

GCC Front-End Internals

Andi Hellmund
gcc@andihellmund.com

March 6, 2011
version 0.1

Abstract

The GNU Compiler Collection (GCC) is a set of compilers intended for translating different high-level programming languages (like C, C++, Java) to executable object code runnable on different target platforms (like x86, x86_64, SPARC, ARM, PowerPC, PA-RISC or Itanium). This flexibility is achieved by a modular architecture composed of front-end (language-dependent), middle-end (language- and target-platform-independent) and back-end (target-platform-dependent). GCC and its extensible front-end architecture is the focus of this article. It describes the general structure, the required files and the exposed APIs to create a new language front-end.

1 Introduction

The GNU Compiler Collection (GCC) is a set of open-source compilers published under the GNU Public License (GPL) which allows you to freely use, modify and re-distribute the source code. GCC was initially written by Richard Stallman in 1987 as the fundamental building block of the GNU project. Meanwhile, it is widely used in the GNU/Linux community and for the Linux-based software development of many software companies.

GCC now exists for more than 23 years and it has massively evolved since then. The current version 4.6.0 has more than 2 million lines of code, while it supports 7 programming languages[4], more than 30 target architectures[1] and a very high number of compiler optimizations. Due to its open nature and wide distribution, GCC has also gained a high attractiveness for compiler research projects like GRAPHITE[11] (polyhedral representation for loop optimizations). A recent development approach in the GCC community is to further modularize the GCC architecture[12] to achieve an even stricter separation between the various components of the compiler. Therefore, the interfaces for the GCC

front-end development might change slightly from version to version. Nevertheless, to set a common base for this article, I will reference the GCC front-end interface as available in version 4.6.0 (available from [3]).

The architecture of GCC is divided into three main parts: source language-dependent front-end, language- and target-independent middle-end (optimizations, SSA transformation) and target-dependent back-end (target-specific optimizations and code generation). Figure 1 depicts the very high-level overview of the GCC architecture including the used intermediate representations.

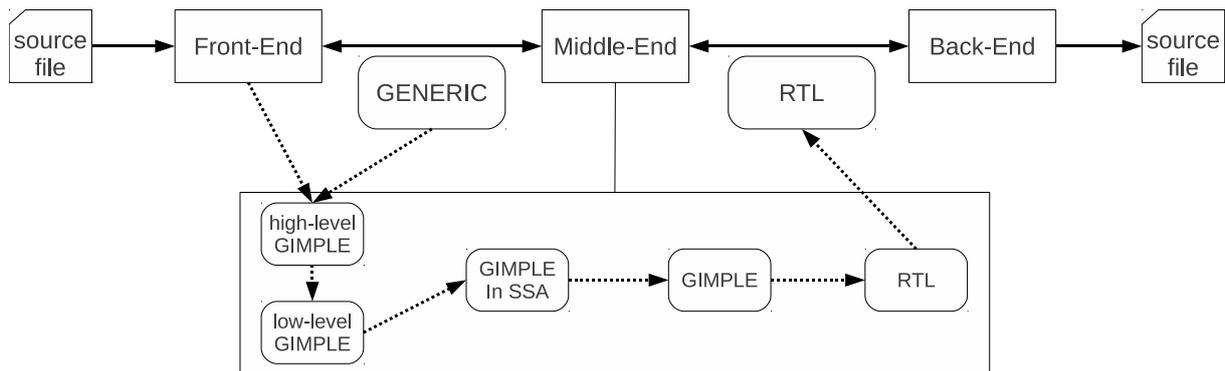


Figure 1: GCC Infrastructure and IRs

1.1 Bootstrapping

A bootstrapping compiler is a compiler which is able to compile its own source code. This implies that the compiler is written in the programming language which it is supposed to translate. The GCC build process enables bootstrap compiling for the C compiler so that the C compiler, for example released with your GNU/Linux distribution, is finally compiled by itself. How does this build process work?

The prerequisite for a GCC bootstrap build is an already working C compiler capable to translate the GCC source code¹. This compiler will further on be called *host compiler*. The GCC bootstrap process of the C compiler is divided into three stages while each stage produces new compiler executables by compiling the whole C compiler source tree:

1. The *host compiler* creates a stage1 GCC compiler
2. The stage1 compiler creates a stage2 GCC compiler
3. In the last step, the stage2 GCC compiler creates a stage3 GCC compiler.

¹This is important because the source code uses GNU-specific extensions to the C standard, e.g. the `__attribute__` syntax

The stage2 and stage3 compilers are finally compared for binary equality. If unequal, the bootstrap revealed an error in the compiler. If equal, the stage2 and stage3 compilers are both bootstrapped final compilers. One question might still be opened: why is it necessary to build the stage3 compiler? The reason for this is that the stage2 compiler's object code (assembly code) is really created by its own code generation, while the object code of the stage1 compiler was generated by the host compiler. If the stage3 compiler however would not have been built, the (self-)generated object code of the stage2 compiler would not have been tested for execution. Though, the stage3 compiler serves the testing purpose of the self-generated object code. Compilers other than the C compiler are only built in stage2 and stage3 using the bootstrapped C compiler so that all compilers finally benefit from the most recent features and code optimizations.

