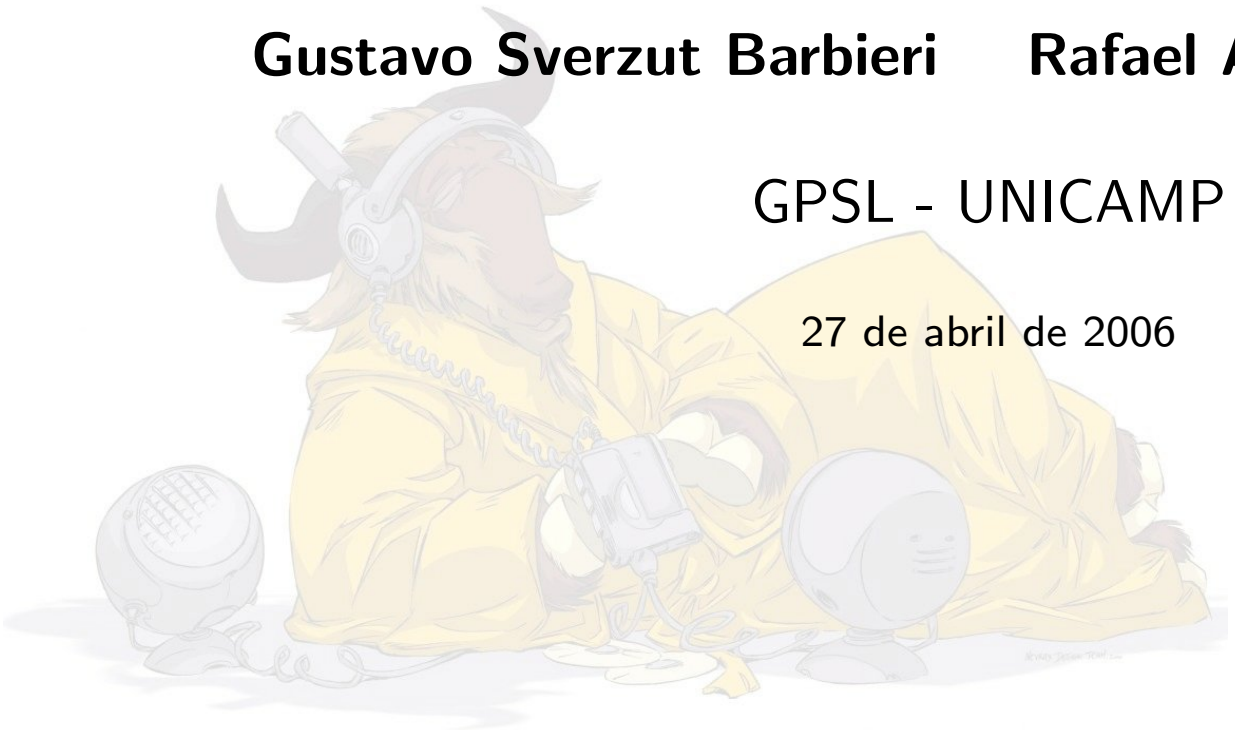
The Hello World GCC Front End

Gustavo Sverzut Barbieri **Rafael Ávila de Espíndola**

GPSL - UNICAMP

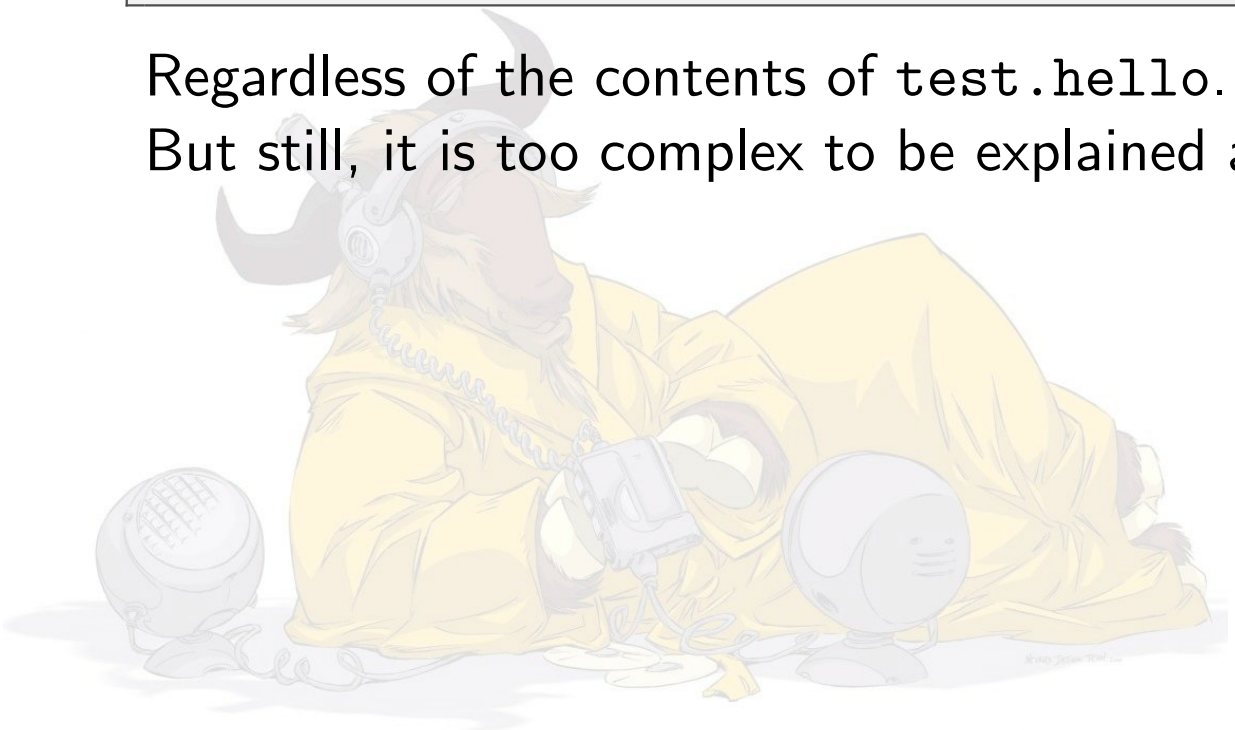
27 de abril de 2006



The Hello World front end is the smallest GCC front end:

```
$ ./ghello test.hello -o hello
$ ./hello
HelloWorld
```

Regardless of the contents of `test.hello`.
But still, it is too complex to be explained all at once.

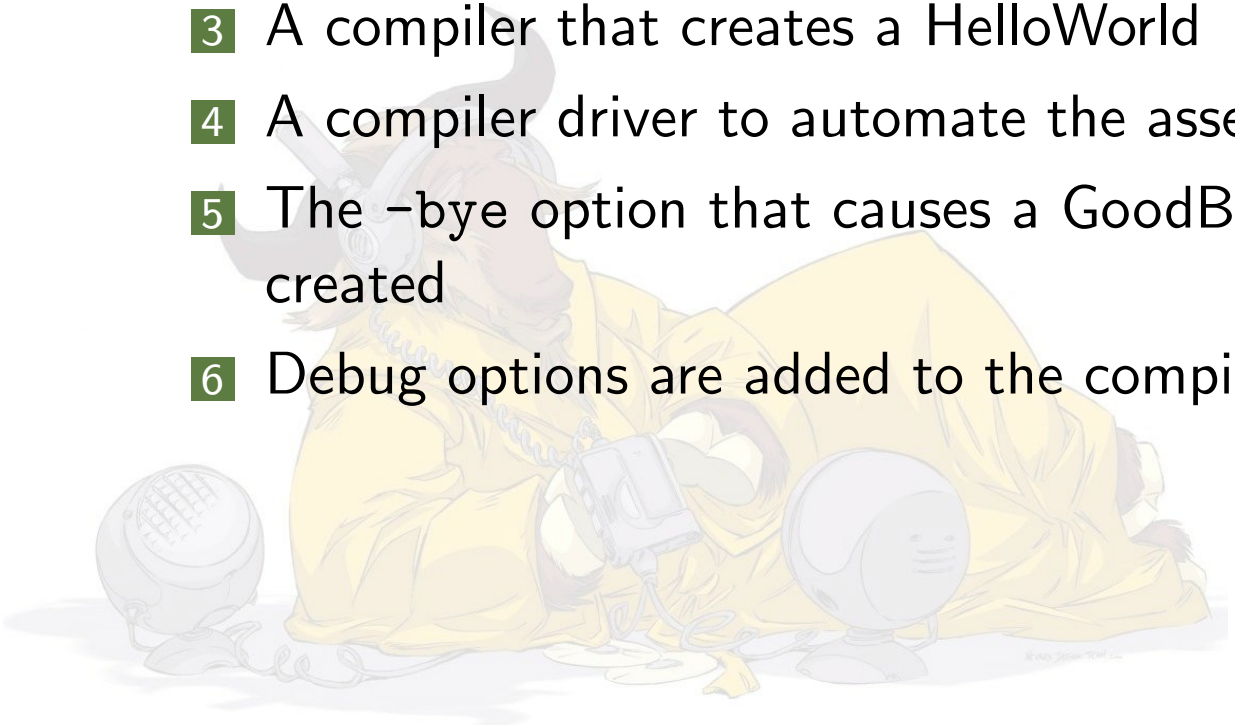


Steps

Setup

The front end will be constructed incrementally. At each step a new functionality will be added:

1. A dummy program that links with the GCC middle and back end.
2. A compiler that creates an empty `main()`
3. A compiler that creates a HelloWorld
4. A compiler driver to automate the assembling and linking
5. The `-bye` option that causes a GoodBye program to be created
6. Debug options are added to the compiler



Installing a Snapshot

Setup

- A snapshot of the necessary parts of the GCC source, the sources used in the tutorial *and* and the resulting binaries are available in `http://tux05.ltc.ic.unicamp.br/~rafael/snapshot.torrent`
- The GCC `configure` creates a `Makefile` with absolute path names :-(
- You *must* uncompress the snapshot into `/tmp`

```
$ cd /tmp
$ btdownloadcurses http://tux05.ltc.ic.unicamp.br/~rafael/snapshot.torrent
$ tar xjf snapshot-gcc.tar.bz2
$ cd /tmp/snapshot/
```

- Compiling GCC takes a *long* time. We don't have that time now.
- Incremental compiles will be much faster
- You should follow this presentation using full source code and **The gcc hello world front end HOWTO**, located at `http://tux05.ltc.ic.unicamp.br/~rafael/gcc.pdf`

