CIL statements are the structural elements that make the CFG. They are represented using the type `Cil.stmt[5]`. Every statement has a (possibly empty) list of labels. The `Cil.stmtkind[5]` field of a statement indicates what kind of statement it is.

Use `Cil.mkStmt[5]` to make a statement and the fill-in the fields.

CIL also comes with support for control-flow graphs. The `sid` field in `stmt` can be used to give unique numbers to statements, and the `succs` and `preds` fields can be used to maintain a list of successors and predecessors for every statement. The CFG information is not computed by default. Instead you must explicitly use the functions `Cil.prepareCFG[5]` and `Cil.computeCFGInfo[5]` to do it.

```

type stmt = {
  mutable labels : label list ;
      Whether the statement starts with some labels, case statements or default statements.

  mutable skind : stmtkind ;
      The kind of statement

  mutable sid : int ;
      A number ( $\geq 0$ ) that is unique in a function. Filled in only after the CFG is computed.

  mutable succs : stmt list ;
      The successor statements. They can always be computed from the skind and the
      context in which this statement appears. Filled in only after the CFG is computed.

  mutable preds : stmt list ;
      The inverse of the succs function.
}

Statements.

```

```

type label =
| Label of string * location * bool
      A real label. If the bool is "true", the label is from the input source program. If the
      bool is "false", the label was created by CIL or some other transformation

| Case of exp * location
      A case statement. This expression is lowered into a constant if
      Cil.lowerConstants[5] is set to true.

| Default of location

```

A default statement

Labels

type stmtkind =

| Instr of instr list

A group of instructions that do not contain control flow. Control implicitly falls through.

| Return of exp option * location

The return statement. This is a leaf in the CFG.

| Goto of stmt Pervasives.ref * location

A goto statement. Appears from actual goto's in the code or from goto's that have been inserted during elaboration. The reference points to the statement that is the target of the Goto. This means that you have to update the reference whenever you replace the target statement. The target statement **MUST** have at least a label.

| Break of location

A break to the end of the nearest enclosing Loop or Switch

| Continue of location

A continue to the start of the nearest enclosing Loop

| If of exp * block * block * location

A conditional. Two successors, the "then" and the "else" branches. Both branches fall-through to the successor of the If statement.

| Switch of exp * block * stmt list * location

A switch statement. The statements that implement the cases can be reached through the provided list. For each such target you can find among its labels what cases it implements. The statements that implement the cases are somewhere within the provided block.

| Loop of block * location * stmt option * stmt option

A **while**(1) loop. The termination test is implemented in the body of a loop using a **Break** statement. If prepareCFG has been called, the first stmt option will point to the stmt containing the continue label for this loop and the second will point to the stmt containing the break label for this loop.

| Block of block

Just a block of statements. Use it as a way to keep some block attributes local

| TryFinally of block * block * location

| TryExcept of block * (instr list * exp) * block * location

The various kinds of control-flow statements

Instructions. An instruction `Cil.instr[5]` is a statement that has no local (intraprocedural) control flow. It can be either an assignment, function call, or an inline assembly instruction.

type instr =

| Set of lval * exp * location

An assignment. The type of the expression is guaranteed to be the same with that of the lvalue

| Call of lval option * exp * exp list * location

A function call with the (optional) result placed in an lval. It is possible that the returned type of the function is not identical to that of the lvalue. In that case a cast is printed. The type of the actual arguments are identical to those of the declared formals. The number of arguments is the same as that of the declared formals, except for vararg functions. This construct is also used to encode a call to `"__builtin_va_arg"`. In this case the second argument (which should be a type T) is encoded `SizeOf(T)`

| Asm of attributes * string list * (string option * string * lval) list
* (string option * string * exp) list * string list * location

There are for storing inline assembly. They follow the GCC specification:

```
asm [volatile] ("...template..." "..template.."
                : "c1" (o1), "c2" (o2), ..., "cN" (oN)
                : "d1" (i1), "d2" (i2), ..., "dM" (iM)
                : "r1", "r2", ..., "nL" );
```

where the parts are

- **volatile** (optional): when present, the assembler instruction cannot be removed, moved, or otherwise optimized
- **template**: a sequence of strings, with %0, %1, %2, etc. in the string to refer to the input and output expressions. I think they're numbered consecutively, but the docs don't specify. Each string is printed on a separate line. This is the only part that is present for MSVC inline assembly.
- **"ci" (oi)**: pairs of constraint-string and output-lval; the constraint specifies that the register used must have some property, like being a floating-point register; the constraint string for outputs also has "=" to indicate it is written, or "+" to indicate it is both read and written; 'oi' is the name of a C lvalue (probably a variable name) to be used as the output destination
- **"dj" (ij)**: pairs of constraint and input expression; the constraint is similar to the "ci"s. the 'ij' is an arbitrary C expression to be loaded into the corresponding register
- **"rk"**: registers to be regarded as "clobbered" by the instruction; "memory" may be specified for arbitrary memory effects

an example (from gcc manual):

```
asm volatile ("movc3 %0,%1,%2"
              : /* no outputs */
              : "g" (from), "g" (to), "g" (count)
              : "r0", "r1", "r2", "r3", "r4", "r5");
```

Starting with gcc 3.1, the operands may have names:

```

asm volatile ("movc3 %[in0],%1,%2"
              : /* no outputs */
              : [in0] "g" (from), "g" (to), "g" (count)
              : "r0", "r1", "r2", "r3", "r4", "r5");

```

Instructions.

```

type location = {
  line : int ;
      The line number. -1 means "do not know"

  file : string ;
      The name of the source file

  byte : int ;
      The byte position in the source file
}

```

Describes a location in a source file.

```

type typsig =
| TArray of typsig * int64 option * attribute list
| TSPtr of typsig * attribute list
| TSComp of bool * string * attribute list
| TSFun of typsig * typsig list * bool * attribute list
| TSEnum of string * attribute list
| TSBase of typ

```

Type signatures. Two types are identical iff they have identical signatures. These contain the same information as types but canonicalized. For example, two function types that are identical except for the name of the formal arguments are given the same signature. Also, `TNamed` constructors are unrolled.

Lowering Options

```

val lowerConstants : bool Pervasives.ref
    Do lower constants (default true)

```

```

val insertImplicitCasts : bool Pervasives.ref
    Do insert implicit casts (default true)

```

```

type featureDescr = {
  fd_enabled : bool Pervasives.ref ;
      The enable flag. Set to default value

  fd_name : string ;
      This is used to construct an option "-doxxx" and "-dontxxx" that enable and disable
      the feature

  fd_description : string ;
      A longer name that can be used to document the new options
}

```

```

fd_extraopt : (string * Arg.spec * string) list ;
    Additional command line options

fd_doit : file -> unit ;
    This performs the transformation

fd_post_check : bool ;
    Whether to perform a CIL consistency checking after this stage, if checking is enabled
    (-check is passed to cilly). Set this to true if your feature makes any changes for the
    program.
}

```

To be able to add/remove features easily, each feature should be package as an interface with the following interface. These features should be

```

val compareLoc : location -> location -> int
    Comparison function for locations. * Compares first by filename, then line, then byte

```

Values for manipulating globals

```

val emptyFunction : string -> fundec
    Make an empty function

val setFormals : fundec -> varinfo list -> unit
    Update the formals of a fundec and make sure that the function type has the same
    information. Will copy the name as well into the type.

val setFunctionType : fundec -> typ -> unit
    Set the types of arguments and results as given by the function type passed as the second
    argument. Will not copy the names from the function type to the formals

val setFunctionTypeMakeFormals : fundec -> typ -> unit
    Set the type of the function and make formal arguments for them

val setMaxId : fundec -> unit
    Update the smaxid after you have populated with locals and formals (unless you constructed
    those using Cil.makeLocalVar[5] or Cil.makeTempVar[5].

val dummyFunDec : fundec
    A dummy function declaration handy when you need one as a placeholder. It contains inside
    a dummy varinfo.

val dummyFile : file
    A dummy file

val saveBinaryFile : file -> string -> unit
    Write a Cil.file[5] in binary form to the filesystem. The file can be read back in later
    using Cil.loadBinaryFile[5], possibly saving parsing time. The second argument is the
    name of the file that should be created.