1.2 Compiler driver versus compiler proper

If I would ask many people what the executable gcc is doing, most of the people would answer, 'Well, its a compiler, though, its compiling the source file?'. No no, that is not correct. The executable gcc is not a compiler although this abbreviation means GNU C compiler². gcc represents what is generally called a compiler driver (in the following just called driver). If you now think that I am completely insane, please just add the -v option to one of your gcc commands and see what gcc is really doing. Interesting, right!?

While a compiler proper (in the following just called compiler) is responsible for transforming the source file into possibly optimized target machine code, the driver is the high-level "organizer" in the overall compilation process creating a shared library, object or executable file from one or more source files. Thereby, the driver divides the compilation process into several phases. gcc in version 4.6.0 uses the following phases as shown in figure 2 assuming the command 'gcc *.c -o exe' is executed.

In earlier versions of gcc, the driver added a separate pre-processing phase before the compiler phase, but in recent versions the pre-processor phase is omitted, because the compiler (cc1) incorporates the pre-processor, at least when using the default options³

Assuming the default options, the input C source file gets translated into an assembly file (.s extension) which then gets assembled into an object file by the assembler (usually gas from the GNU binutils package). The object file (.o extension) is then finally linked into

²There is a clear distinction between GCC and gcc. While GCC contains the whole set of compilers, gcc only relates to the C programming language.

³By using the -no-integrated-cpp option, you instruct the driver to split the compiler phase into 2 distinct phases: pre-processing and compilation proper.

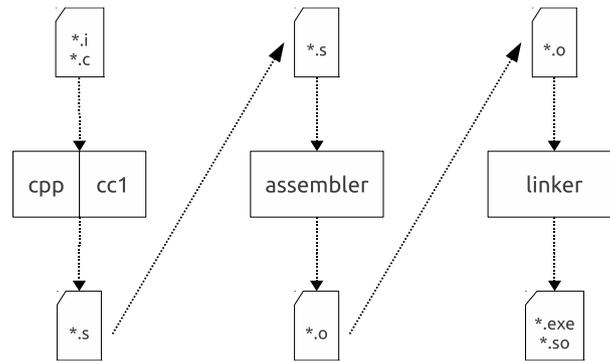


Figure 2: GCC compilation driver

an executable file by the linker.⁴ Unless using the `-pipe` option, the driver uses temporary files to pass the result of one phase to the subsequent phase. If the `-save-temps` option is used, the temporary files of all phases are stored to the current working directory.

The gcc driver selects the starting phase by looking at the file extension of the input files though it is possible to start at any of the 4 phases listed in the picture. Furthermore, gcc allows to stop after any of the 4 phases by specifying the appropriate option, while the default behavior of gcc is to run through all of the phases to produce a ready-to-run executable:

- `-E` : stop after the pre-processing
- `-S` : stop after the compiler
- `-c` : stop after the assembler
- *default* : stop after collect2/linker

A short example about how the driver works; assume the following gcc command `'gcc 1.c 2.s 3.o -o exe'`: firstly, the driver would run the C source file `1.c` through the first two phases and produce a temporary file `tmp1.o`⁵. Then the assembly file `2.s` would be run through the assembler phase and a temporary file `tmp2.o` would be created. And finally, the three object files `tmp1.o`, `tmp2.o` and `3.o` will be passed to the linker to create an executable file.

1.3 Source code hierarchy

Figure 3 shows a fundamental structure of the GCC source code. Directories that are required for front-end creation are marked bold. As indicated in figure 3, when talking

⁴In default installations, the linker (`ld`) is not called directly, but indirectly via a linker wrapper called `collect2`[6].

⁵GCC uses more cryptic temporary file names, though `tmp1.o` is just used for exemplary purpose.

about gcc-x.y.z, I am referencing the top-level directory of the extracted gcc archive in the further course of this article.

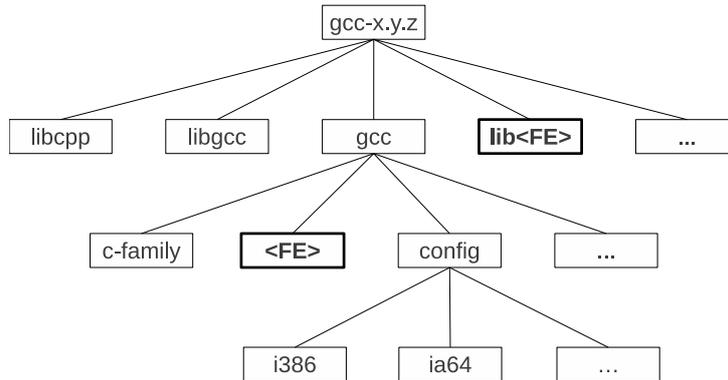


Figure 3: GCC source code hierarchy

1.4 Exemplary front-end

For demonstration purpose of this article, I uploaded the source code of a sample GCC front-end, called `gcalc`, to the GCC-wiki[5]. `gcalc` is a front-end for a very simple language purely working with integer types and arithmetic operators (+,-,*,/) and furthermore supporting a function call to print out values to the screen. The `gcalc` front-end also employs a runtime library (like C++ runtime) whose only implementation is the `print()` function.