Directory Tree

Setup

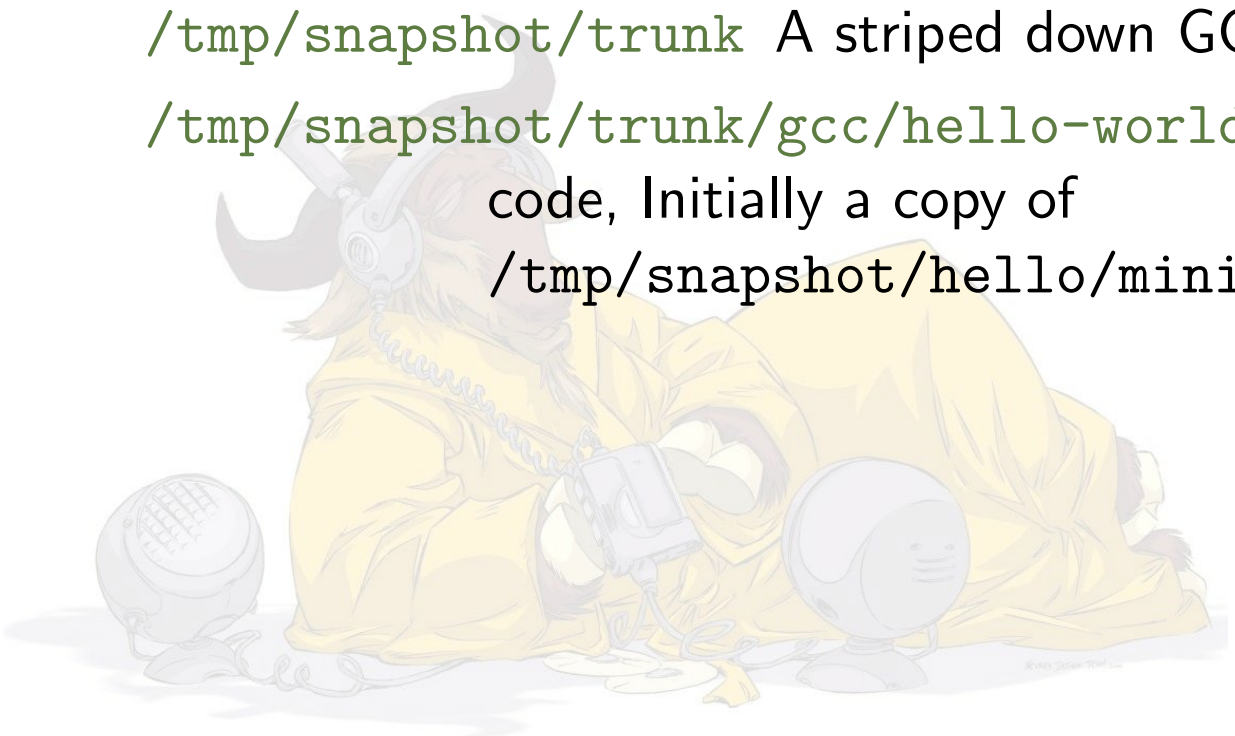
After extracting, you will have the following directories:

`/tmp/snapshot/hello-world` The sources of each step of this tutorial

`/tmp/snapshot/build` The directory used to build GCC

`/tmp/snapshot/trunk` A striped down GCC source tree

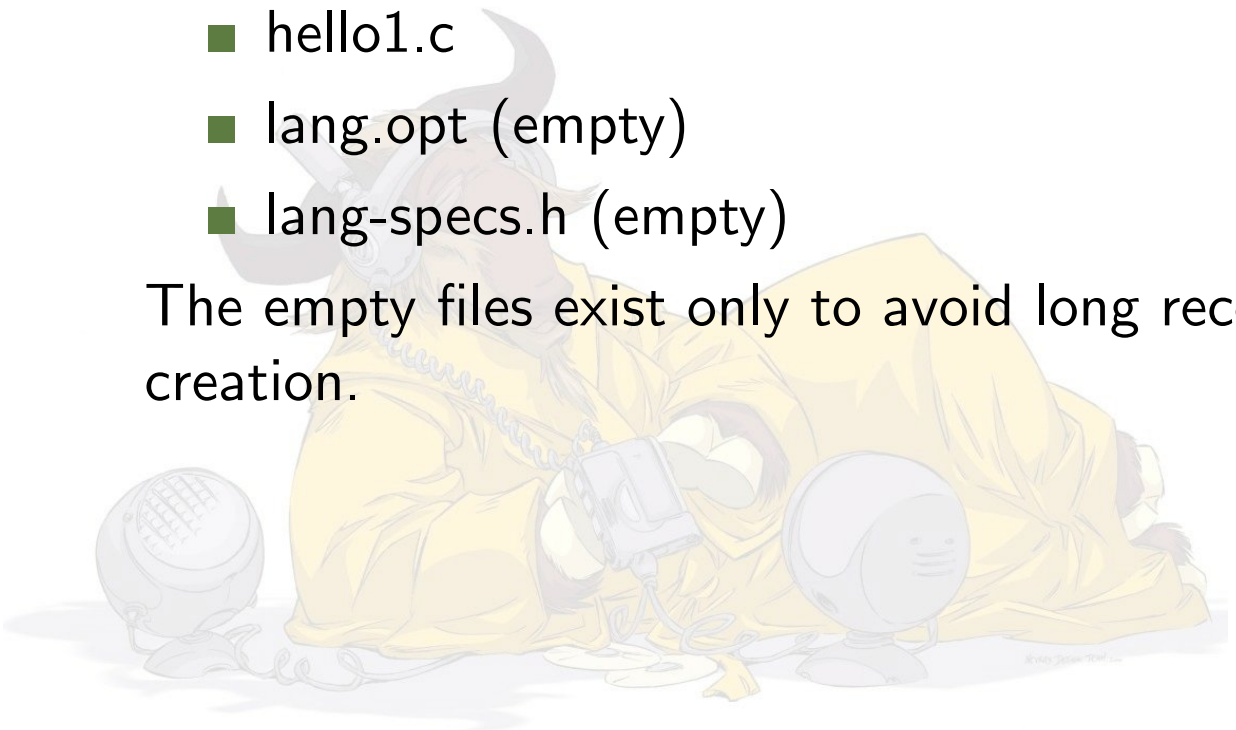
`/tmp/snapshot/trunk/gcc/hello-world` Current front-end code, Initially a copy of `/tmp/snapshot/hello/minimal`



Each front end lives in a subdirectory of `gcc`. The hello world front end is in `/tmp/snapshot/trunk/gcc/hello-world`. In it you will find

- config-lang.in
- Make-lang.in
- hello1.c
- lang.opt (empty)
- lang-specs.h (empty)

The empty files exist only to avoid long recompiles after their creation.

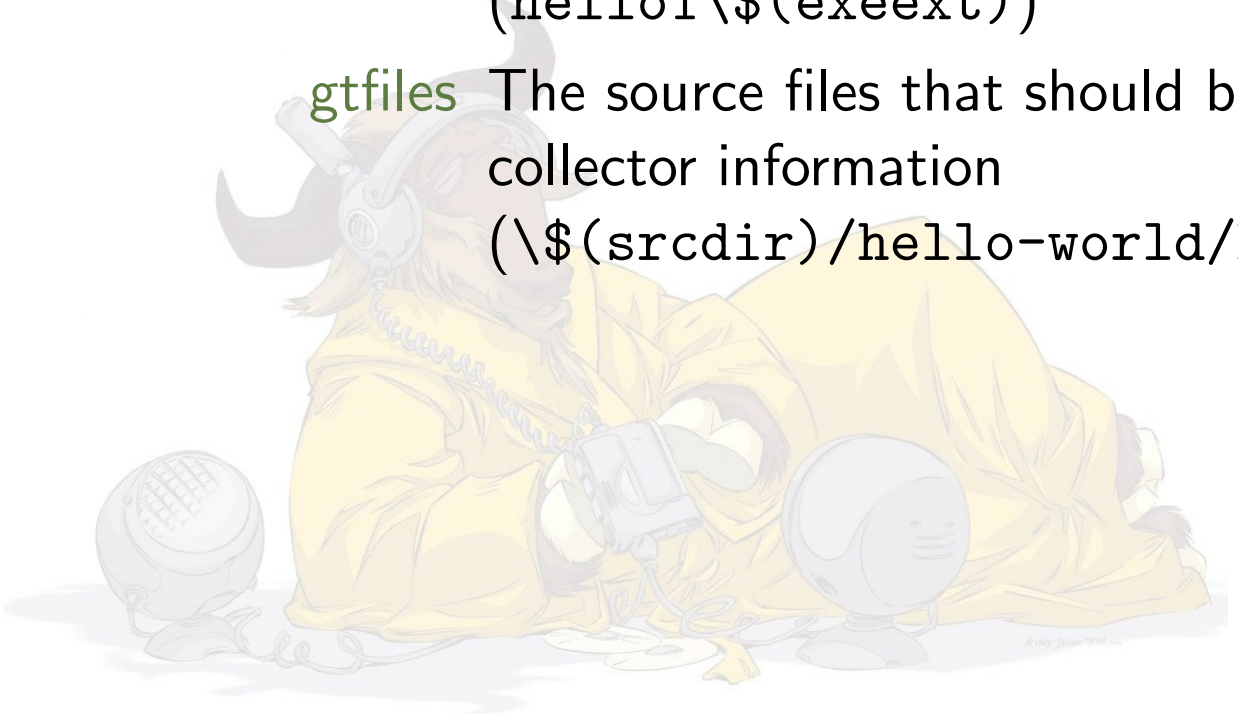


This file is a shell script that should define the following variables:

language — The language name. Will be the name of the main target of `Make-lang.in` (hello-world)

compilers — A list of compilers that will be created (`hello1\$(exeext)`)

gtfiles — The source files that should be scanned for garbage collector information (`\$(srcdir)/hello-world/hello1.c`)