```

```

val saveBinaryFileChannel : file -> Pervasives.out_channel -> unit
    Write a Cil.file[5] in binary form to the filesystem. The file can be read back in later
    using Cil.loadBinaryFile[5], possibly saving parsing time. Does not close the channel.

val loadBinaryFile : string -> file
    Read a Cil.file[5] in binary form from the filesystem. The first argument is the name of a
    file previously created by Cil.saveBinaryFile[5].

val getGlobInit : ?main_name:string -> file -> fundec
    Get the global initializer and create one if it does not already exist. When it creates a global
    initializer it attempts to place a call to it in the main function named by the optional
    argument (default "main")

val iterGlobals : file -> (global -> unit) -> unit
    Iterate over all globals, including the global initializer

val foldGlobals : file -> ('a -> global -> 'a) -> 'a -> 'a
    Fold over all globals, including the global initializer

val mapGlobals : file -> (global -> global) -> unit
    Map over all globals, including the global initializer and change things in place

val findOrCreateFunc : file -> string -> typ -> varinfo
    Find a function or function prototype with the given name in the file. If it does not exist,
    create a prototype with the given type, and return the new varinfo. This is useful when you
    need to call a libc function whose prototype may or may not already exist in the file.

    Because the new prototype is added to the start of the file, you shouldn't refer to any struct
    or union types in the function type.

val new_sid : unit -> int
val prepareCFG : fundec -> unit
    Prepare a function for CFG information computation by Cil.computeCFGInfo[5]. This
    function converts all Break, Switch, Default and Continue Cil.stmtkind[5]s and
    Cil.label[5]s into Ifs and Gotos, giving the function body a very CFG-like character. This
    function modifies its argument in place.

val computeCFGInfo : fundec -> bool -> unit
    Compute the CFG information for all statements in a fundec and return a list of the
    statements. The input fundec cannot have Break, Switch, Default, or Continue
    Cil.stmtkind[5]s or Cil.label[5]s. Use Cil.prepareCFG[5] to transform them away. The
    second argument should be true if you wish a global statement number, false if you wish a
    local (per-function) statement numbering. The list of statements is set in the sallstmts field
    of a fundec.

    NOTE: unless you want the simpler control-flow graph provided by prepareCFG, or you
    need the function's smaxstmtid and sallstmt fields filled in, we recommend you use
    Cfg.computeFileCFG instead of this function to compute control-flow information.
    Cfg.computeFileCFG is newer and will handle switch, break, and continue correctly.

```



```
val copyFunction : fundec -> string -> fundec
```

Create a deep copy of a function. There should be no sharing between the copy and the original function

```
val pushGlobal :  
  global ->  
  types:global list Pervasives.ref ->  
  variables:global list Pervasives.ref -> unit
```

CIL keeps the types at the beginning of the file and the variables at the end of the file. This function will take a global and add it to the corresponding stack. Its operation is actually more complicated because if the global declares a type that contains references to variables (e.g. in sizeof in an array length) then it will also add declarations for the variables to the types stack

```
val invalidStmt : stmt
```

An empty statement. Used in pretty printing

```
val gccBuiltins : (string, typ * typ list * bool) Hashtbl.t
```

A list of the GCC built-in functions. Maps the name to the result and argument types, and whether it is vararg

```
val msvcBuiltins : (string, typ * typ list * bool) Hashtbl.t
```

A list of the MSVC built-in functions. Maps the name to the result and argument types, and whether it is vararg

```
val builtinLoc : location
```

This is used as the location of the prototypes of builtin functions.

Values for manipulating initializers

```
val makeZeroInit : typ -> init
```

Make a initializer for zero-ing a data type

```
val foldLeftCompound :  
  implicit:bool ->  
  doinit:(offset -> init -> typ -> 'a -> 'a) ->  
  ct:typ -> init1:(offset * init) list -> acc:'a -> 'a
```

Fold over the list of initializers in a Compound (not also the nested ones). `doinit` is called on every present initializer, even if it is of compound type. The parameters of `doinit` are: the offset in the compound (this is `Field(f,NoOffset)` or `Index(i,NoOffset)`), the initializer value, expected type of the initializer value, accumulator. In the case of arrays there might be missing zero-initializers at the end of the list. These are scanned only if `implicit` is true. This is much like `List.fold_left` except we also pass the type of the initializer.

This is a good way to use it to scan even nested initializers :

```
let rec myInit (lv: lval) (i: init) (acc: 'a) : 'a =  
  match i with  
  | SingleInit e -> ... do something with lv and e and acc ...
```

```

    | CompoundInit (ct, initl) ->
        foldLeftCompound ~implicit:false
            ~doinit:(fun off' i' t' acc ->
                myInit (addOffsetLval lv off') i' acc)
            ~ct:ct
            ~initl:initl
            ~acc:acc

```

Values for manipulating types

```

val voidType : typ
    void

val isVoidType : typ -> bool
    is the given type "void"?

val isVoidPtrType : typ -> bool
    is the given type "void *"?

val intType : typ
    int

val uintType : typ
    unsigned int

val longType : typ
    long

val ulongType : typ
    unsigned long

val charType : typ
    char

val charPtrType : typ
    char *

val wcharKind : ikind Pervasives.ref
    wchar_t (depends on architecture) and is set when you call Cil.initCIL[5].

val wcharType : typ Pervasives.ref

val charConstPtrType : typ
    char const *

val voidPtrType : typ
    void *

val intPtrType : typ

```

```

    int *

val uintPtrType : typ
    unsigned int *

val doubleType : typ
    double

val upointType : typ Pervasives.ref
    An unsigned integer type that fits pointers. Depends on Cil.msvcMode[5] and is set when
    you call Cil.initCIL[5].

val typeOfSizeOf : typ Pervasives.ref
    An unsigned integer type that is the type of sizeof. Depends on Cil.msvcMode[5] and is set
    when you call Cil.initCIL[5].

val isSigned : ikind -> bool
    Returns true if and only if the given integer type is signed.

val mkCompInfo :
    bool ->
    string ->
    (compinfo ->
        (string * typ * int option * attributes * location) list) ->
    attributes -> compinfo
    Creates a a (potentially recursive) composite type. The arguments are: (1) a boolean
    indicating whether it is a struct or a union, (2) the name (always non-empty), (3) a function
    that when given a representation of the structure type constructs the type of the fields
    recursive type (the first argument is only useful when some fields need to refer to the type of
    the structure itself), and (4) a list of attributes to be associated with the composite type.
    The resulting compinfo has the field "cdefined" only if the list of fields is non-empty.

val copyCompInfo : compinfo -> string -> compinfo
    Makes a shallow copy of a Cil.compinfo[5] changing the name and the key.

val missingFieldName : string
    This is a constant used as the name of an unnamed bitfield. These fields do not participate
    in initialization and their name is not printed.

val compFullName : compinfo -> string
    Get the full name of a comp

val isCompleteType : typ -> bool
    Returns true if this is a complete type. This means that sizeof(t) makes sense. Incomplete
    types are not yet defined structures and empty arrays.

val unrollType : typ -> typ

```

Unroll a type until it exposes a non `TNamed`. Will collect all attributes appearing in `TNamed`!!!