The `gcc` configure and build steps are not the focus of this article, though please reference the official guide at [2]. When trying to build `gcalc`, please firstly extract the GCC front-end archive into `gcc-4.6.0` and make sure that `gcalc` is added to the configure option `-enable-languages`.

1.5 Front-end files

Beside the real implementation of the front-end using C/C++ source code, each language front-end requires a set of mandatory files as listed below. Optional files follow the mandatory files and are marked specifically.

config-lang.in Mandatory. General front-end language configuration used by the configure/make build process, like the name of the language (parameter: *language*) or the file name of the compiler (parameter: *compilers*) among others. This file gets read by the configure script and the contents get incorporated into the generated

makefiles. For a detailed description of the single parameters, please refer [7] and [16].

lang-specs.h Mandatory. This contains the specification used by the GCC driver to handle the specific phases of the compilation process. As described earlier, GCC selects the phases and corresponding tools (e.g. compiler) based on file extensions. Though assume, as an example, you want to create a new language which should not be directly translated into machine code, but beforehand into C code. By using the lang-specs.h file, you could then instruct your language driver to pass your new language file to your compiler which creates a .c file. This .c file would then further be processed by the common tools, e.g. cc1, as, ld, etc. For reference, please check the file gcc-x.y.z/gcc/gcc.c which greatly explains the syntax of the lang-specs.h file. It's very complex.

Make-lang.in Mandatory.

Language-specific makefile fragment. This fragment gets included into the generated gcc makefile. This makefile should contain the instructions to build and install the language-specific driver, compiler, man pages and documentation. Chapter ?? provides more insights into the details of the makefile fragment.

lang.opt Mandatory.

Language-specific compiler options which are automatically parsed by GCC common code and passed to a front-end specific function for final analysis and internal processing. This file is mandatory due to the *Language* directive which gets transformed into a C macro (e.g. CL_gcalc) whose value is finally required for the option processing code to determine which options are appropriate for which compilers.

For a sample file layout, check out the files gcc-x.y.z/gcc/common.opt. The syntax of this file is explained in [10].

'lang'-tree.def, e.g. gcalc-tree.def Optional.

To simplify the creation of new front-ends and the interaction of the front-end with the middle-end/back-end, GCC provides a language-independent abstract-syntax tree named GENERIC[15]. Details and internals of GENERIC will be presented in the further course of this article. The short version: GENERIC is a tree-based representation while each tree node has a unique tree code. All the available tree codes are listed in the file gcc-x.y.z/gcc/tree.def. Most of these tree codes might suffice the purposes of a new language front-end, but you might need additional ones for specific language constructs. These extra tree codes are put into a language-specific tree definition file. The naming convention for this file is to use the front-end directory as the first part of the name instead of the *language* parameter in the config-lang.in

file. For example, the C++ front-end located in the directory `gcc-x.y.z/gcc/cp` defines the file `gcc-x.y.z/gcc/cp/cp-tree.def`. For a sample layout, please check the file `gcc-x.y.z/gcc/tree.def`.

2 Driver

The driver is quite easy to implement since most of the code (e.g. option handling or language spec processing) is already implemented in `gcc-x.y.z/gcc/gcc.c`. A front-end however must provide an implementation and definition for these three functions and symbols:

lang_specific_driver Function.

This function is called in `process_command` (`gcc-x.y.z/gcc/gcc.c`) and allows the front-end driver to add front-end specific flags before the command gets processed. The C++ front-end for example uses this function to add the `-lstdc++` flag to always link C++ executable to the C++ standard library. Otherwise, users of `g++` would have to specify `-lstdc++` for each and every compilation. By the way, this is the reason why it is required to compile C++ applications with `g++` even though `gcc` would be able to compile the C++ source files. But `gcc` wouldn't add the C++ specific link requirements. Why is `gcc` able to compile C++ source code? Remember, the compiler driver works based on file extensions and each driver knows the language specs of each front-end language enabled via the configure script.

lang_specific_pre_link Function.

This function is called in `main` (`gcc-x.y.z/gcc/gcc.c`) before the linking step. Most of the available front-ends do not use this function at all, but it could be used, for example, to add further object files required to build an executable in the link step. This function does not take any input parameters, but the important data structures, `outfiles` and `lang_specific_extra_outfiles` are globally defined.

lang_specific_extra_outfiles Variable.

This variable holds the number of files added in `lang_specific_pre_link()` to the `outfiles` array. The `outfiles` array holds the names of output files, e.g. `/tmp/file.o` when compiling `file.c` to an executable. If used, this variable must be set in `lang_specific_driver` since this function is called before the size of the `outfiles` array (dependent on `lang_specific_extra_outfiles`) is allocated. When `lang_specific_pre_link()` is an empty implementation, this variable should be set to zero.

Note: it is not required to have such a compiler driver. If your language does not have specific command-line option or linker requirements, it is acceptable to omit the compiler

driver. In this case, you could use the default gcc driver (executable 'gcc') to locate and call your compiler executable by specifying the *-x language* option to gcc. The order of input source files and the *-x* option is important. The *-x* option must be specified before any of the corresponding input files.