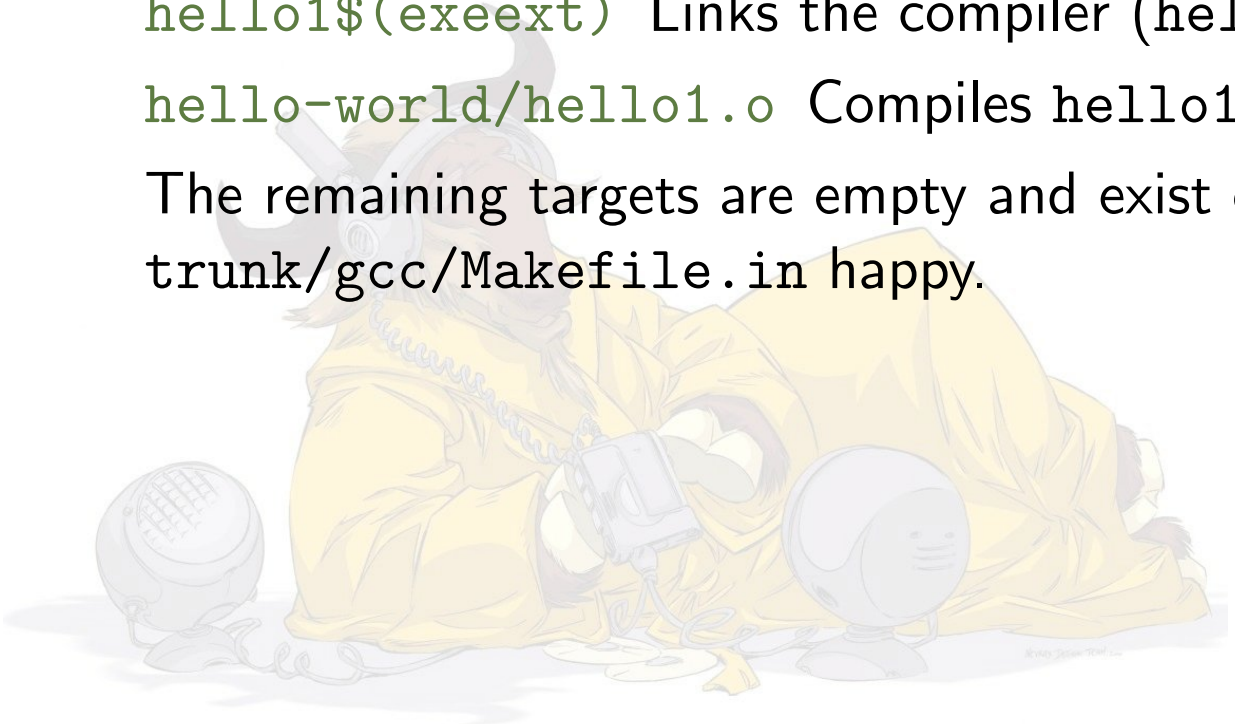
This file is *included* in `trunk/gcc/Makefile.in`. Because of this, all paths are relative to `trunk/gcc`. In this file you will find three useful targets:

`hello-world` Main entry point

`hello1$(exeext)` Links the compiler (`hello1`)

`hello-world/hello1.o` Compiles `hello1.c` into `hello1.o`

The remaining targets are empty and exist only to make `trunk/gcc/Makefile.in` happy.



This is the only source file. In it you will find:

- 5 empty data types definitions. They make the garbage collector happy.
- Many empty functions (`insert_block()`, ..., `hello_type_for_mode()`). They will be callbacks.
- The initialization of the `lang_hooks` variable. It contains pointers to the callbacks.
- `tree.def` is included three times with a bit of macro magic. This initializes some data structures that implement GCC's intermediate representation.
- The garbage collector headers are included. These headers are automatically generated.

The snapshot includes an initial build. Following this steps will start a build from scratch. **Don't do it now!**

1. `configure` must be run from a build dir that is distinct from the source dir
2. to build the Hello World front end, add `--enable-languages=hello-world`.

```
$ cd /tmp/snapshot/build/
$ ../trunk/configure --enable-languages=hello-world
$ make
```

1. `touch /tmp/snapshot/trunk/gcc/hello-world/hello1.c`
2. `cd /tmp/snapshot/build/gcc`
3. `make hello1`
4. Very little will be rebuilt

```
$ ./hello1
Execution times (seconds)
 TOTAL                 :    0.01               0.00               0.02
12 kB
Extra diagnostic checks enabled; compiler may run slowly.
Configure with --disable-checking to disable checks.
$
```

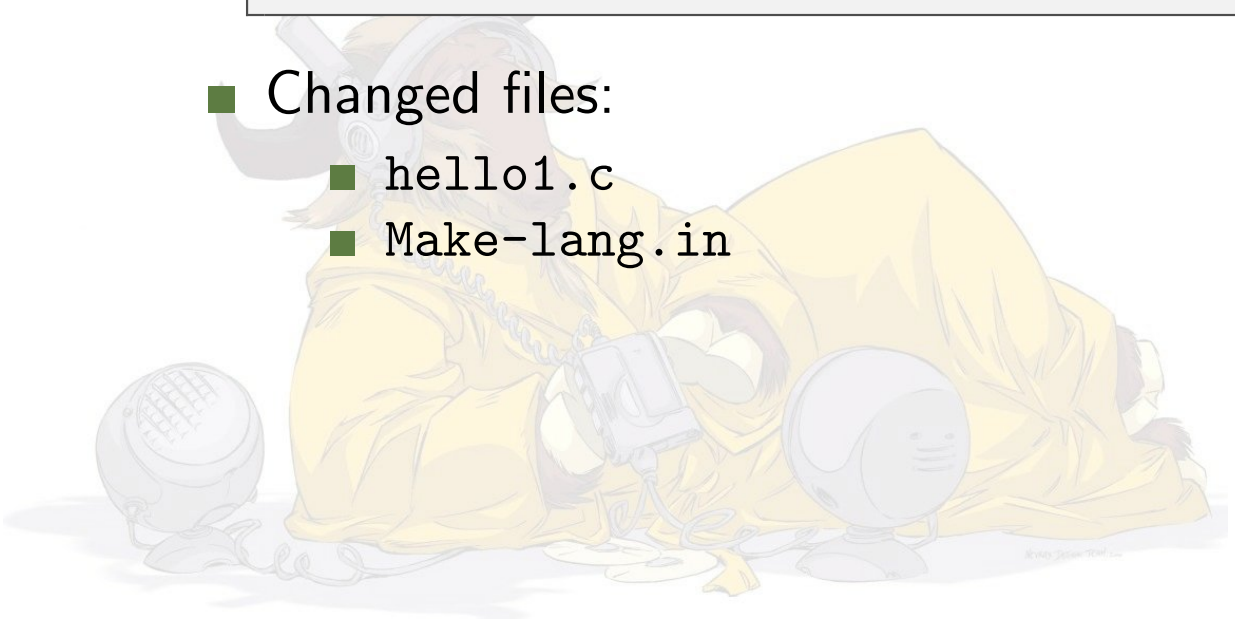
This step

An Empty Main

- Compiler will always produce an assembly with an empty main function
- Commands:

```
$ cd /tmp/snapshot/build/gcc
$ cp /tmp/snapshot/hello-world/1-main/* \
     /tmp/snapshot/trunk/gcc/hello-world
$ make hello1
```

- Changed files:
  - `hello1.c`
  - `Make-lang.in`



Basic API

An Empty Main

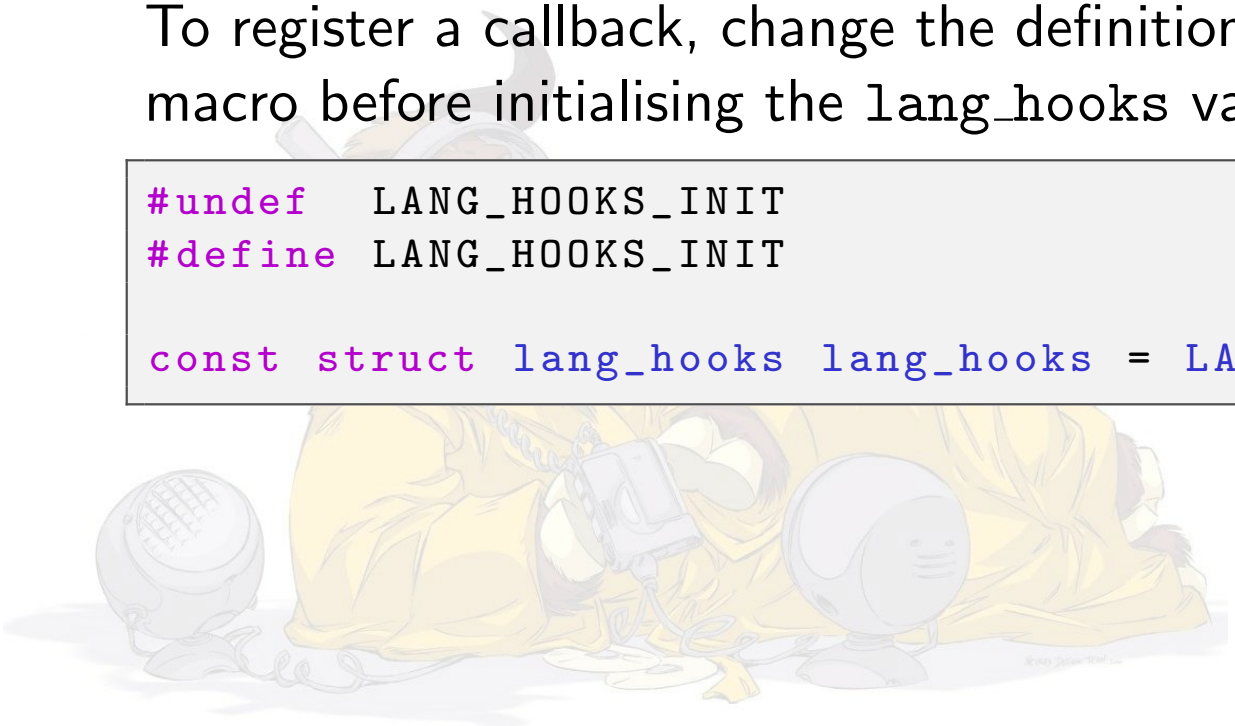
GCC provides the `main()` function. The front end must implement some callbacks:

- `hello_expand_function`
- `hello_init`
- `hello_parse_file`

To register a callback, change the definition of the corresponding macro before initialising the `lang_hooks` variable:

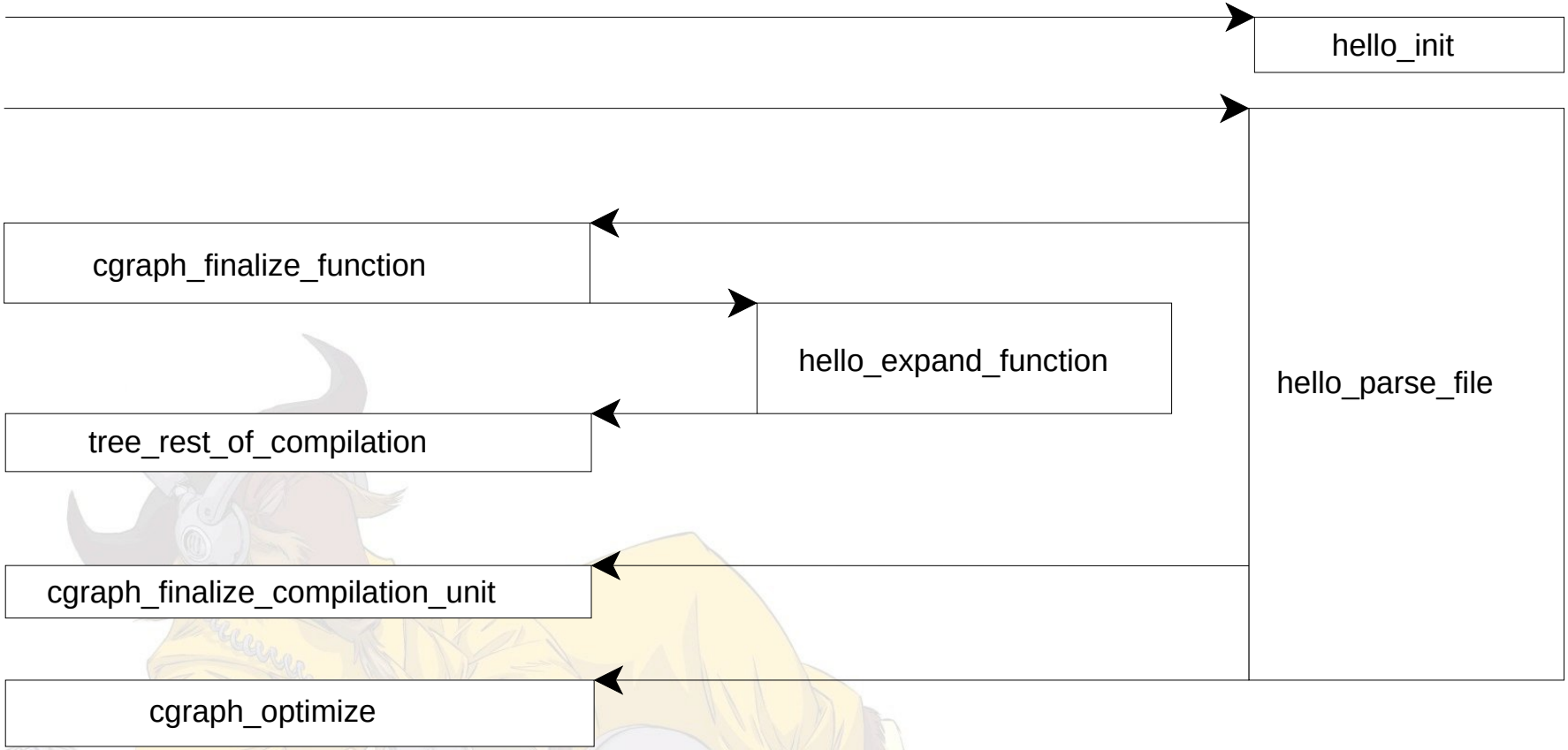
```
#undef  LANG_HOOKS_INIT
#define LANG_HOOKS_INIT                    hello_init

const struct lang_hooks lang_hooks = LANG_HOOKS_INITIALIZER;
```



Call Graph

An Empty Main



Intermediate Representation

An Empty Main

GCC uses three different intermediate representations:

GENERIC A high level representation based on trees.

GIMPLE Uses GENERIC's data structures, but is in static single assignment form (SSA).

RTL Low level representation used by the target specific part of the compiler (backend).

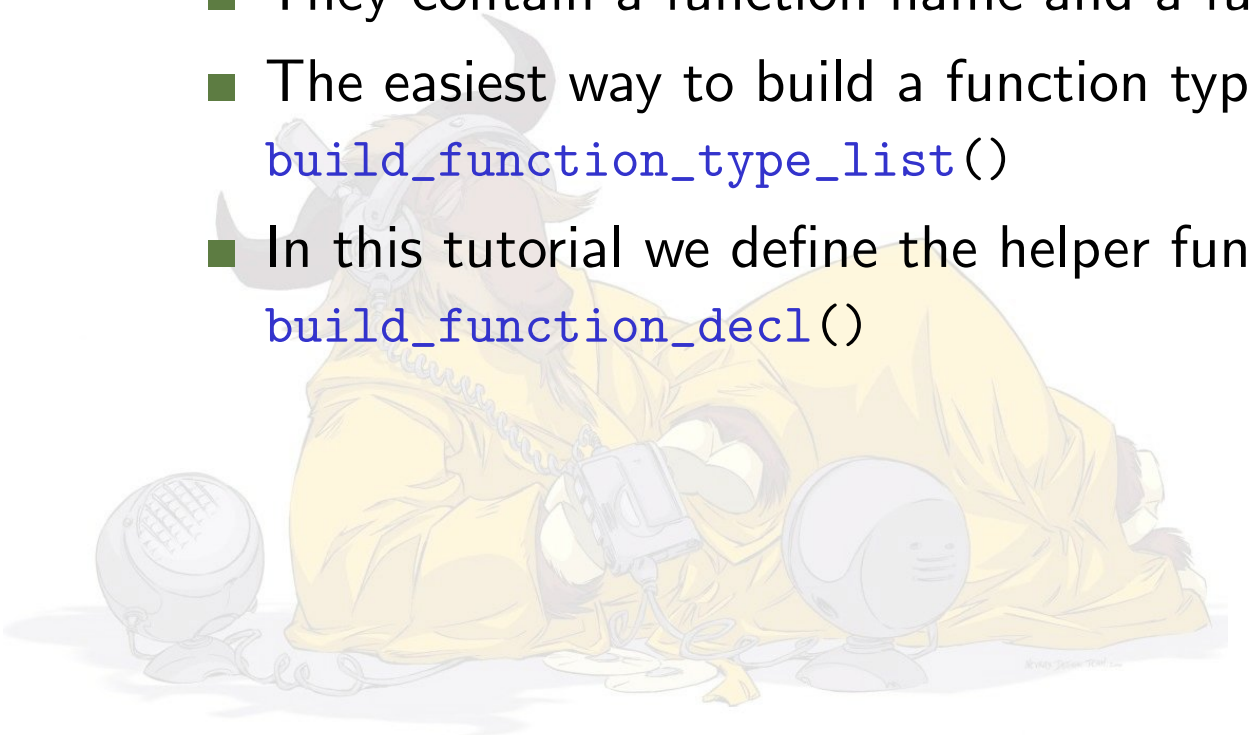
- The front end uses GENERIC to transfer one function a time to the middle end.
- The middle end uses GIMPLE to optimize
- The backend uses RTL to generate assembly code

Function Declaration

An Empty Main

Some facts about function declarations

- In GENERIC, every use a function is represented with a function declaration
- Function declarations are build with `build_fn_decl()`
- They contain a function name and a function type
- The easiest way to build a function type is with `build_function_type_list()`
- In this tutorial we define the helper function `build_function_decl()`



- The function body is also stored into the declaration:
  - The body itself in `DECL_SAVED_TREE`
  - The return in `DECL_RESULT`
- In this tutorial we defined the helper function `build_function()`

Changes to `hello1.c`

An Empty Main

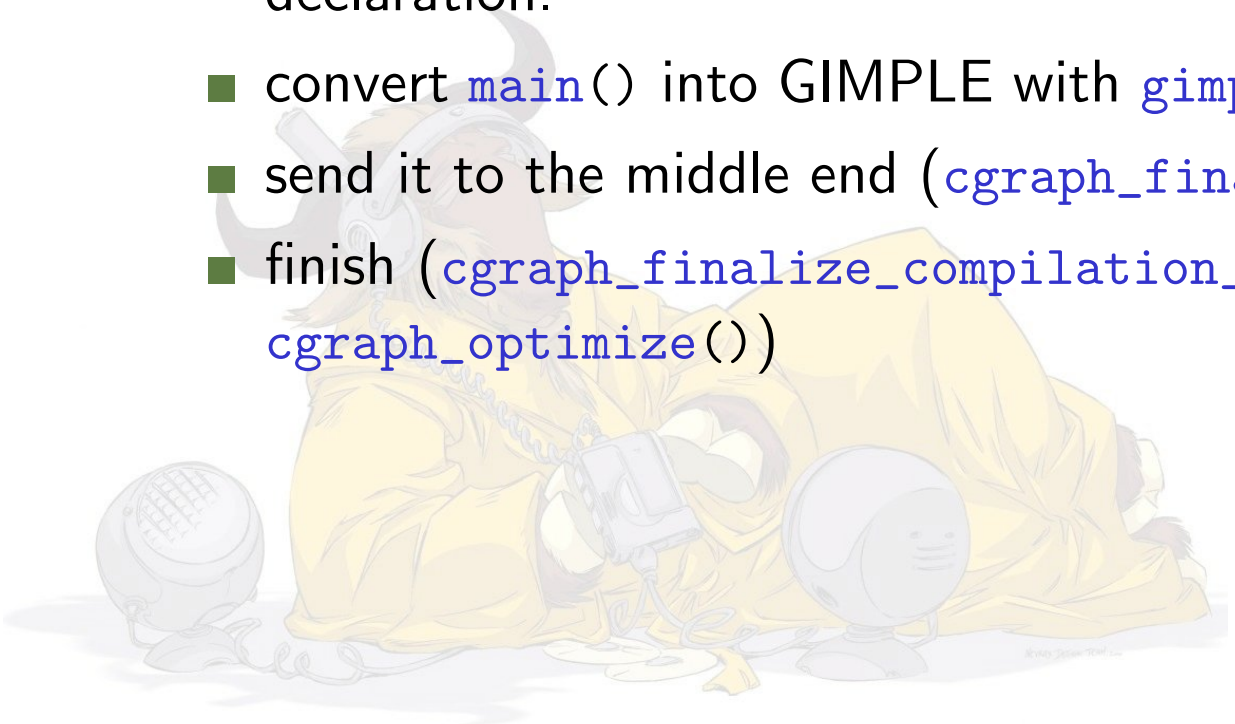
- include `tree-gimple.h` (defines `alloc_stmt_list()`)
- Make `getdecls()` return `NULL_TREE`
- `hello_init()`
  - `build_common_tree_nodes()` (`char` is signed?, size is signed?)
  - `build_common_tree_nodes2()` (`double` == `float`)
- `hello_expand_function()`
  - call `tree_rest_of_compilation()`
- `hello_parse_file()` This function will pretend that it has parsed the program

```
int main(void) {}
```

hello_parse_file

An Empty Main

- call `build_function_type_list()` to construct `main()`'s type
- call `build_function_decl()` to declare main
- build an empty block (the `{}`) and statement list
- use `build_function()` to add the function body to the declaration.
- convert `main()` into GIMPLE with `gimplify_function_tree()`
- send it to the middle end (`cgraph_finalize_function()`)
- finish (`cgraph_finalize_compilation_unit()` and `cgraph_optimize()`)



Changes to `Make-lang.in`

An Empty Main

- Have target `hello1.o` dependent on `$(TREE_GIMPLE_H)`

Running

An Empty Main

```
$ touch test.hello
$ ./hello1 test.hello -o test.s
$ gcc test.s -o test
$ ./test
```