`val unrollTypeDeep : typ -> typ`

Unroll all the `TNamed` in a type (even under type constructors such as `TPtr`, `TFun` or `TArray`. Does not unroll the types of fields in `TComp` types. Will collect all attributes

`val separateStorageModifiers :`

`attribute list -> attribute list * attribute list`

Separate out the storage-modifier name attributes

`val isIntegralType : typ -> bool`

True if the argument is an integral type (i.e. integer or enum)

`val isArithmeticType : typ -> bool`

True if the argument is an arithmetic type (i.e. integer, enum or floating point

`val isPointerType : typ -> bool`

True if the argument is a pointer type

`val isFunctionType : typ -> bool`

True if the argument is a function type

`val argsToList :`

`(string * typ * attributes) list option ->`

`(string * typ * attributes) list`

Obtain the argument list ([] if None)

`val isArrayType : typ -> bool`

True if the argument is an array type

`exception LenOfArray`

Raised when `Cil.lenOfArray[5]` fails either because the length is `None` or because it is a non-constant expression

`val lenOfArray : exp option -> int`

Call to compute the array length as present in the array type, to an integer. Raises `Cil.LenOfArray[5]` if not able to compute the length, such as when there is no length or the length is not a constant.

`val getCompField : compinfo -> string -> fieldinfo`

Return a named fieldinfo in compinfo, or raise `Not_found`

`type existsAction =`

`| ExistsTrue`

We have found it

`| ExistsFalse`

Stop processing this branch

| **ExistsMaybe**

This node is not what we are looking for but maybe its successors are

A datatype to be used in conjunction with **existsType**

```
val existsType : (typ -> existsAction) -> typ -> bool
```

Scans a type by applying the function on all elements. When the function returns **ExistsTrue**, the scan stops with true. When the function returns **ExistsFalse** then the current branch is not scanned anymore. Care is taken to apply the function only once on each composite type, thus avoiding circularity. When the function returns **ExistsMaybe** then the types that construct the current type are scanned (e.g. the base type for **TPtr** and **TArray**, the type of fields for a **TComp**, etc).

```
val splitFunctionType :
```

```
  typ ->
```

```
  typ * (string * typ * attributes) list option * bool *  
  attributes
```

Given a function type split it into return type, arguments, **is_vararg** and attributes. An error is raised if the type is not a function type

Same as **Cil.splitFunctionType[5]** but takes a **varinfo**. Prints a nicer error message if the **varinfo** is not for a function

```
val splitFunctionTypeVI :
```

```
  varinfo ->
```

```
  typ * (string * typ * attributes) list option * bool *  
  attributes
```

Type signatures

Type signatures. Two types are identical iff they have identical signatures. These contain the same information as types but canonicalized. For example, two function types that are identical except for the name of the formal arguments are given the same signature. Also, **TNamed** constructors are unrolled.

```
val d_typsig : unit -> typsig -> Pretty.doc
```

Print a type signature

```
val typeSig : typ -> typsig
```

Compute a type signature

```
val typeSigWithAttrs :
```

```
  ?ignoreSign:bool ->
```

```
  (attributes -> attributes) -> typ -> typsig
```

Like **Cil.typeSig[5]** but customize the incorporation of attributes. Use **~ignoreSign:true** to convert all signed integer types to unsigned, so that signed and unsigned will compare the same.

```
val setTypeSigAttrs : attributes -> typsig -> typsig
```

Replace the attributes of a signature (only at top level)

`val typeSigAttrs : typsig -> attributes`
Get the top-level attributes of a signature

LVALUES

`val makeVarinfo : bool -> string -> typ -> varinfo`
Make a varinfo. Use this (rarely) to make a raw varinfo. Use other functions to make locals (`Cil.makeLocalVar[5]` or `Cil.makeFormalVar[5]` or `Cil.makeTempVar[5]`) and globals (`Cil.makeGlobalVar[5]`). Note that this function will assign a new identifier. The first argument specifies whether the varinfo is for a global.

`val makeFormalVar : fundec -> ?where:string -> string -> typ -> varinfo`
Make a formal variable for a function. Insert it in both the sformals and the type of the function. You can optionally specify where to insert this one. If `where = "^"` then it is inserted first. If `where = "$"` then it is inserted last. Otherwise `where` must be the name of a formal after which to insert this. By default it is inserted at the end.

`val makeLocalVar : fundec -> ?insert:bool -> string -> typ -> varinfo`
Make a local variable and add it to a function's slocals (only if `insert = true`, which is the default). Make sure you know what you are doing if you set `insert=false`.

`val makeTempVar : fundec -> ?name:string -> typ -> varinfo`
Make a temporary variable and add it to a function's slocals. The name of the temporary variable will be generated based on the given name hint so that to avoid conflicts with other locals.

`val makeGlobalVar : string -> typ -> varinfo`
Make a global variable. Your responsibility to make sure that the name is unique

`val copyVarinfo : varinfo -> string -> varinfo`
Make a shallow copy of a `varinfo` and assign a new identifier

`val newVID : unit -> int`
Generate a new variable ID. This will be different than any variable ID that is generated by `Cil.makeLocalVar[5]` and friends

`val addOffsetLval : offset -> lval -> lval`
Add an offset at the end of an lvalue. Make sure the type of the lvalue and the offset are compatible.

`val addOffset : offset -> offset -> offset`
`addOffset o1 o2` adds `o1` to the end of `o2`.

`val removeOffsetLval : lval -> lval * offset`
Remove ONE offset from the end of an lvalue. Returns the lvalue with the trimmed offset and the final offset. If the final offset is `NoOffset` then the original `lval` did not have an offset.

val removeOffset : offset -> offset * offset
Remove ONE offset from the end of an offset sequence. Returns the trimmed offset and the final offset. If the final offset is **NoOffset** then the original **lval** did not have an offset.

val typeOfLval : lval -> typ
Compute the type of an lvalue

val typeOffset : typ -> offset -> typ
Compute the type of an offset from a base type

Values for manipulating expressions

val zero : exp
0

val one : exp
1

val mone : exp
-1

val kinteger64 : ikind -> int64 -> exp
Construct an integer of a given kind, using OCaml's int64 type. If needed it will truncate the integer to be within the representable range for the given kind.

val kinteger : ikind -> int -> exp
Construct an integer of a given kind. Converts the integer to int64 and then uses **kinteger64**. This might truncate the value if you use a kind that cannot represent the given integer. This can only happen for one of the Char or Short kinds

val integer : int -> exp
Construct an integer of kind **IInt**. You can use this always since the OCaml integers are 31 bits and are guaranteed to fit in an **IInt**

val isInteger : exp -> int64 option
True if the given expression is a (possibly cast'ed) character or an integer constant

val i64_to_int : int64 -> int
Convert a 64-bit int to an OCaml int, or raise an exception if that can't be done.

val isConstant : exp -> bool
True if the expression is a compile-time constant

val isZero : exp -> bool
True if the given expression is a (possibly cast'ed) integer or character constant with value zero

val charConstToInt : char -> constant

Given the character `c` in a `(CChr c)`, sign-extend it to 32 bits. (This is the official way of interpreting character constants, according to ISO C 6.4.4.4.10, which says that character constants are chars cast to ints) Returns `CInt64(sign-extended c, IInt, None)`

val constFold : bool -> exp -> exp

Do constant folding on an expression. If the first argument is true then will also compute compiler-dependent expressions such as `sizeof`. See also `Cil.constFoldVisitor[5]`, which will run `constFold` on all expressions in a given AST node.