3 Compiler

As depicted in figure 1, a compiler in the GCC world consists of front-end, middle-end and back-end. The basic idea behind GCC's modularity is that the middle-end provides a compiler framework containing the general flow of control, option processing, intermediate representations and code optimizations. This framework is completely independent of the front-end and back-end being used so that these two components could be exchanged on the fly, currently only at build-time of GCC. The communication between the framework and front-end/back-end is done via function pointers and attributes bundled into so called language and target hooks. Target hooks are beyond the scope of this article. Though, each language front-end provides such a language hook structure which the framework then uses to call back into the front-end, where required, e.g. to parse the input file.

GCC currently supports more than 50 language hooks which serve the various language-specific purpose: option processing, source file scanning and parsing and the general interaction with the middle-end e.g. by return a list of global declarations in the program. It is far beyond the scope of this article to discuss all of them, since most of them are quite specific (e.g. GOMP related language hooks), but I will go through the mostly used and general purpose language hooks. But before I describe the language hooks, I will shortly go through one of GCC's high-level intermediate languages, called GENERIC, whereof some knowledge is beneficial for the following sections.

3.1 GENERIC

GENERIC is one of GCC's intermediate representations (IR) of input source programs. The two other intermediate representations are GIMPLE (tuple-based representation) and RTL (lisp-like representation of low-level operations). This article assumes that a front-end firstly generates GENERIC and then uses the middle-end to translate GENERIC into GIMPLE, a process called gimplification. However, a front-end must not use GENERIC as IR. The middle-end's dominating IR is GIMPLE so that a front-end is free to employ whatever IR is appropriate and finally translate this proprietary IR into GIMPLE before passing control back to the middle-end. Even though GIMPLE might be useful for a

front-end, this article only covers GENERIC, because I think that its generation is easier and more straightforward.

GENERIC is a tree-based intermediate representation whose tree nodes are mostly oriented to programming languages like C/C++. Each of GENERIC's tree nodes has a unique tree code, e.g. a tree code for a variable declaration or for a plus (addition) expression. All the tree codes covered by GENERIC are listed in `gcc-x.y.z/gcc/tree.def`. If the coverage of GENERIC is not sufficient for a front-end however, a front-end might define new tree codes in file `'lang'-tree.def` as explained above. Once a front-end extends GENERIC by self-defined tree codes, these additional tree codes must be manually translated into GIMPLE by the front-end in the language hook `LANG_HOOK_GIMPLIFY_EXPR` as described below. The documentation of the GENERIC tree codes in `gcc-x.y.z/gcc/tree.def` is already very good, so that I will focus only on a few details.

First of all, for those who are a kind of unfamiliar with tree-based IR, figure 4 shows how a simple declaration like `'int a'` might schematically look like in GENERIC.

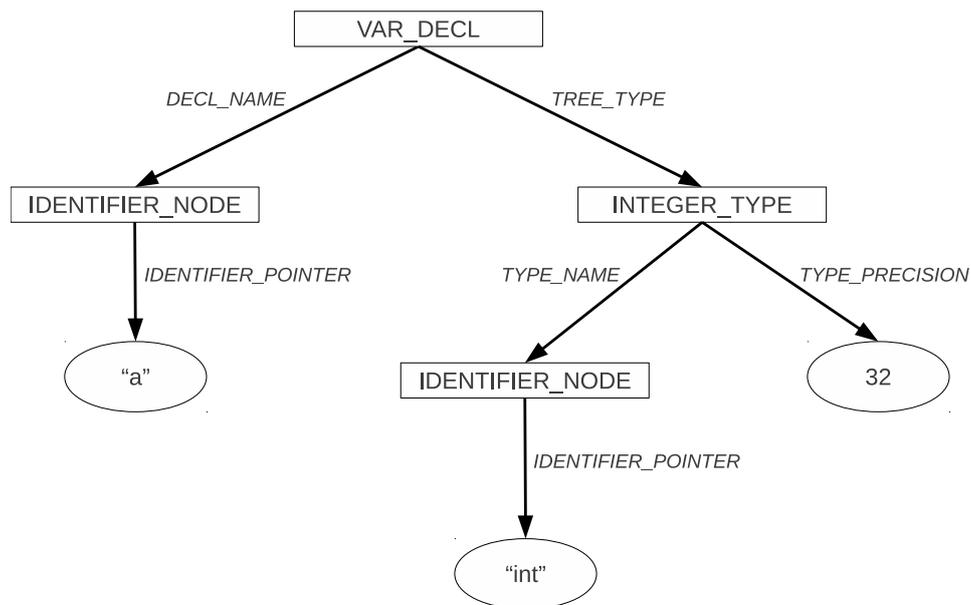


Figure 4: Graphical representation of GENERIC

GENERIC tree nodes are represented in C code by a union named `tree_node` (defined in `gcc-x.y.z/gcc/tree.h`). This union has a lot of members which serve the specific purposes of tree nodes, e.g. members for declarations, constants or expressions⁶. In terms of implementation, there are at least two ways to create these tree node structures from C code: the first way is to create them manually via the `make_node(tree_code)` function which

⁶If you might wonder about this style of coding, please keep in mind that the C programming languages lacks object-oriented features. The intention of this implementation is to realize a struct hierarchy with "struct inheritance".