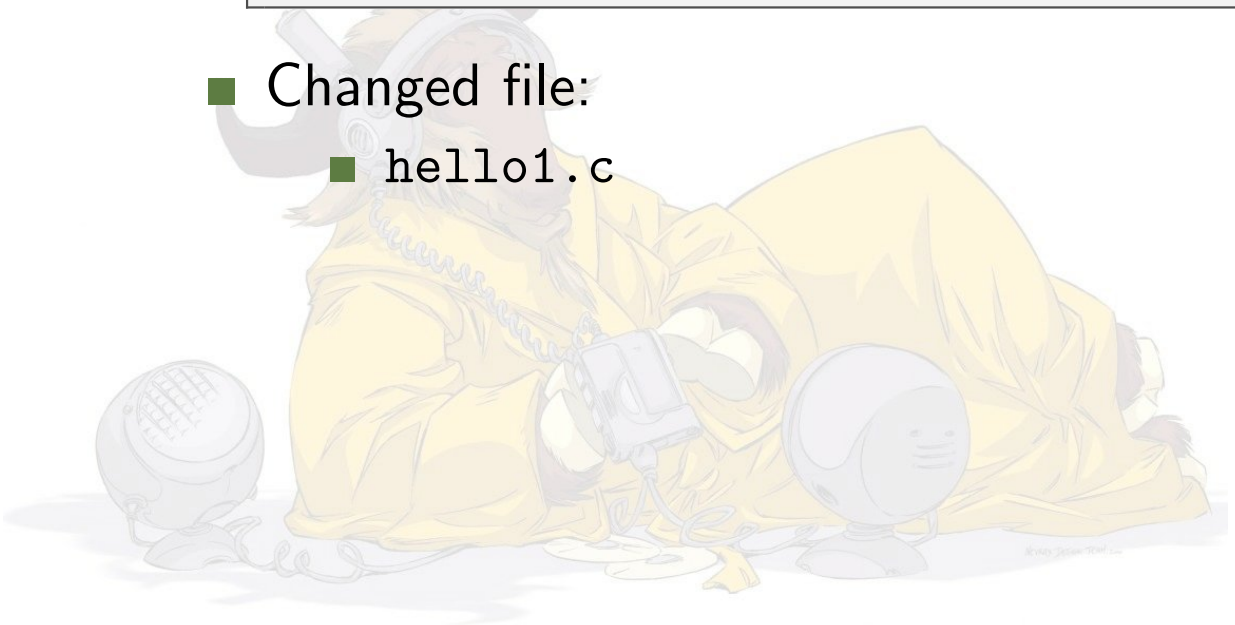
This step

Hello World

- Compiler will produce a hello world program.
- Commands:

```
$ cd /tmp/snapshot/build/gcc
$ cp /tmp/snapshot/hello-world/2-hello/* \
     /tmp/snapshot/trunk/gcc/hello-world
$ make hello1
```

- Changed file:
  - `hello1.c`

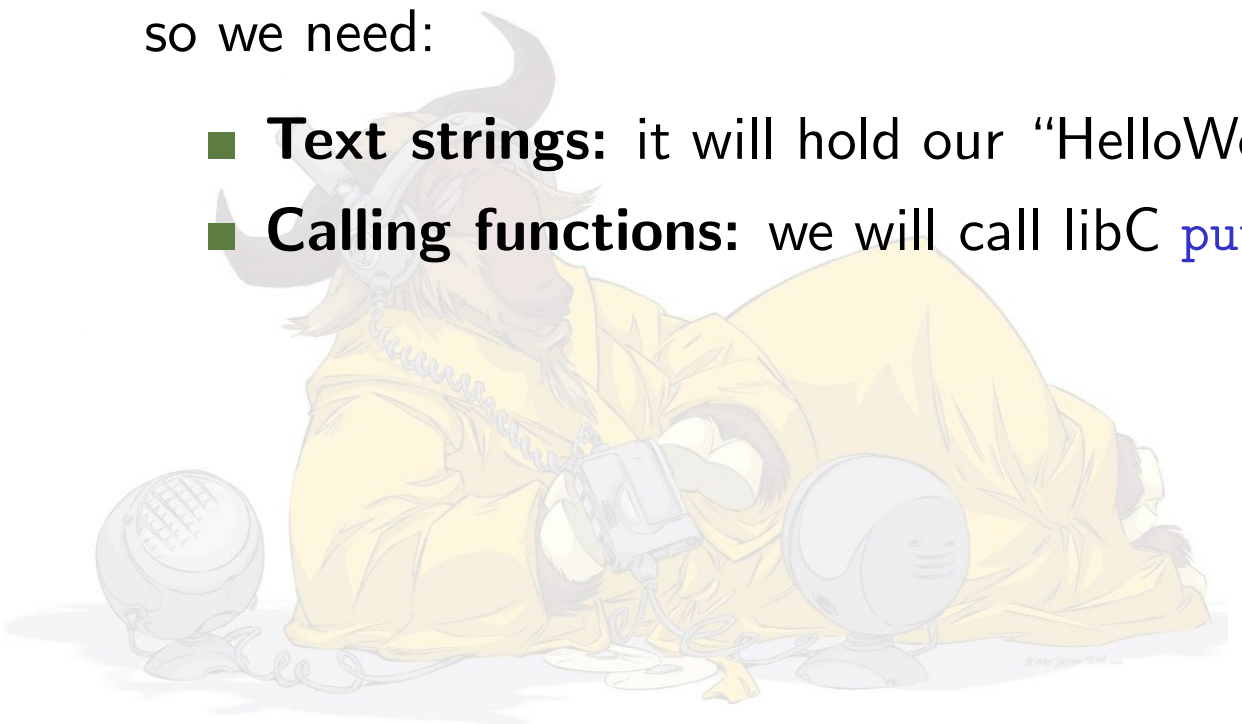


Our program will be the equivalent of:

```
int main() {
  puts ("HelloWorld");
}
```

so we need:

- **Text strings:** it will hold our “HelloWorld”.
- **Calling functions:** we will call libC `puts()`



Building Strings

Hello World

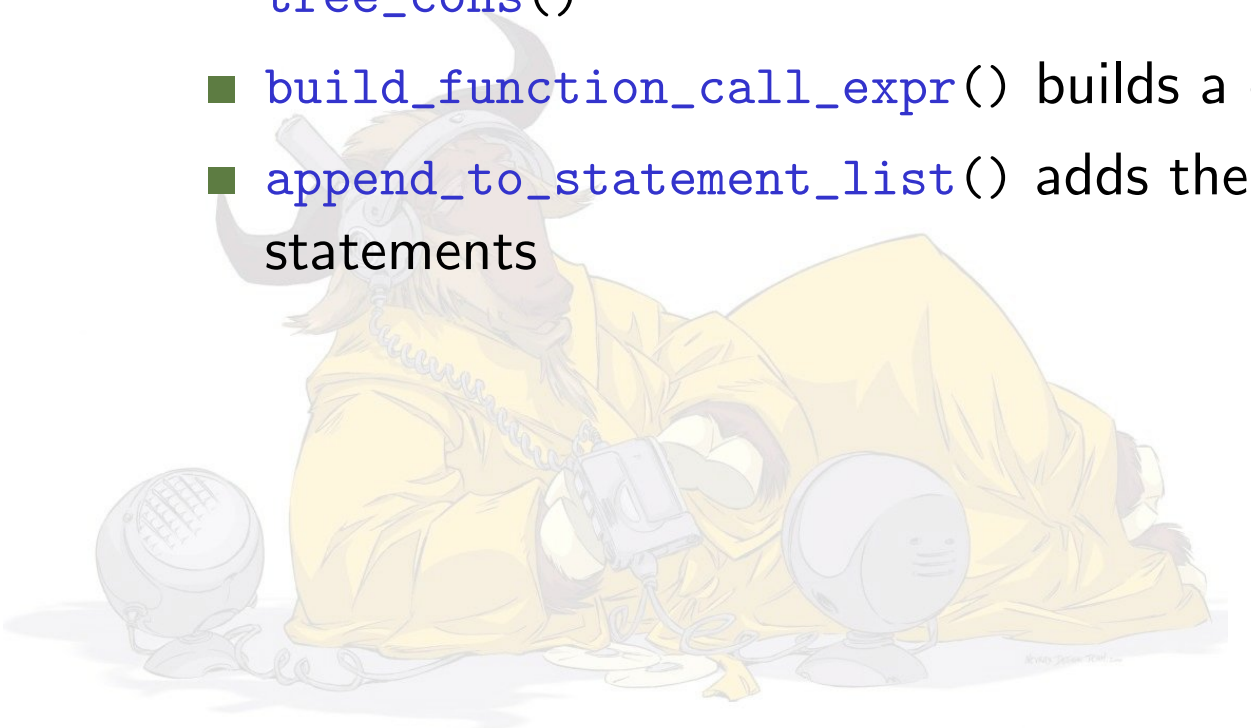
To compile a program that prints *Hello World*, we must first build a string constant:

- We will use a null (`'\0'`) terminated string to be able to use libC `puts()`
- GCC provides `build_string()`
- Some front ends index from 0, others from 1. We must set the type of the string constant!
- To build an array type, pass the element type and an index type to `build_array_type()`
- `build_index_type()` is used to build an index type from 0 to its argument.
- In this front end, `build_string_literal()` builds the string, sets the type, and returns a pointer to it.