val constFoldBinOp : bool -> binop -> exp -> exp -> typ -> exp

Do constant folding on a binary operation. The bulk of the work done by `constFold` is done here. If the first argument is true then will also compute compiler-dependent expressions such as `sizeof`

val increm : exp -> int -> exp

Increment an expression. Can be arithmetic or pointer type

val var : varinfo -> lval

Makes an lvalue out of a given variable

val mkAddrOf : lval -> exp

Make an `AddrOf`. Given an lvalue of type `T` will give back an expression of type `ptr(T)`. It optimizes somewhat expressions like `"& v"` and `"& v0"`

val mkAddrOrStartOf : lval -> exp

Like `mkAddrOf` except if the type of lval is an array then it uses `StartOf`. This is the right operation for getting a pointer to the start of the storage denoted by lval.

val mkMem : addr:exp -> off:offset -> lval

Make a `Mem`, while optimizing `AddrOf`. The type of the `addr` must be `TPtr(t)` and the type of the resulting lval is `t`. Note that in CIL the implicit conversion between an array and the pointer to the first element does not apply. You must do the conversion yourself using `StartOf`

val mkString : string -> exp

Make an expression that is a string constant (of pointer type)

val mkCastT : e:exp -> oldt:typ -> newt:typ -> exp

Construct a cast when having the old type of the expression. If the new type is the same as the old type, then no cast is added.

val mkCast : e:exp -> newt:typ -> exp

Like `Cil.mkCastT[5]` but uses `typeOf` to get `oldt`

val stripCasts : exp -> exp

Removes casts from this expression, but ignores casts within other expression constructs. So we delete the `(A)` and `(B)` casts from `"(A)(B)(x + (C)y)"`, but leave the `(C)` cast.


```

val typeOf : exp -> typ
    Compute the type of an expression

val parseInt : string -> exp
    Convert a string representing a C integer literal to an expression. Handles the prefixes 0x
    and 0 and the suffixes L, U, UL, LL, ULL

Values for manipulating statements

val mkStmt : stmtkind -> stmt
    Construct a statement, given its kind. Initialize the sid field to -1, and labels, succs and
    preds to the empty list

val mkBlock : stmt list -> block
    Construct a block with no attributes, given a list of statements

val mkStmtOneInstr : instr -> stmt
    Construct a statement consisting of just one instruction

val compactStmts : stmt list -> stmt list
    Try to compress statements so as to get maximal basic blocks. use this instead of List.@
    because you get fewer basic blocks

val mkEmptyStmt : unit -> stmt
    Returns an empty statement (of kind Instr)

val dummyInstr : instr
    A instr to serve as a placeholder

val dummyStmt : stmt
    A statement consisting of just dummyInstr

val mkWhile : guard:exp -> body:stmt list -> stmt list
    Make a while loop. Can contain Break or Continue

val mkForIncr :
    iter:varinfo ->
    first:exp ->
    stopat:exp -> incr:exp -> body:stmt list -> stmt list
    Make a for loop for(i=start; i<past; i += incr) { ... }. The body can contain Break but not
    Continue. Can be used with i a pointer or an integer. Start and done must have the same
    type but incr must be an integer

val mkFor :
    start:stmt list ->
    guard:exp -> next:stmt list -> body:stmt list -> stmt list
    Make a for loop for(start; guard; next) { ... }. The body can contain Break but not
    Continue !!!

```

Values for manipulating attributes

```
type attributeClass =  
  | AttrName of bool  
      Attribute of a name. If argument is true and we are on MSVC then the attribute is  
      printed using __declspec as part of the storage specifier  
  | AttrFunType of bool  
      Attribute of a function type. If argument is true and we are on MSVC then the  
      attribute is printed just before the function name  
  | AttrType  
      Attribute of a type  
      Various classes of attributes  
  
val attributeHash : (string, attributeClass) Hashtbl.t  
    This table contains the mapping of predefined attributes to classes. Extend this table with  
    more attributes as you need. This table is used to determine how to associate attributes  
    with names or types  
  
val partitionAttributes :  
  default:attributeClass ->  
  attributes ->  
  attribute list * attribute list * attribute list  
    Partition the attributes into classes: name attributes, function type, and type attributes  
  
val addAttribute : attribute -> attributes -> attributes  
    Add an attribute. Maintains the attributes in sorted order of the second argument  
  
val addAttributes : attribute list -> attributes -> attributes  
    Add a list of attributes. Maintains the attributes in sorted order. The second argument  
    must be sorted, but not necessarily the first  
  
val dropAttribute : string -> attributes -> attributes  
    Remove all attributes with the given name. Maintains the attributes in sorted order.  
  
val dropAttributes : string list -> attributes -> attributes  
    Remove all attributes with names appearing in the string list. Maintains the attributes in  
    sorted order  
  
val filterAttributes : string -> attributes -> attributes  
    Retains attributes with the given name  
  
val hasAttribute : string -> attributes -> bool  
    True if the named attribute appears in the attribute list. The list of attributes must be  
    sorted.  
  
val typeAttrs : typ -> attribute list
```

Returns all the attributes contained in a type. This requires a traversal of the type structure, in case of composite, enumeration and named types

```
val setTypeAttrs : typ -> attributes -> typ
```

```
val typeAddAttributes : attribute list -> typ -> typ
```

Add some attributes to a type

```
val typeRemoveAttributes : string list -> typ -> typ
```

Remove all attributes with the given names from a type. Note that this does not remove attributes from typedef and tag definitions, just from their uses

The visitor

```
type 'a visitAction =
```

```
| SkipChildren
```

Do not visit the children. Return the node as it is.

```
| DoChildren
```

Continue with the children of this node. Rebuild the node on return if any of the children changes (use == test)

```
| ChangeTo of 'a
```

Replace the expression with the given one

```
| ChangeDoChildrenPost of 'a * ('a -> 'a)
```

First consider that the entire exp is replaced by the first parameter. Then continue with the children. On return rebuild the node if any of the children has changed and then apply the function on the node

Different visiting actions. 'a will be instantiated with `exp`, `instr`, etc.

```
class type cilVisitor =
```

```
object
```

```
method vvdec : Cil.varinfo -> Cil.varinfo Cil.visitAction
```

Invoked for each variable declaration. The subtrees to be traversed are those corresponding to the type and attributes of the variable. Note that variable declarations are all the `GVar`, `GVarDecl`, `GFun`, all the `varinfo` in formals of function types, and the formals and locals for function definitions. This means that the list of formals in a function definition will be traversed twice, once as part of the function type and second as part of the formals in a function definition.

```
method vvrb1 : Cil.varinfo -> Cil.varinfo Cil.visitAction
```

Invoked on each variable use. Here only the `SkipChildren` and `ChangeTo` actions make sense since there are no subtrees. Note that the type and attributes of the variable are not traversed for a variable use

```
method vexpr : Cil.exp -> Cil.exp Cil.visitAction
```

Invoked on each expression occurrence. The subtrees are the subexpressions, the types (for a `Cast` or `SizeOf` expression) or the variable use.

`method vlval : Cil.lval -> Cil.lval Cil.visitAction`

Invoked on each lvalue occurrence

`method voffs : Cil.offset -> Cil.offset Cil.visitAction`

Invoked on each offset occurrence that is **not** as part of an initializer list specification, i.e. in an lval or recursively inside an offset.