allocates the memory for the structure and finally returns a pointer to a `tree_node` union (which in GCC is typedefed to `'tree'`⁷). I would not generally recommend this approach. The second way is to use GENERIC build-wrapper as defined in `gcc-x.y.z/gcc/tree.c`. For example, there is a build wrapper to create a new variable declaration (`build_decl`) or a binary expression (`build2`). The declaration depicted in figure 4 could be coded by

```
tree vardecl = build_decl(BUILTINS_LOCATION, VAR_DECL,
                          get_identifier("a"), integer_type_node);
```

3.2 Language Hooks

All of the language hooks are defined in the header file `gcc-x.y.z/gcc/langhooks.h`. A front-end must not provide a definition for each of the language hooks. If a language hook is not defined, a default definition will be taken which in the worst case just does nothing or results in a compiler error (in GCC terms called ICE for internal compiler error). The default language hooks are listed in `gcc-x.y.z/gcc/langhooks-def.h`. This file also lists the macros to re-define individual language hooks (e.g. `LANG_HOOKS_PARSE_FILE`) and provides the initializer macro (`LANG_HOOKS_INITIALIZER`) to finally define the language hooks structure in the front-end. A common set of instructions would look like the following to re-define a language hook and define the front-specific language hook structure. By the way, it is important to name the language hooks structure `'lang_hooks'`.

```
#undef LANG_HOOKS_PARSE_FILE
#define LANG_HOOKS_PARSE_FILE calc_parse_file

...

// create the language hook structure to be used by the
// GCC compiler framework
struct lang_hooks lang_hooks = LANG_HOOKS_INITIALIZER;
```

This language hook structure - whose members are functions pointers and variables/attributes - is then used by the GCC compiler framework to call back into the front-end to do the language-dependent processing or to retrieve language-dependent attributes. The following list describes the language hooks which I think are generally used. For non-listed language hooks, please check out the comments in `gcc-x.y.z/gcc/langhooks.h`; the documentation is quite good. If you want to dive deeper into the logic of GCC and where quite specific language hooks are called from, search the source code for `lang_hooks.'member name'`, e.g. `lang_hooks.parse_file` for the `LANG_HOOKS_PARSE_FILE` language hook.

⁷If you are new to GCC, you might not find this type definition immediately, unless you are using a very good source code search tool like `cscope`, then please look at `gcc-x.y.z/gcc/coretypes.h`

LANG_HOOKS_NAME Attribute.

This language hook holds the name of the front-end, e.g. "GNU C++" in the case of C++. The name of the front-end is displayed in GCC verbose (-v option) mode, for example.

LANG_HOOKS_INIT Function.

It may be used by a front-end to e.g. initialize internal data structures like symbol tables, etc. To work properly, a front-end is required to call the following functions which are usually called from this init function:

- **build_common_tree_nodes:** creates GENERIC type nodes for all integer types known from C programming languages. These integer types are required by the middle-end and may be used by a front-end to handle integer type data declarations.
- **set_sizetype:** The size_t type known from C/C++ programming (e.g. as return type of the sizeof operator) is also an integer type, but a front-end must specifically set the size of the internal, corresponding GENERIC type. This is again required by the middle-end.
- **build_common_tree_nodes_2:** creates GENERIC type nodes for various other types, e.g. like floating point types.

LANG_HOOKS_OPTION_LANG_MASK Function.

This function is used by the option handling part of GCC and it is required to return a mask unique to this front-end, e.g. CL_gcalc. This mask is then used to check if supplied options are valid for this front-end.

LANG_HOOKS_INIT_OPTIONS Function.

This function might be used to initialize the front-end's specific options with default values.

LANG_HOOKS_HANDLE_OPTION Function.

The GCC option handling part calls this function for each option that is passed to the compiler. To distinguish the various options, enum values (included via options.h) are available which take the form of OPT_*, e.g. OPT_fprofile_use_ for -fprofile-use.

LANG_HOOKS_POST_OPTIONS Function.

Once all options were processed, the front-end could post-process the specified options in this language hook.

LANG_HOOKS_FINISH Function.

This is the counter-part language hook to `LANG_HOOKS_INIT` and might be used to clean-up allocated data structures.

LANG_HOOKS_PARSE_FILE Function.

This functions gets called by the compiler framework when the input source file(s) should be scanned and parsed into intermediate representation.

LANG_HOOKS_GETDECLS Function.

This function gets called from the middle-end (`write_global_declarations()`) and is expected to return the global declarations to be written out into the assembly file. Function declarations are not written out by this function, but they should be nevertheless put into the list of global declarations, because useful warnings (e.g. unused functions) are printed by `write_global_declarations`. By the way, `write_global_declarations` itself is a language-hook and could be overwritten if it does not meet the front-end needs.

LANG_HOOKS_GLOBAL_BINDINGS_P Function.

Called from multiple call sites, e.g. `fold_range_test` (`gcc-x.y.z/gcc/fold-const.c`) This function is expected to return a non-zero value if the current scope of the parser is global one. This again is quite C/C++ centric, but yes, the GNU Compiler Collection was originally developed as a C/C++ compiler and hence might possibly not fit all new languages which are very different from procedural languages. A front-end must define this function.