Calling Functions

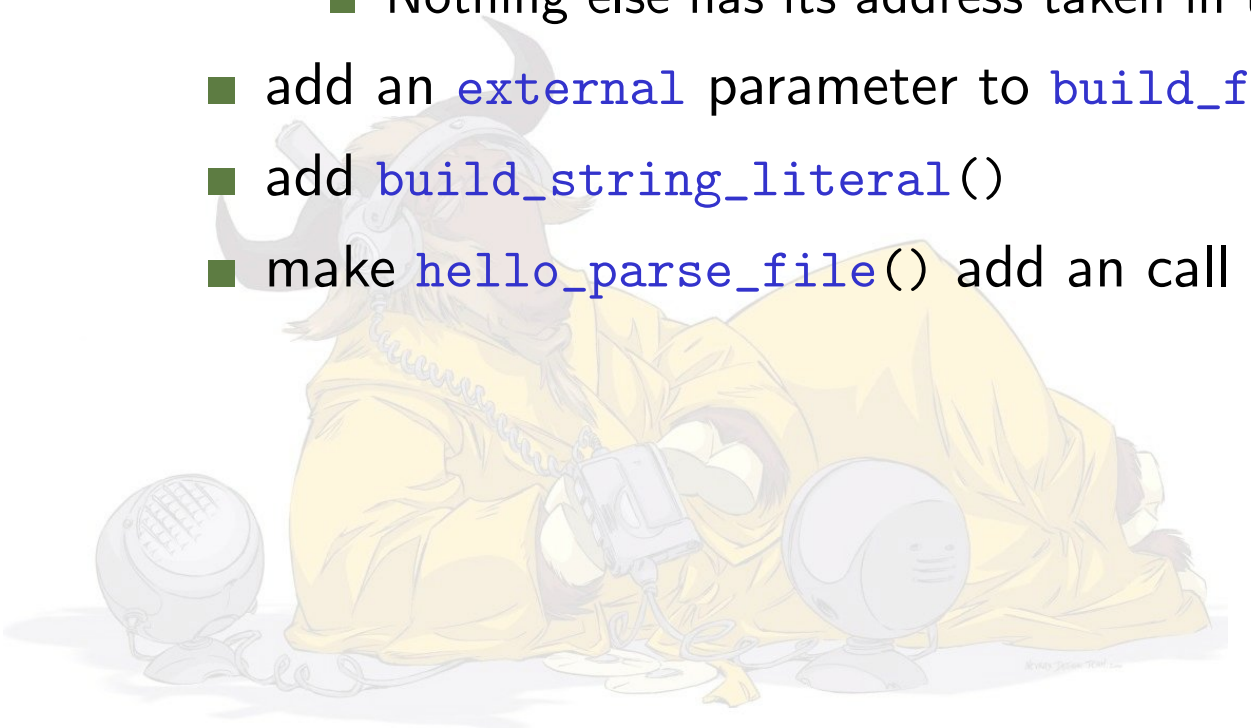
Hello World

- We will call `puts()` to print `"Hello World"`
- First, build a function prototype (analogous to `main()`)
- The arguments are represented with a list built with `tree_cons()`
- `build_function_call_expr()` builds a call statement
- `append_to_statement_list()` adds the call to `main()`'s statements



Changes to `trunk/gcc/hello-world/hello1.c`

- GCC calls `hello_mark_addressable()` to inform that something had its address taken
  - There is nothing to be done about strings
  - For function declarations, set `TREE_ADDRESSABLE`
  - Nothing else has its address taken in this front end
- add an `external` parameter to `build_function_decl()`
- add `build_string_literal()`
- make `hello_parse_file()` add an call to `puts()`



```
$ touch test.hello
$ ./hello1 test.hello -o test.s
$ gcc test.s -o test
$ ./test
HelloWorld
```

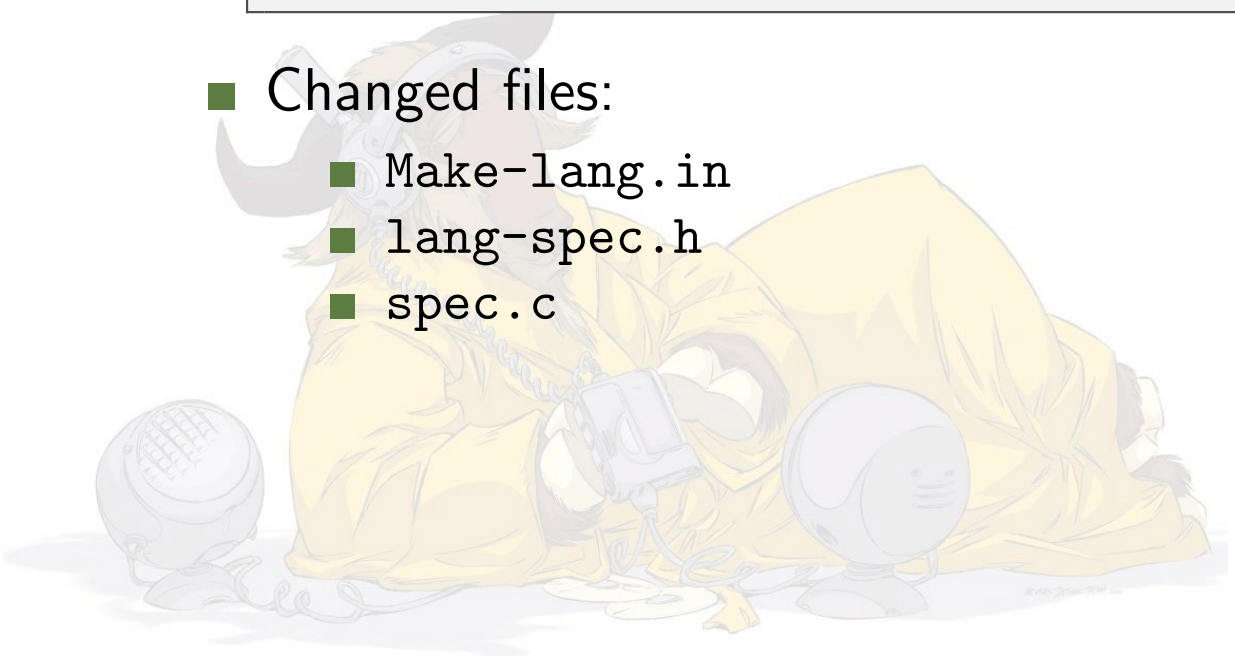
This step

Compiler Driver

- Compiler, assembler and linker will be driven by `ghello`
- Commands:

```
$ cd /tmp/snapshot/build/gcc
$ cp /tmp/snapshot/hello-world/3-driver/* \
     /tmp/snapshot/trunk/gcc/hello-world
$ make hello1
$ make ghello
```

- Changed files:
  - `Make-lang.in`
  - `lang-spec.h`
  - `spec.c`



Creating trunk/gcc/hello-world/lang-spec.h

Compiler Driver

We will now create the compiler driver. The file `lang-spec.h` contains two table entries.

```
{".hello", "@hello", NULL, 0, 0}
```

A file ending with `.hello` should be handled according to the `@hello` entry

```
{"@hello",
   "hello1 %i %(cc1_options) "
   "%(invoke_as)", NULL, 0, 0
}
```

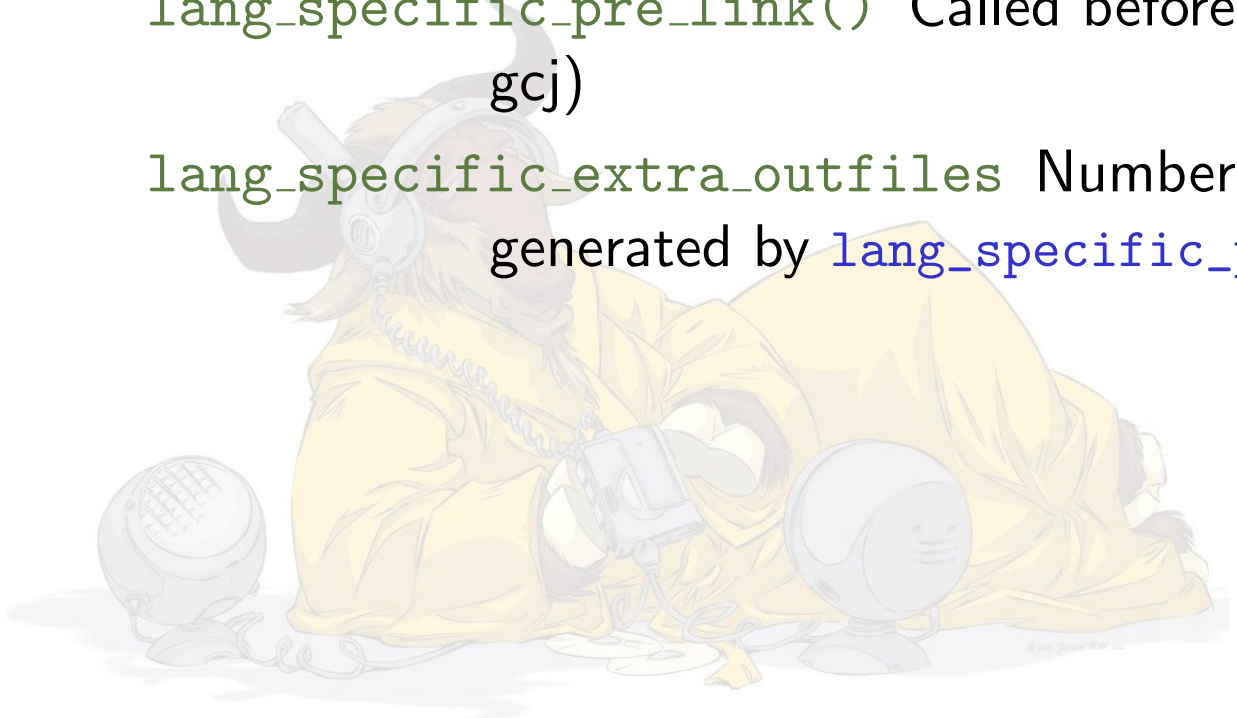
- call `hello1` with the input file name (`%i`) and common options (`%(cc1_option)`)
- call the assembler (`%(invoke_as)`)

This file contains the language specific parts of the driver. It must define:

`lang_specific_driver()` "main" function. May process, add, or remove arguments to the compiler.

`lang_specific_pre_link()` Called before linking (only used by gcj)

`lang_specific_extra_outfiles` Number of extra output files generated by `lang_specific_pre_link()`



Changes to `trunk/gcc/hello-world/Make-lang.in`

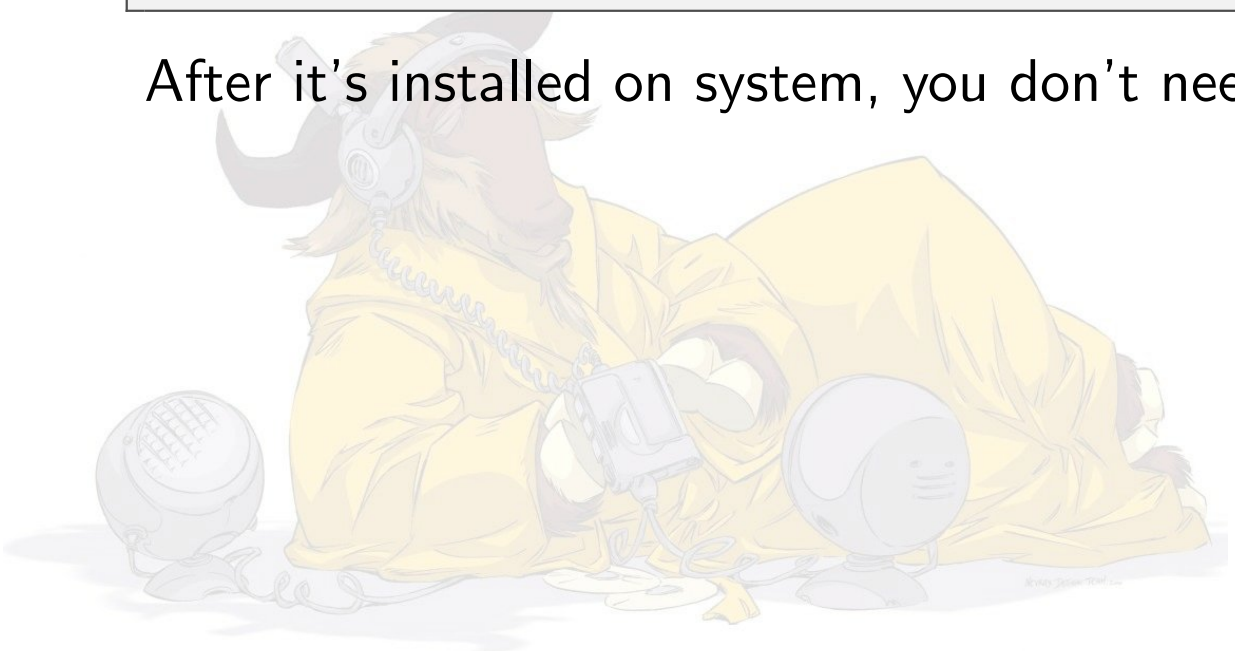
Compiler Driver

- Build the driver with the `ghello$(exeext)` target
- Change the `hello-world` target to depend on it
- Change the `hello-world.install-common` target to install ghello

The -B options informs ghello to search for hello1 in the current directory

```
$ ./ghello -B. test.hello -o test
$ ./test
HelloWorld
```

After it's installed on system, you don't need -B anymore!