`method vinitoffs : Cil.offset -> Cil.offset Cil.visitAction`

Invoked on each offset appearing in the list of a `CompoundInit` initializer.

`method vinst : Cil.instr -> Cil.instr list Cil.visitAction`

Invoked on each instruction occurrence. The `ChangeTo` action can replace this instruction with a list of instructions

`method vstmt : Cil.stmt -> Cil.stmt Cil.visitAction`

Control-flow statement. The default `DoChildren` action does not create a new statement when the components change. Instead it updates the contents of the original statement. This is done to preserve the sharing with `Goto` and `Case` statements that point to the original statement. If you use the `ChangeTo` action then you should take care of preserving that sharing yourself.

`method vblock : Cil.block -> Cil.block Cil.visitAction`

Block.

`method vfunc : Cil.fundec -> Cil.fundec Cil.visitAction`

Function definition. Replaced in place.

`method vglob : Cil.global -> Cil.global list Cil.visitAction`

Global (vars, types, etc.)

`method vinit :`

`Cil.varinfo -> Cil.offset -> Cil.init -> Cil.init Cil.visitAction`

Initializers for globals, pass the global where this occurs, and the offset

`method vtype : Cil.typ -> Cil.typ Cil.visitAction`

Use of some type. Note that for structure/union and enumeration types the definition of the composite type is not visited. Use `vglob` to visit it.

`method vattr : Cil.attribute -> Cil.attribute list Cil.visitAction`

Attribute. Each attribute can be replaced by a list

`method vattrparam : Cil.attrparam -> Cil.attrparam Cil.visitAction`

Attribute parameters.

```
method queueInstr : Cil.instr list -> unit
```

Add here instructions while visiting to queue them to precede the current statement or instruction being processed. Use this method only when you are visiting an expression that is inside a function body, or a statement, because otherwise there will no place for the visitor to place your instructions.

```
method unqueueInstr : unit -> Cil.instr list
```

Gets the queue of instructions and resets the queue. This is done automatically for you when you visit statements.

```
end
```

A visitor interface for traversing CIL trees. Create instantiations of this type by specializing the class `Cil.nopCilVisitor`[5]. Each of the specialized visiting functions can also call the `queueInstr` to specify that some instructions should be inserted before the current instruction or statement. Use syntax like `self#queueInstr` to call a method associated with the current object.

```
class nopCilVisitor : cilVisitor
```

Default Visitor. Traverses the CIL tree without modifying anything

```
val visitCilFile : cilVisitor -> file -> unit
```

Visit a file. This will re-cons all globals TWICE (so that it is tail-recursive). Use `Cil.visitCilFileSameGlobals`[5] if your visitor will not change the list of globals.

```
val visitCilFileSameGlobals : cilVisitor -> file -> unit
```

A visitor for the whole file that does not change the globals (but maybe changes things inside the globals). Use this function instead of `Cil.visitCilFile`[5] whenever appropriate because it is more efficient for long files.

```
val visitCilGlobal : cilVisitor -> global -> global list
```

Visit a global

```
val visitCilFunction : cilVisitor -> fundec -> fundec
```

Visit a function definition

```
val visitCilExpr : cilVisitor -> exp -> exp
```

```
val visitCilLval : cilVisitor -> lval -> lval
```

Visit an lvalue

```
val visitCilOffset : cilVisitor -> offset -> offset
```

Visit an lvalue or recursive offset

```
val visitCilInitOffset : cilVisitor -> offset -> offset
```

Visit an initializer offset

```

val visitCilInstr : cilVisitor -> instr -> instr list
    Visit an instruction

val visitCilStmt : cilVisitor -> stmt -> stmt
    Visit a statement

val visitCilBlock : cilVisitor -> block -> block
    Visit a block

val visitCilType : cilVisitor -> typ -> typ
    Visit a type

val visitCilVarDecl : cilVisitor -> varinfo -> varinfo
    Visit a variable declaration

val visitCilInit : cilVisitor -> varinfo -> offset -> init -> init
    Visit an initializer, pass also the global to which this belongs and the offset.

val visitCilAttributes : cilVisitor -> attribute list -> attribute list
    Visit a list of attributes

```

Utility functions

```

val msvcMode : bool Pervasives.ref
    Whether the pretty printer should print output for the MS VC compiler. Default is GCC.
    After you set this function you should call Cil.initCIL[5].

val useLogicalOperators : bool Pervasives.ref
    Whether to use the logical operands LAnd and LOr. By default, do not use them because
    they are unlike other expressions and do not evaluate both of their operands

val constFoldVisitor : bool -> cilVisitor
    A visitor that does constant folding. Pass as argument whether you want machine specific
    simplifications to be done, or not.

type lineDirectiveStyle =
| LineComment
    Before every element, print the line number in comments. This is ignored by
    processing tools (thus errors are reproted in the CIL output), but useful for visual
    inspection
| LineCommentSparse
    Like LineComment but only print a line directive for a new source line
| LinePreprocessorInput
    Use # nnn directives (in gcc mode)
| LinePreprocessorOutput
    Use #line directives

```

Styles of printing line directives

```
val lineDirectiveStyle : lineDirectiveStyle option Pervasives.ref
```

How to print line directives

```
val print_CIL_Input : bool Pervasives.ref
```

Whether we print something that will only be used as input to our own parser. In that case we are a bit more liberal in what we print

```
val printCilAsIs : bool Pervasives.ref
```

Whether to print the CIL as they are, without trying to be smart and print nicer code. Normally this is false, in which case the pretty printer will turn the while(1) loops of CIL into nicer loops, will not print empty "else" blocks, etc. There is one case however in which if you turn this on you will get code that does not compile: if you use varargs the `__builtin_va_arg` function will be printed in its internal form.

```
val lineLength : int Pervasives.ref
```

The length used when wrapping output lines. Setting this variable to a large integer will prevent wrapping and make #line directives more accurate.

```
val forgcc : string -> string
```

Return the string 's' if we're printing output for gcc, suppress it if we're printing for CIL to parse back in. the purpose is to hide things from gcc that it complains about, but still be able to do lossless transformations when CIL is the consumer

Debugging support

```
val currentLoc : location Pervasives.ref
```

A reference to the current location. If you are careful to set this to the current location then you can use some built-in logging functions that will print the location.

```
val currentGlobal : global Pervasives.ref
```

A reference to the current global being visited

CIL has a fairly easy to use mechanism for printing error messages. This mechanism is built on top of the pretty-printer mechanism (see `Pretty.doc[1]`) and the error-message modules (see `Errormsg.error[2]`).