LANG_HOOKS_PUSHDECL Function.

The purpose of this language hook is to add a declaration to the current lexical scope. As imposed by the lexical scope, this language hook is quite C/C++ centric. Unfortunately however, this function must be defined because a default language hook does not exist. Though, a front-end must define this function, but it is not required by the middle-end.

LANG_HOOKS_TYPE_FOR_SIZE Function.

This function gets called from some middle-end optimization passes (e.g. constant folding) with two parameters: a precision value and a signedness flag. This function is then expected to return an appropriate integer type node with at least the input precision and the specified signedness.

LANG_HOOKS_TYPE_FOR_MODE Function.

This language hook is quite similar to `LANG_HOOKS_TYPE_FOR_SIZE`, but the first parameter is not a size parameter but a 'mode' parameter. Modes are used by back-ends to represent the various "raw" integer and floating point processor types, e.g. 1-byte integer. For a more detailed explanation of all the modes, please refer

to [9]. This function is then expected to return an integer type node for the given 'mode'.

LANG_HOOKS_GIMPLIFY_EXPR Function.

As mentioned above, this is the language hook which is called by the middle-end to transform a GENERIC node into a GIMPLE tuple. The value returned by this function is very important for the middle-end, so that I will list each of the possible values (defined in gcc-x.y.z/gcc/gimple.h):

- **GS_ERROR**: Error condition. Not good!
- **GS_UNHANDLED**: This return value indicates that the front-end has no idea how to transform this code and requests the middle-end to do the transformation. The default language hook for example always returns this value.
- **GS_OK**: This return code indicates that a partial (possibly front-end specific) transformation was done, but that further processing might be necessary. The middle-end will then check if further processing is required.
- **GS_ALL_DONE**: If the front-end completely transformed the GENERIC code into GIMPLE so that no further action is required by the middle-end, it will return GS_ALL_DONE.

Unfortunately, there is one exception. The framework requires a function `convert()` to be defined by the front-end which is not contained in the language hooks data structure. The `convert()` function is called from multiple locations in the middle-end to convert an expression to a new type, though to do a type cast. `convert()` accepts two parameters, the first one is the type to convert to and the second one is the expression to convert. It is expected to return a new expression with the new type.

3.3 Front-end data structures

Beside the language hooks, the GCC framework also expects the front-end code to define a certain set of C structs and unions. The following list gives a short description of each data structure. Once you look through the code of some GCC front-ends and GCC in general, you will recognize `GTY()` expressions in some of GCC data structures including those required by the front-end. The `GTY()` expressions are part of the GCC-internal garbage collection to automatically free resources of important data structures (like union `tree_node`). I will not go into the details of the garbage collection system, but the interested reader might want to have a look at [8] for further details.

struct lang_decl Data.

This structure is part of the data structure `tree_decl_common` which is one of the base structures (remember the hierarchy) for the GENERIC/tree-based data structures specific to declarations (variable, param, record members). A front-end might then put language-specific data into these generic data structures. If not used, it is allowed to put in a dummy member (e.g. of type `char`) to satisfy the compiler.

struct lang_type Data.

The same like `lang_decl`, but related to type-specific GENERIC/tree-based data structures. It might contain a dummy element, if unused.

struct language_function Data.

There is an internal GCC data structure of the middle-end called 'struct function' whereof one exists for each function to be processed by the middle-end. One of its members is of type 'struct language_function' and might be used to put language-specific attributes into this generic structure. Currently, the C and C++ front-ends heavily define this structure. It might also contain a dummy element.

struct lang_identifier Data.

While the above data structures only need to be defined, the structure 'lang_identifier' must have useful members, because each GENERIC/tree-based identifier node (e.g. as returned by `get_identifier`) basically IS of type 'struct lang_identifier'. The structure 'struct tree_identifier' must be the first member (remember struct inheritance), because various part of GCC expect 'struct tree_identifier' to be the first member of structure `lang_identifier` or cast 'struct lang_identifier' to 'tree' to access the attributes of identifier node.

union lang_tree_node Data.

The union `lang_tree_node` allows the front-end to expand the GENERIC `tree_node` by new tree-based data structures, e.g. in the case when a front-end extends GENERIC by additional tree codes. If no additional tree node data structures are required, the only member of 'union lang_tree_node' should be 'union tree_node' to make the language specific tree node a GENERIC tree node.

3.4 Middle-end/framework interaction

While the language hooks are the API for the middle-end/framework to call back to the front-end, there is also a set of API functions "exposed" by the middle-end/framework to be used by the front-end to accomplish specific tasks. Again, there is a whole bunch of functions, but I'll for now only concentrate on the most important ones.

layout_decl Function.

This function is usually called after a declaration, e.g. variable declaration, has been completely scanned/parsed. Among others, it then sets the size, mode (as described above) and alignment of the declaration or its members in the case of a record type (representing a struct, union or C++ class).

rest_of_decl_compilation Function.

This function might be called by a front-end to complete a declaration, after `layout_decl` has been called. Among other actions, this will then generate assembly code for this data declaration. Please note that it is not really necessary to call this function for each declaration in the front-end (the C front-end is actually doing it), because `write_global_declarations` - if the language hook is not overwritten - also generates assembly code for each declaration that has not yet been output to the assembly file.

layout_type Function.