This step

Adding an Option

- Compiler will accept the option -bye to print “GoodBye”.
- Commands:

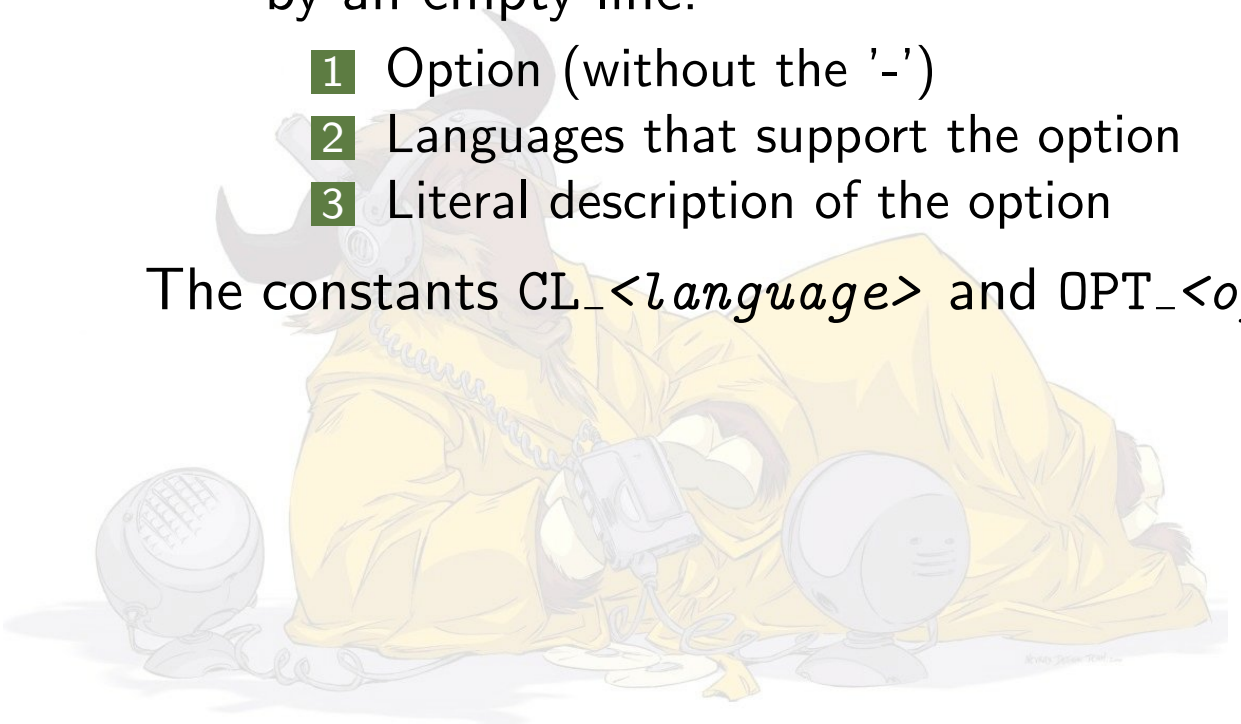
```
$ cd /tmp/snapshot/build/gcc
$ cp /tmp/snapshot/hello-world/4-option/* \
     /tmp/snapshot/trunk/gcc/hello-world
$ make hello1
$ make ghello
```

- Compiling `hello1` will take longer then usual
- Changed files:
  - `hello1.c`
  - `lang-spec.h`
  - `lang.opt`

GCC does the options parsing, but we need to declare options in `lang.opt` file:

1. Language declaration
2. Option declarations. Each option is made of 3 lines followed by an empty line:
   1. Option (without the '-')
   2. Languages that support the option
   3. Literal description of the option

The constants `CL_<language>` and `OPT_<option>` will be defined.



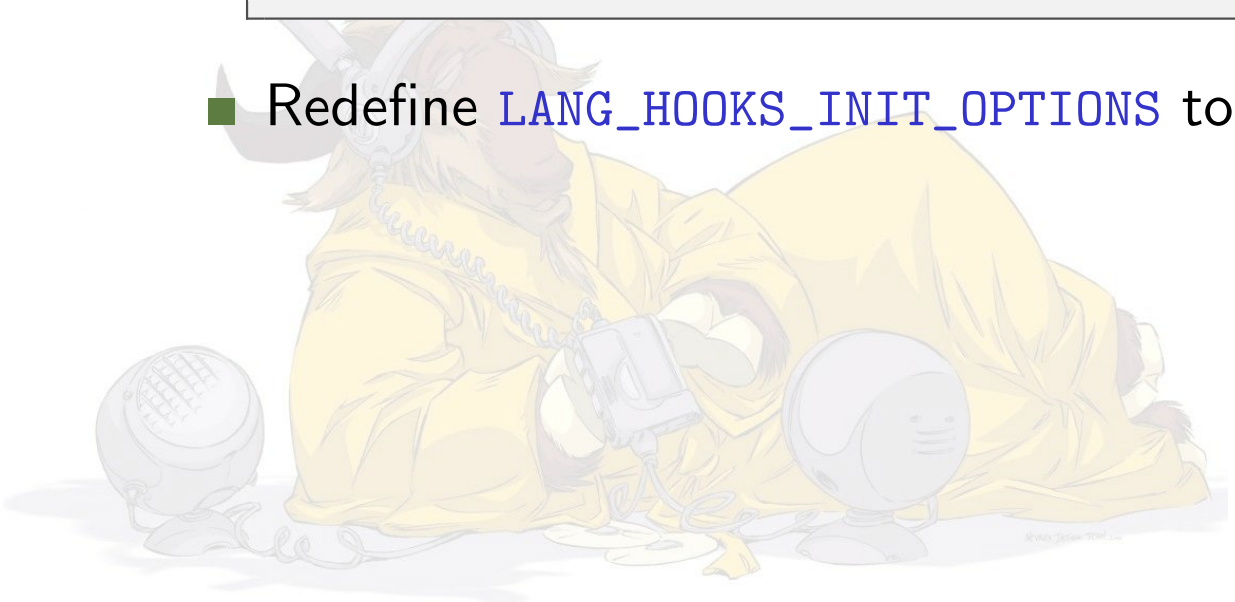
Changes to `trunk/gcc/hello-world/hello1.c`

Adding an Option

- Add a new callback:

```
unsigned int
hello_init_options (unsigned int argc,
                    const char **argv )
{
  return CL_Hello;
}
```

- Redefine `LANG_HOOKS_INIT_OPTIONS` to use it.



Changes to trunk/gcc/hello-world/hello1.c

Adding an Option

- Handle the option:

```
static int say_bye = 0;

static int
hello_handle_option (size_t scode,
                     const char *arg ATTRIBUTE_UNUSED,
                     int value ATTRIBUTE_UNUSED){
  enum opt_code code = (enum opt_code) scode;
  if (code == OPT_bye) {
      say_bye = 1;
      return 1;
  }
  return 0;
}
```

- Use it:

```
  const char *msg = say_bye ? "GoodBye" : "HelloWorld";
  tree hello_str = build_string_literal (msg);
```

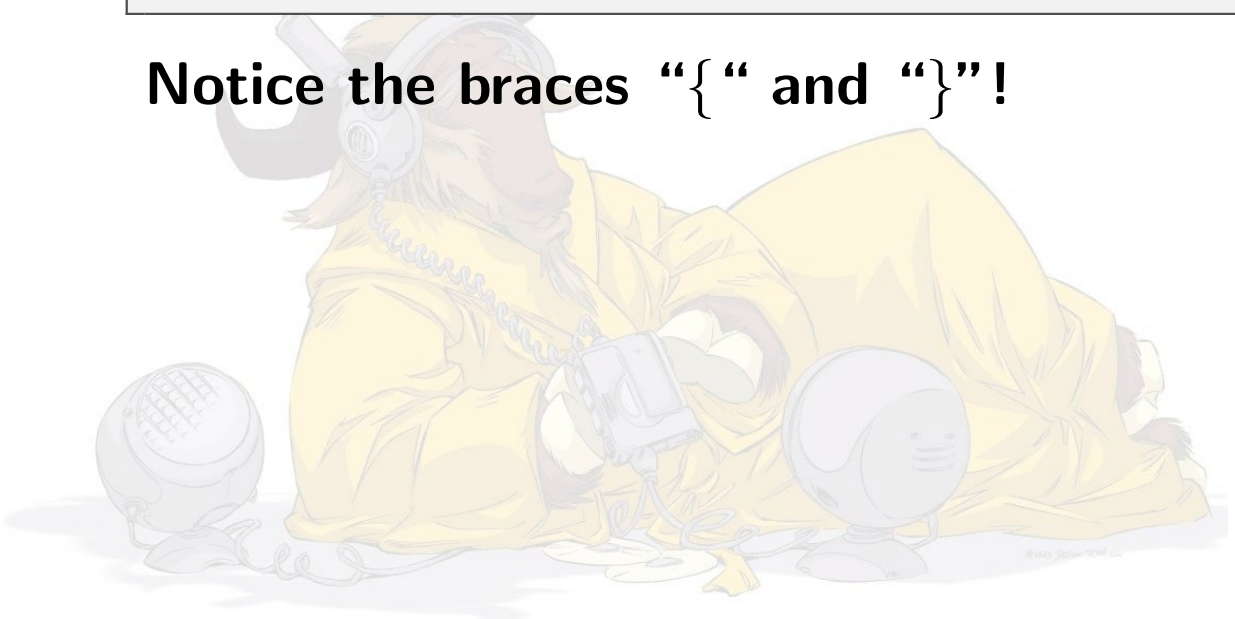
Changes to trunk/gcc/hello-world/lang-spec.h

Adding an Option

You must instruct the compiler driver to pass the option to the compiler. Just add %{bye} to the @hello entry:

```
{"@hello",
   "hello1 %i %{bye} %(cc1_options) "
   "%(invoke_as)", NULL, 0, 0
}
```

Notice the braces “{“ and “}”!