Here is a typical example for printing a log message:

```
ignore (Errormsg.log "Expression %a is not positive (at %s:%i)\n"
                  d_exp e loc.file loc.line)
```

and here is an example of how you print a fatal error message that stop the execution:

```
Errormsg.s (Errormsg.bug "Why am I here?")
```

Notice that you can use C format strings with some extension. The most useful extension is "%a" that means to consumer the next two argument from the argument list and to apply the first to `unit` and then to the second and to print the resulting `Pretty.doc[1]`. For each major type in CIL there is a corresponding function that pretty-prints an element of that type:

```
val d_loc : unit -> location -> Pretty.doc
```

```

    Pretty-print a location

val d_thisloc : unit -> Pretty.doc
    Pretty-print the Cil.currentLoc[5]

val d_ikind : unit -> ikind -> Pretty.doc
    Pretty-print an integer of a given kind

val d_fkind : unit -> fkind -> Pretty.doc
    Pretty-print a floating-point kind

val d_storage : unit -> storage -> Pretty.doc
    Pretty-print storage-class information

val d_const : unit -> constant -> Pretty.doc
    Pretty-print a constant

val derefStarLevel : int
val indexLevel : int
val arrowLevel : int
val addrOfLevel : int
val additiveLevel : int
val comparativeLevel : int
val bitwiseLevel : int
val getParenthLevel : exp -> int
    Parentheses level. An expression "a op b" is printed parenthesized if its parentheses level is
     $\geq$  that that of its context. Identifiers have the lowest level and weakly binding operators
    (e.g. |) have the largest level. The correctness criterion is that a smaller level MUST
    correspond to a stronger precedence!

class type cilPrinter =
  object
    method pVDecl : unit -> Cil.varinfo -> Pretty.doc
        Invoked for each variable declaration. Note that variable declarations are all the GVar,
        GVarDecl, GFun, all the varinfo in formals of function types, and the formals and
        locals for function definitions.

    method pVar : Cil.varinfo -> Pretty.doc
        Invoked on each variable use.

    method pLval : unit -> Cil.lval -> Pretty.doc
        Invoked on each lvalue occurrence

    method pOffset : Pretty.doc -> Cil.offset -> Pretty.doc

```


Invoked on each offset occurrence. The second argument is the base.

```
method pInstr : unit -> Cil.instr -> Pretty.doc
```

Invoked on each instruction occurrence.

```
method pLabel : unit -> Cil.label -> Pretty.doc
```

Print a label.

```
method pStmt : unit -> Cil.stmt -> Pretty.doc
```

Control-flow statement. This is used by `Cil.printGlobal[5]` and by `Cil.dumpGlobal[5]`.

```
method dStmt : Pervasives.out_channel -> int -> Cil.stmt -> unit
```

Dump a control-flow statement to a file with a given indentation. This is used by `Cil.dumpGlobal[5]`.

```
method dBlock : Pervasives.out_channel -> int -> Cil.block -> unit
```

Dump a control-flow block to a file with a given indentation. This is used by `Cil.dumpGlobal[5]`.

```
method pBlock : unit -> Cil.block -> Pretty.doc
```

```
method pBlock : unit -> Cil.block -> Pretty.doc
```

Print a block.

```
method pGlobal : unit -> Cil.global -> Pretty.doc
```

Global (vars, types, etc.). This can be slow and is used only by `Cil.printGlobal[5]` but not by `Cil.dumpGlobal[5]`.

```
method dGlobal : Pervasives.out_channel -> Cil.global -> unit
```

Dump a global to a file with a given indentation. This is used by `Cil.dumpGlobal[5]`

```
method pFieldDecl : unit -> Cil.fieldinfo -> Pretty.doc
```

A field declaration

```
method pType : Pretty.doc option -> unit -> Cil.typ -> Pretty.doc
```

Use of some type in some declaration. The first argument is used to print the declared element, or is `None` if we are just printing a type with no name being declared. Note that for structure/union and enumeration types the definition of the composite type is not visited. Use `vglob` to visit it.

```
method pAttr : Cil.attribute -> Pretty.doc * bool
```

Attribute. Also return an indication whether this attribute must be printed inside the `__attribute__` list or not.

```

method pAttrParam : unit -> Cil.attrparam -> Pretty.doc
    Attribute parameter

method pAttrs : unit -> Cil.attributes -> Pretty.doc
    Attribute lists

method pLineDirective : ?forcefile:bool -> Cil.location -> Pretty.doc
    Print a line-number. This is assumed to come always on an empty line. If the forcefile
    argument is present and is true then the file name will be printed always. Otherwise
    the file name is printed only if it is different from the last time this function is
    called. The last file name is stored in a private field inside the cilPrinter object.

method pStmtKind : Cil.stmt -> unit -> Cil.stmtkind -> Pretty.doc
    Print a statement kind. The code to be printed is given in the Cil.stmtkind[5]
    argument. The initial Cil.stmt[5] argument records the statement which follows the
    one being printed; Cil.defaultCilPrinterClass[5] uses this information to prettify
    statement printing in certain special cases.

method pExp : unit -> Cil.exp -> Pretty.doc
    Print expressions

method pInit : unit -> Cil.init -> Pretty.doc
    Print initializers. This can be slow and is used by Cil.printGlobal[5] but not by
    Cil.dumpGlobal[5].

method dInit : Pervasives.out_channel -> int -> Cil.init -> unit
    Dump a global to a file with a given indentation. This is used by Cil.dumpGlobal[5]

end

A printer interface for CIL trees. Create instantiations of this type by specializing the class
Cil.defaultCilPrinterClass[5].

class defaultCilPrinterClass : cilPrinter
val defaultCilPrinter : cilPrinter
class plainCilPrinterClass : cilPrinter
    These are pretty-printers that will show you more details on the internal CIL representation,
    without trying hard to make it look like C

val plainCilPrinter : cilPrinter
val printerForMaincil : cilPrinter Pervasives.ref
    zra: This is the pretty printer that Maincil will use. by default it is set to defaultCilPrinter

val printType : cilPrinter -> unit -> typ -> Pretty.doc

```

Print a type given a pretty printer

```
val printExp : cilPrinter -> unit -> exp -> Pretty.doc
```

Print an expression given a pretty printer

```
val printLval : cilPrinter -> unit -> lval -> Pretty.doc
```

Print an lvalue given a pretty printer

```
val printGlobal : cilPrinter -> unit -> global -> Pretty.doc
```

Print a global given a pretty printer

```
val printAttr : cilPrinter -> unit -> attribute -> Pretty.doc
```

Print an attribute given a pretty printer

```
val printAttrs : cilPrinter -> unit -> attributes -> Pretty.doc
```

Print a set of attributes given a pretty printer

```
val printInstr : cilPrinter -> unit -> instr -> Pretty.doc
```

Print an instruction given a pretty printer

```
val printStmt : cilPrinter -> unit -> stmt -> Pretty.doc
```

Print a statement given a pretty printer. This can take very long (or even overflow the stack) for huge statements. Use `Cil.dumpStmt[5]` instead.

```
val printBlock : cilPrinter -> unit -> block -> Pretty.doc
```

Print a block given a pretty printer. This can take very long (or even overflow the stack) for huge block. Use `Cil.dumpBlock[5]` instead.

```
val dumpStmt : cilPrinter -> Pervasives.out_channel -> int -> stmt -> unit
```

Dump a statement to a file using a given indentation. Use this instead of `Cil.printStmt[5]` whenever possible.

```
val dumpBlock : cilPrinter -> Pervasives.out_channel -> int -> block -> unit
```

Dump a block to a file using a given indentation. Use this instead of `Cil.printBlock[5]` whenever possible.

```
val printInit : cilPrinter -> unit -> init -> Pretty.doc
```

Print an initializer given a pretty printer. This can take very long (or even overflow the stack) for huge initializers. Use `Cil.dumpInit[5]` instead.

```
val dumpInit : cilPrinter -> Pervasives.out_channel -> int -> init -> unit
```

Dump an initializer to a file using a given indentation. Use this instead of `Cil.printInit[5]` whenever possible.

```
val d_type : unit -> typ -> Pretty.doc
```