Like `layout_decl`, but related to types, though 'typedef' in terms of C/C++ programming.

rest_of_type_compilation Function.

This function does not print out assembly code for type definitions, but debug information if debug mode (-g option) is enabled. This should be called for each type definition, if type definitions are used.

cgraph_finalize_function Function.

This function usually gets called after a function definition has been scanned/parsed. Among other actions, this will add the function definition to the cgraph (call graph). The call graph is one of the main middle-end data structures for handling functions and optimizations on functions.

cgraph_finalize_compilation_unit Function.

This function must be called at the end of the compilation unit, though once the complete input source file has been parsed. This will then process the functions of the call graph, call the gimplifier (if necessary), call the optimization pass manager to run the source code optimizations and finally generate assembly code (via the back-end).

3.5 Debugging/Troubleshooting

Once you finished your front-end code and start testing/using your front-end, you will possibly get faced with beloved internal compiler errors or even worse, incorrect code generation. This section will not purely deal with debugging the C code of GCC using

GDB, but it will show some builtin mechanisms to get debugging output of the front-, middle- and back-end. GCC provides a rich set of internal debugging options which usually start with `-fdump-*`. Most of these options are beyond the scope of a front-end developer - because they deal with the various code optimization passes performed by GCC -, but some of these dumps are quite beneficial:

-fdump-translation-unit This prints out the complete translation unit in GENERIC form. The declaration `'int a'` from above would then be represented in the debugging output by (some details are omitted):

@2694	var_decl	name: @2701	type: @3
		align: 32	used: 0
@2701	identifier_node	strg: a	lngt: 6
@3	integer_type	name: @1	
		prec: 32	sign: signed
@1	type_decl	name: @2	type: @3
@2	identifier_node	strg: int	lngt: 3

Unfortunately, this output also contains all the builtin types and functions so that it is quite oversized, at least in the case of C/C++. Like `-fdump-tree-original`, the output generated by this flag is the responsibility of the front-end..

-fdump-tree-original This flag causes the front-end to print out each function in its original form without optimizations. The C front-end does not - like the name suggests - print out the functions in tree-like notation (like above), but in pure C statements. But finally, it is up to the front-end which output form to choose for `-fdump-tree-original`.

-fdump-tree-gimple This prints out each function in GIMPLE form, though after the transformation from GENERIC. If a front-end does not use GENERIC, but directly provides GIMPLE to the middle-end, this is the first debugging file to look at.

-fdump-tree-vcg This provides a control flow graph for each function in VCG (visualizing compiler graphs) notation to be viewed by an VCG displaying tool, e.g. [14].

While most of the debugging dumps generated by GCC are part of the middle-end, the front-end however could/should produce a debugging output for `-fdump-translation-unit` and for `-fdump-tree-original`. The good news is that the functions to dump out a GENERIC node are already available. This is how a front-end could produce the dumps (very simplistic):

```
FILE * stream = dump_begin (TDI_tu, &flags);
if (stream)
{
```

```

dump_node (<tree-node>, <flags>, stream);
dump_end (TDI_tu, stream);
}

```

However, there is also an approach to review the tree-based GENERIC IR for live-debugging sessions in GDB. There is a "tool" (well, a set of functions which then get called from within GDB) called tree browser embedded into GCC. Recently, there were discussions on the mailing list about removing the tree browser from the GCC source code⁸ and implement an equivalent tool using the Python extension in GDB. Be prepared that this tool might get removed from the GCC mainline. For a detailed reference, please check out [13].

4 Makefile fragment

As described above, each language front-end has its own makefile fragment, named Make-lang.in, which gets included into the generated makefile in build-x.y.z/gcc. As an entry point to the GCC makefile hierarchy, lets consider which targets are called when building GCC and specifically a GCC front-end. Assuming a bootstrap build with the gcalc front-end as an example, whenever you do a make (which is basically make all) or a make bootstrap, the following targets get called⁹. The targets which must be available in the language makefile are marked bold.

bootstrap ⇒ stage3-bubble ⇒ all-stage3 ⇒ all-stage3-gcc

The *all-stage3-gcc* target changes into the build-x.y.z/gcc directory and finally calls 'make all':

```

all
⇒ all.internal ⇒ native ⇒ gcalc
⇒ start.encap ⇒ lang.start.encap ⇒ gcalc.start.encap
⇒ rest.encap ⇒ lang.rest.encap ⇒ gcalc.rest.encap

```

These three language targets are the main targets for building the compilation driver and the compiler. Other targets like for building the documentation are not discussed here, while the installation target will be discussed below. The following rules apply to these three targets:

gcalc (or 'lang') This target is usually used to build the core compiler, e.g. cc1plus.

⁸Due to buggy code, apparently. I haven't yet faced any problems, but I don't use the tree browser that frequently.