As before:

```
$ ./ghello -B. test.hello -o test
$ ./test
HelloWorld
```

With our new option:

```
$ ./ghello -B. -bye test.hello -o test
$ ./test
GoodBye
```

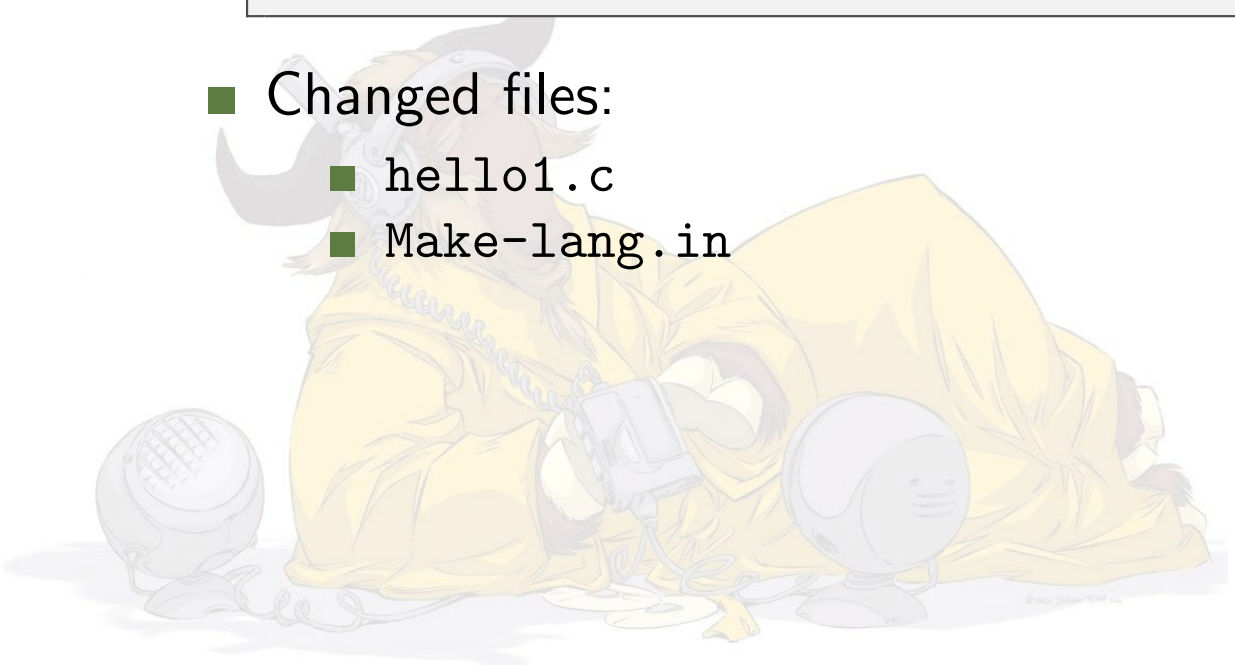
This step

Debug

- Compiler will produce tree dumps
- Commands:

```
$ cd /tmp/snapshot/build/gcc
$ cp /tmp/snapshot/hello-world/5-debug/* \
     /tmp/snapshot/trunk/gcc/hello-world
$ make hello1
```

- Changed files:
  - `hello1.c`
  - `Make-lang.in`



How to debug GCC

Debug

Since almost everything in GCC is a tree and these structures are quite huge, it may be difficult to debug, so there are two main tools to help with this task:

Tree dumps: dump internal trees to files at various stages. This require changes to source code that we will show.

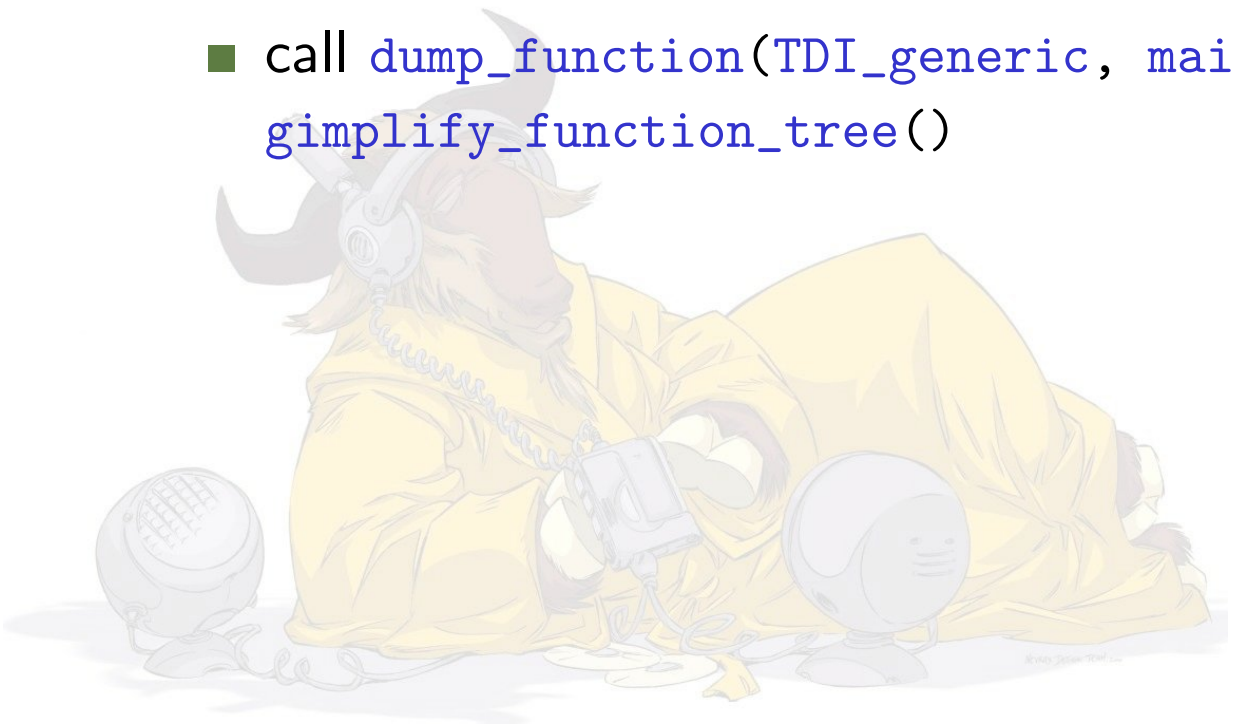
Tree browser: browse tree interactively. This let you navigate to children, print nodes, inspect attributes and much more. This requires no changes to source since it can be launched from GDB. Just call `debug_tree()` or `browse_tree()` on tree node:

```
$ gdb ./hello1
(gdb) b hello1.c:260
(gdb) r
(gdb) p debug_tree (main_fndecl)
(gdb) p browse_tree (main_fndecl)
```

Changes to `trunk/gcc/hello-world/hello1.c`

Debug

- include `tree-dump.h`
- call `dump_function(TDI_original, main_fndecl)` before calling `gimplify_function_tree()`
- call `dump_function(TDI_generic, main_fndecl)` after calling `gimplify_function_tree()`

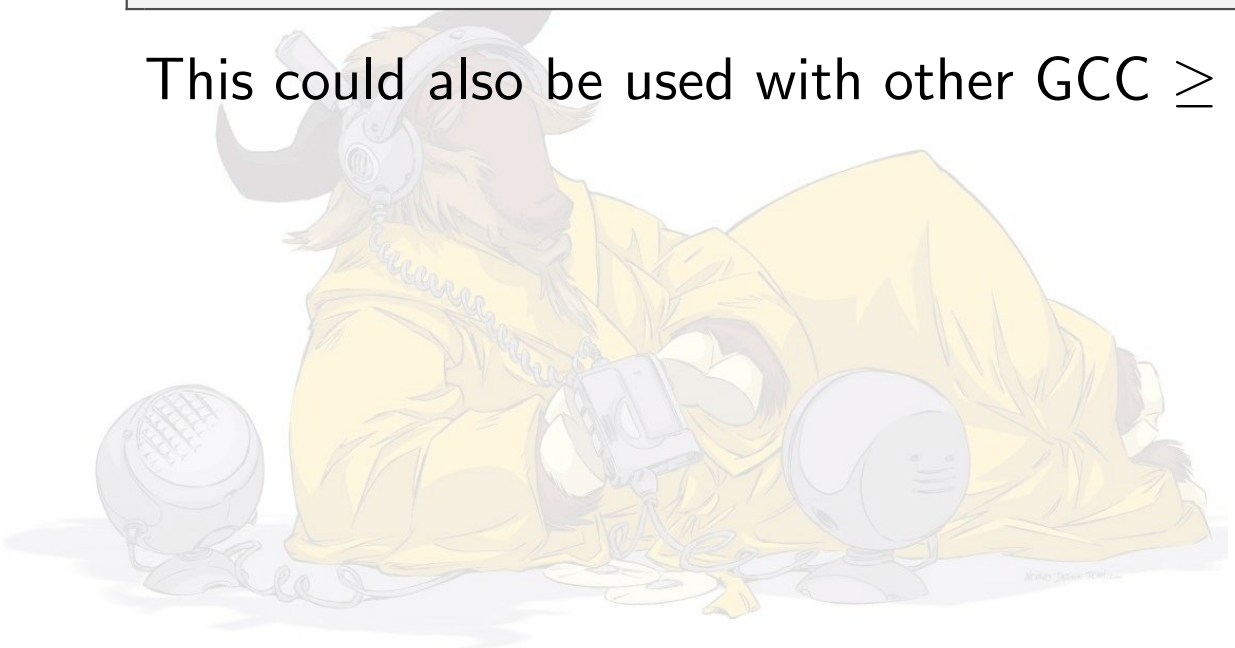


Include `$(TREE_DUMP_H)` as dependency of target `hello-world/hello1.o`

Debug example


```
$ ./ghello -B. -fdump-tree-all test.hello -o test
$ cat test.hello.t02.original
main ()
{
  puts ("HelloWorld");
}
```

This could also be used with other GCC $\geq$ 4 compilers!



Gustavo Sverzut Barbieri

Email:	barbieri@gmail.com
Website:	http://www.gustavobarbieri.com.br
ICQ:	17249123
MSN:	barbieri@gmail.com
Jabber:	gsbarbieri@jabber.org

Rafael Ávila de Espíndola

Email:	rafael.espindola@gmail.com
Jabber:	rafael.espindola@jabber.org