Pretty-print a type using `Cil.defaultCilPrinter[5]`

```

val d_exp : unit -> exp -> Pretty.doc
    Pretty-print an expression using Cil.defaultCilPrinter[5]

val d_lval : unit -> lval -> Pretty.doc
    Pretty-print an lvalue using Cil.defaultCilPrinter[5]

val d_offset : Pretty.doc -> unit -> offset -> Pretty.doc
    Pretty-print an offset using Cil.defaultCilPrinter[5], given the pretty printing for the
    base.

val d_init : unit -> init -> Pretty.doc
    Pretty-print an initializer using Cil.defaultCilPrinter[5]. This can be extremely slow (or
    even overflow the stack) for huge initializers. Use Cil.dumpInit[5] instead.

val d_binop : unit -> binop -> Pretty.doc
    Pretty-print a binary operator

val d_unop : unit -> unop -> Pretty.doc
    Pretty-print a unary operator

val d_attr : unit -> attribute -> Pretty.doc
    Pretty-print an attribute using Cil.defaultCilPrinter[5]

val d_attrparam : unit -> attrparam -> Pretty.doc
    Pretty-print an argument of an attribute using Cil.defaultCilPrinter[5]

val d_attrlist : unit -> attributes -> Pretty.doc
    Pretty-print a list of attributes using Cil.defaultCilPrinter[5]

val d_instr : unit -> instr -> Pretty.doc
    Pretty-print an instruction using Cil.defaultCilPrinter[5]

val d_label : unit -> label -> Pretty.doc
    Pretty-print a label using Cil.defaultCilPrinter[5]

val d_stmt : unit -> stmt -> Pretty.doc
    Pretty-print a statement using Cil.defaultCilPrinter[5]. This can be extremely slow (or
    even overflow the stack) for huge statements. Use Cil.dumpStmt[5] instead.

val d_block : unit -> block -> Pretty.doc
    Pretty-print a block using Cil.defaultCilPrinter[5]. This can be extremely slow (or even
    overflow the stack) for huge blocks. Use Cil.dumpBlock[5] instead.

val d_global : unit -> global -> Pretty.doc
    Pretty-print the internal representation of a global using Cil.defaultCilPrinter[5]. This
    can be extremely slow (or even overflow the stack) for huge globals (such as arrays with lots
    of initializers). Use Cil.dumpGlobal[5] instead.

```

```
val dn_exp : unit -> exp -> Pretty.doc
```

Versions of the above pretty printers, that don't print #line directives

```
val dn_lval : unit -> lval -> Pretty.doc
```

```
val dn_init : unit -> init -> Pretty.doc
```

```
val dn_type : unit -> typ -> Pretty.doc
```

```
val dn_global : unit -> global -> Pretty.doc
```

```
val dn_attrlist : unit -> attributes -> Pretty.doc
```

```
val dn_attr : unit -> attribute -> Pretty.doc
```

```
val dn_attrparam : unit -> attrparam -> Pretty.doc
```

```
val dn_stmt : unit -> stmt -> Pretty.doc
```

```
val dn_instr : unit -> instr -> Pretty.doc
```

```
val d_shortglobal : unit -> global -> Pretty.doc
```

Pretty-print a short description of the global. This is useful for error messages

```
val dumpGlobal : cilPrinter -> Pervasives.out_channel -> global -> unit
```

Pretty-print a global. Here you give the channel where the printout should be sent.

```
val dumpFile : cilPrinter -> Pervasives.out_channel -> string -> file -> unit
```

Pretty-print an entire file. Here you give the channel where the printout should be sent.

the following error message producing functions also print a location in the code. use `Errormsg.bug[2]` and `Errormsg.unimp[2]` if you do not want that

```
val bug : ('a, unit, Pretty.doc) Pervasives.format -> 'a
```

Like `Errormsg.bug[2]` except that `Cil.currentLoc[5]` is also printed

```
val unimp : ('a, unit, Pretty.doc) Pervasives.format -> 'a
```

Like `Errormsg.unimp[2]` except that `Cil.currentLoc[5]` is also printed

```
val error : ('a, unit, Pretty.doc) Pervasives.format -> 'a
```

Like `Errormsg.error[2]` except that `Cil.currentLoc[5]` is also printed

```
val errorLoc : location -> ('a, unit, Pretty.doc) Pervasives.format -> 'a
```

Like `Cil.error[5]` except that it explicitly takes a location argument, instead of using the `Cil.currentLoc[5]`

```
val warn : ('a, unit, Pretty.doc) Pervasives.format -> 'a
```

Like `Errormsg.warn[2]` except that `Cil.currentLoc[5]` is also printed

```
val warnOpt : ('a, unit, Pretty.doc) Pervasives.format -> 'a
```

Like `Errormsg.warnOpt[2]` except that `Cil.currentLoc[5]` is also printed. This warning is printed only if `Errormsg.warnFlag[2]` is set.

```
val warnContext : ('a, unit, Pretty.doc) Pervasives.format -> 'a
```

Like `Errormsg.warn[2]` except that `Cil.currentLoc[5]` and context is also printed

```
val warnContextOpt : ('a, unit, Pretty.doc) Pervasives.format -> 'a
```

Like `Errormsg.warn[2]` except that `Cil.currentLoc[5]` and context is also printed. This warning is printed only if `Errormsg.warnFlag[2]` is set.

```
val warnLoc : location -> ('a, unit, Pretty.doc) Pervasives.format -> 'a
```

Like `Cil.warn[5]` except that it explicitly takes a location argument, instead of using the `Cil.currentLoc[5]`

Sometimes you do not want to see the syntactic sugar that the above pretty-printing functions add. In that case you can use the following pretty-printing functions. But note that the output of these functions is not valid C

```
val d_plainexp : unit -> exp -> Pretty.doc
```

Pretty-print the internal representation of an expression

```
val d_plaininit : unit -> init -> Pretty.doc
```

Pretty-print the internal representation of an integer

```
val d_plainlval : unit -> lval -> Pretty.doc
```

Pretty-print the internal representation of an lvalue

Pretty-print the internal representation of an lvalue offset `val d_plainoffset: unit → offset → Pretty.doc`

```
val d_plaintype : unit -> typ -> Pretty.doc
```

Pretty-print the internal representation of a type

ALPHA conversion has been moved to the Alpha module.

```
val uniqueVarNames : file -> unit
```

Assign unique names to local variables. This might be necessary after you transformed the code and added or renamed some new variables. Names are not used by CIL internally, but once you print the file out the compiler downstream might be confused. You might have added a new global that happens to have the same name as a local in some function. Rename the local to ensure that there would never be confusion. Or, viceversa, you might have added a local with a name that conflicts with a global

Optimization Passes

```
val peepHole2 : (instr * instr -> instr list option) -> stmt list -> unit
```

A peephole optimizer that processes two adjacent statements and possibly replaces them both. If some replacement happens, then the new statements are themselves subject to optimization

```
val peepHole1 : (instr -> instr list option) -> stmt list -> unit
```

Similar to `peepHole2` except that the optimization window consists of one statement, not two

Machine dependency

```
exception SizeOfError of string * typ
```

Raised when one of the `bitsSizeOf` functions cannot compute the size of a type. This can happen because the type contains array-length expressions that we don't know how to compute or because it is a type whose size is not defined (e.g. `TFun` or an undefined `compinfo`). The string is an explanation of the error

val `bitsSizeOf` : `typ` -> `int`

The size of a type, in bits. Trailing padding is added for structs and arrays. Raises `Cil.SizeOfError[5]` when it cannot compute the size. This function is architecture dependent, so you should only call this after you call `Cil.initCIL[5]`. Remember that on GCC `sizeof(void)` is 1!

val `sizeof` : `typ` -> `exp`

The size of a type, in bytes. Returns a constant expression or a "sizeof" expression if it cannot compute the size. This function is architecture dependent, so you should only call this after you call `Cil.initCIL[5]`.