⁹This view on the Makefile hierarchy is not complete, since the stage1 and stage2 targets are not displayed, but stage1 and stage2 are built as prerequisites to stage3.

gcalc.start.encap (or 'lang'.start.encap) This target allows to include all those parts which do not rely on a working gcc-driver version. Working gcc-driver version in this context just means a gcc-driver created by this build, because the gcc-driver (usually called xgcc before the installation) is also built by this target. This target is usually used to build the compilation drivers, e.g. like g++. Please note, all the build instructions here use the gcc driver/compiler from the previous stage or the host compiler.

gcalc.rest.encap (or 'lang'.rest.encap) This target finally allows to include all those parts which rely on a working gcc-driver version, so if your front-end requires any parts to be built by the newly created gcc, put these instructions here. I checked several GCC front-ends and none of these use this target.

Next, I will go through a sample make command to explain and show how to include dependent libraries or how to get the GCC backend integrated into your compiler:

```
calc1$(exeext): <object_files> $(BACKEND) $(LIBSDEPS) attribs.o
                $(CC) $(ALL_CFLAGS) $(LDFLAGS) -o $$@ <object_files> $(LIBS) \
                $(BACKEND) attribs.o $(GMP LIBS) $(BACKEND LIBS)
```

So, the GCC make infrastructure provides a lot of variables to simplify the dependency notation and the build commands. Because there are so many variables defined by the infrastructure, I will not list them here, except the variable BACKEND. The BACKEND variable lists all the object files provided by the GCC framework to connect your front-end to the middle-end and back-end of GCC. For a reference of the mainly used variables, please check the makefiles of the other GCC front-ends. To find out which variable has which specific value, I would recommend to grep the makefile in the build-x.y.z/gcc directory.

4.1 Front-end Installation

For the installation of the front-end executables, the language front-end needs to define a separate installation target, named 'lang'.install-common:

```
EXES = gcalc

gcalc.install.common: installdirs
for name in $(EXES); \
do \
if [ -f $$name ] ; then \
name2="'echo `basename $$name` | sed -e '$(program_transform_name)'" ; \
rm -f $(DESTDIR)$(bindir)/$$name2$(exeext); \
$(INSTALL_PROGRAM) $$name$(exeext) $(DESTDIR)$(bindir)/$$name2$(exeext); \
chmod a+x $(DESTDIR)$(bindir)/$$name2$(exeext); \
fi ; \
done
```

When looking through the above makefile extract, you will notice that the installation target only installs the compilation driver and not the compiler. The compiler gets automatically installed by the GCC makefile. If you are interested in the details, the compiler gets installed by the target `install-common` of the `gcc` makefile in `build-x.y.z/gcc`. While the compilation driver is installed in the `prefix/bin` directory, the compiler is put into the `{prefix}/libexec/gcc/{target_noncanonical}/{version}` directory. As a note, `{prefix}` is the directory specified for the `-prefix` option of the configure script, `{target_noncanonical}` is a string like `x86_64-unknown-linux-gnu` (YMMV) and the `{version}` usually has a form of `x.y.z`.

References

- [1] *GCC back-end architecture listing*. Informational web page. <http://gcc.gnu.org/backends.html>.
- [2] *GCC build and installation instructions*. Informational web page. <http://gcc.gnu.org/install/>.
- [3] *GCC download mirrors*. <http://gcc.gnu.org/mirrors.html>.
- [4] *GCC front-end language listing*. Informational web page. <http://gcc.gnu.org/frontends.html>.
- [5] *GCC gcalc front-end source code*. <http://gcc.gnu.org/wiki/WritingANewFrontEnd?action=AttachFile&do=get&target=gcc-4.6.0.gcalc.tar.gz>.
- [6] *GCC internal documentation about collect2*. Documentation. <http://gcc.gnu.org/onlinedocs/gccint/Collect2.html\#Collect2>.
- [7] *GCC internal documentation about config-lang.in*. Documentation. <http://gcc.gnu.org/onlinedocs/gccint/Front-End-Config.html\#Front-End-Config>.
- [8] *GCC internal documentation about GTY(()) expressions*. Documentation. <http://gcc.gnu.org/onlinedocs/gccint/GTY-Options.html\#GTY-Options>.
- [9] *GCC internal documentation about machine modes*. Documentation. <http://gcc.gnu.org/onlinedocs/gccint/Machine-Modes.html\#Machine-Modes>.

- [10] *GCC internal documentation about the format of lang.opt file.* Documentation. <http://gcc.gnu.org/onlinedocs/gccint/Option-file-format.html\#Option-file-format>.
- [11] *GRAPHITE information page in GCC wiki.* Informational web page. <http://gcc.gnu.org/wiki/Graphite>.
- [12] *ModularGCC project page in GCC wiki.* Informational web page. <http://gcc.gnu.org/wiki/ModularGCC>.
- [13] *Tree browser discussion on gcc mailing list.* Mailing list archive. <http://gcc.gnu.org/ml/gcc/2010-05/msg00070.html>.
- [14] *vcgviewer.* Project web page. <http://code.google.com/p/vcgviewer/>.
- [15] Jason Merrill. *GENERIC and GIMPLE: A New Tree Representation for Entire Functions.* 2003. Download document from <ftp://gcc.gnu.org/pub/gcc/summit/2003/GENERIC%20and%20GIMPLE.pdf>.
- [16] Ian Lance Taylor. *The Go frontend for GCC.* 2010. Download document from <http://gcc.gnu.org/wiki/summit2010>.