val `alignOf_int` : `typ` -> `int`

The minimum alignment (in bytes) for a type. This function is architecture dependent, so you should only call this after you call `Cil.initCIL[5]`.

val `bitsOffset` : `typ` -> `offset` -> `int` * `int`

Give a type of a base and an offset, returns the number of bits from the base address and the width (also expressed in bits) for the subobject denoted by the offset. Raises `Cil.SizeOfError[5]` when it cannot compute the size. This function is architecture dependent, so you should only call this after you call `Cil.initCIL[5]`.

val `char_is_unsigned` : `bool` `Pervasives.ref`

Whether "char" is unsigned. Set after you call `Cil.initCIL[5]`

val `little_endian` : `bool` `Pervasives.ref`

Whether the machine is little endian. Set after you call `Cil.initCIL[5]`

val `underscore_name` : `bool` `Pervasives.ref`

Whether the compiler generates assembly labels by prepending "_" to the identifier. That is, will function `foo()` have the label "foo", or "_foo"? Set after you call `Cil.initCIL[5]`

val `locUnknown` : `location`

Represents a location that cannot be determined

val `get_instrLoc` : `instr` -> `location`

Return the location of an instruction

val `get_globalLoc` : `global` -> `location`

Return the location of a global, or `locUnknown`

val `get_stmtLoc` : `stmtkind` -> `location`

Return the location of a statement, or `locUnknown`

```

val dExp : Pretty.doc -> exp
    Generate an Cil.exp[5] to be used in case of errors.

val dInstr : Pretty.doc -> location -> instr
    Generate an Cil.instr[5] to be used in case of errors.

val dGlobal : Pretty.doc -> location -> global
    Generate a Cil.global[5] to be used in case of errors.

val mapNoCopy : ('a -> 'a) -> 'a list -> 'a list
    Like map but try not to make a copy of the list

val mapNoCopyList : ('a -> 'a list) -> 'a list -> 'a list
    Like map but each call can return a list. Try not to make a copy of the list

val startsWith : string -> string -> bool
    sm: return true if the first is a prefix of the second string

```

An Interpreter for constructing CIL constructs

```

type formatArg =
  | Fe of exp
  | Feo of exp option
      For array lengths

  | Fu of unop
  | Fb of binop
  | Fk of ikind
  | FE of exp list
      For arguments in a function call

  | Ff of (string * typ * attributes)
      For a formal argument

  | FF of (string * typ * attributes) list
      For formal argument lists

  | Fva of bool
      For the ellipsis in a function type

  | Fv of varinfo
  | Fl of lval
  | Flo of lval option
  | Fo of offset
  | Fc of compinfo
  | Fi of instr
  | FI of instr list
  | Ft of typ
  | Fd of int
  | Fg of string

```



```

| Fs of stmt
| FS of stmt list
| FA of attributes
| Fp of attrparam
| FP of attrparam list
| FX of string

```

The type of argument for the interpreter

```
val d_formatarg : unit -> formatArg -> Pretty.doc
```

Pretty-prints a format arg

```
val lowerConstants : bool Pervasives.ref
```

Do lower constant expressions into constants (default true)

```
val warnTruncate : bool Pervasives.ref
```

Emit warnings when truncating integer constants (default true)

6 Module Formatcil : An Interpreter for constructing CIL constructs

```
val cExp : string -> (string * Cil.formatArg) list -> Cil.exp
```

Constructs an expression based on the program and the list of arguments. Each argument consists of a name followed by the actual data. This argument will be placed instead of occurrences of "%v.name" in the pattern (where the "v" is dependent on the type of the data). The parsing of the string is memoized. * Only the first expression is parsed.

```
val cLval : string -> (string * Cil.formatArg) list -> Cil.lval
```

Constructs an lval based on the program and the list of arguments. Only the first lvalue is parsed. The parsing of the string is memoized.

```
val cType : string -> (string * Cil.formatArg) list -> Cil.typ
```

Constructs a type based on the program and the list of arguments. Only the first type is parsed. The parsing of the string is memoized.

```
val cInstr :
```

```
string -> Cil.location -> (string * Cil.formatArg) list -> Cil.instr
```

Constructs an instruction based on the program and the list of arguments. Only the first instruction is parsed. The parsing of the string is memoized.

```
val cStmt :
```

```
string ->
```

```
(string -> Cil.typ -> Cil.varinfo) ->
```

```
Cil.location -> (string * Cil.formatArg) list -> Cil.stmt
```

```
val cStmts :
```

```

string ->
(string -> Cil.typ -> Cil.varinfo) ->
Cil.location -> (string * Cil.formatArg) list -> Cil.stmt list
    Constructs a list of statements

val dExp : string -> Cil.exp -> Cil.formatArg list option
    Deconstructs an expression based on the program. Produces an optional list of format
    arguments. The parsing of the string is memoized.

val dLval : string -> Cil.lval -> Cil.formatArg list option
    Deconstructs an lval based on the program. Produces an optional list of format arguments.
    The parsing of the string is memoized.

val dType : string -> Cil.typ -> Cil.formatArg list option
    Deconstructs a type based on the program. Produces an optional list of format arguments.
    The parsing of the string is memoized.

val dInstr : string -> Cil.instr -> Cil.formatArg list option
    Deconstructs an instruction based on the program. Produces an optional list of format
    arguments. The parsing of the string is memoized.

val noMemoize : bool Pervasives.ref
    If set then will not memoize the parsed patterns

val test : unit -> unit
    Just a testing function

```

7 Module Alpha : ALPHA conversion

```

type 'a undoAlphaElement
    This is the type of the elements that are recorded by the alpha conversion functions in order
    to be able to undo changes to the tables they modify. Useful for implementing scoping

type 'a alphaTableData
    This is the type of the elements of the alpha renaming table. These elements can carry some
    data associated with each occurrence of the name.

val newAlphaName :
  alphaTable:(string, 'a alphaTableData Pervasives.ref) Hashtbl.t ->
  undolist:'a undoAlphaElement list Pervasives.ref option ->
  lookupname:string -> data:'a -> string * 'a

```

Create a new name based on a given name. The new name is formed from a prefix (obtained from the given name by stripping a suffix consisting of `_` followed by only digits), followed by a special separator and then by a positive integer suffix. The first argument is a table mapping name prefixes to some data that specifies what suffixes have been used and how to create the new one. This function updates the table with the new largest suffix generated. The "undolist" argument, when present, will be used by the function to record information that can be used by `Alpha.undoAlphaChanges[7]` to undo those changes. Note that the undo information will be in reverse order in which the action occurred. Returns the new name and, if different from the lookupname, the location of the previous occurrence. This function knows about the location implicitly from the `Cil.currentLoc[5]`.

```
val registerAlphaName :
  alphaTable:(string, 'a alphaTableData Pervasives.ref) Hashtbl.t ->
  undolist:'a undoAlphaElement list Pervasives.ref option ->
  lookupname:string -> data:'a -> unit
  Register a name with an alpha conversion table to ensure that when later we call
  newAlphaName we do not end up generating this one
```

```
val docAlphaTable :
  unit ->
  (string, 'a alphaTableData Pervasives.ref) Hashtbl.t -> Pretty.doc
  Split the name in preparation for newAlphaName. The prefix returned is used to index into
  the hashtable. The next result value is a separator (either empty or the separator chosen to
  separate the original name from the index)
```

```
val getAlphaPrefix : lookupname:string -> string
val undoAlphaChanges :
  alphaTable:(string, 'a alphaTableData Pervasives.ref) Hashtbl.t ->
  undolist:'a undoAlphaElement list -> unit
  Undo the changes to a table
```