customizing GCC with MELT  
(a Lispy dialect)
Overview

1. Introduction
2. The MELT language
3. The MELT [meta-] plugin implementation
4. Conclusion

Slides available online at gcc-melt.org under (Creative Commons Attribution Share Alike 4.0 Unported license)
Caveat

**All opinions are mine only**

- I (Basile) don’t speak for my employer, CEA (or my institute LIST)
- I don’t speak for the GCC community
- I don’t speak for anyone else
- My opinions may be highly controversial
- My opinions may change
1 Introduction

2 The MELT language

3 The MELT [meta-] plugin implementation

4 Conclusion
Expected **audience** (FOSDEM2015 Lisp devroom):

- **familiar with some Lisp**-like language (Common Lisp, Scheme, Clojure, Emacs Lisp, ...) and with Linux or some Posix
- so able to code a toy Lisp evaluator in Lisp
- **free-software friendly** and knowledgable
- sometimes **using the Gcc\(^1\) compiler**
  (e.g. to build your favorite Lisp implementation runtime from its source code)
- so knowing a little bit the \textit{C} (or \textit{C++}) programming language
  (knowledge of \textit{gcc} internals is *not* pre-supposed)

\(^1\) *Gnu Compiler Collection*, no more *Gnu C Compiler*!
Introduction (GCC vs LLVM)

I don’t know LLVM internally!

- GCC (GNU compiler collection http://gcc.gnu.org/)
  - GNU, so GPLv3+ licensed (mostly) and FSF copyrighted (was initiated by R.M. Stallman)
  - compile many source languages (C, C++, Obj.C, D, Go, Fortran, Ada, ...)
  - compile for a lot of target processors and systems
  - still (usually) producing slightly faster code (when optimizing) than LLVM
  - legacy code base, now C++, active community and software
  - extensible thru plugins
  - gcc-5.0 (spring 2015) : 5.4MLOC (D.Wheeler sloccount, 225 M.US$) or ≈ 14.5MLOC, 86Mb .tar.bz2

- Clang/LLVM http://llvm.org/ 3.6
  - non-copyleft (BSD-like) license (so Apple is rumored to have proprietary variants);
    originated by C.Lattner (genuine C++)
  - a library libllvm (2.6MLOC) with a C/C++/Obj.C front-end clang (1.6MLOC)
  - with Clang compiles faster than GCC
  - more modern design, active community
  - less frontends (but newer standards) and backends than GCC
  - rumored to be easier to extend
Introduction (Gcc plugins)

Gcc is *extensible* thru plugins (≈ since gcc-4.5 in april 2010)

- **plugins should be free software**, GPL compatible
- there is (in principle) **no stable API** for plugins: A GCC 4.9 plugin should be improved to work with GCC 5.0
- the Gcc compiler gives **some plugin hooks**
- plugins **cannot enhance the source language** (e.g. add a new front-end) or the target processor (new back-end)
- plugins can **add optimization passes** and new attributes, pragmas, ...
- but very few Gcc plugins exist

gcc-5 also provides a **libgccjit** (**Just-In-Time code generation** library by D.Malcolm), also usable AOT like **libllvm**; LLVM always got a “JIT”

---

2 The **GCC runtime library exception**

https://www.gnu.org/licenses/gcc-exception-3.1.en.html forbids compilation of proprietary software with a non-free plugins, but IANAL; in the previous century GCC has been hurt by extensions feeding proprietary tools that made FSF and many people unhappy.
A compiler is working on internal representations of the user code it is currently compiling, much like a baker is kneading dough or pastry. (so the job of a compiler is mostly not parsing or machine code emission)
The `gcc` or `g++` are driver programs. They are starting

- `cc1` (for C) or `cc1plus` ... for the compiler proper (includes preprocessing), emitting assembly code.
- `as` for the assembler
- `lto1` for **Link Time Optimization**
- the `ld` or `gold` linker
- the `collect2` specific linker (creating a table of C++ constructors to be called for static data)

Run `g++ -v` instead of `g++` to understand what is going on.

**GCC plugins are `dlopen-ed by` `cc1`, `cc1plus`, `lto1` ...** So GCC “is mostly” `cc1plus`, or `cc1`, or `g951`, or `gnat1`, etc...

---

3 And also `gccgo` for Go, `gfortran` for Fortran, `gnat` for Ada, `gdc` for D, etc...

4 LTO may use linker plugins.
Introduction

Inside cc1plus

libiberty utilities
Ggc
Gcc Garbage Collector
pass manager
other utilities

frontend

tokens

E

middle-end

gmples

simple gimple passes

inter-procedural gimple passes

bar.cc

CC1 [+1]
overview

parser

genric trees

cc1

asm emitter

foo.s

Ggc

GCC MELT

January 31st, 2015 (FOSDEM, Brussels)

Basile Starynkevitch

your own pass in MELT

MELT

translator & runtime

RTL optim. passes

RTL generator

RTL

back-end

register allocator

Instr. scheduler

peephole optim.

RTL passes

RTL passes

RTL generator

RTL optim. passes

RTL

back-end

register allocator

Instr. scheduler

peephole optim.

RTL passes
Introduction (importance of optimizations)

- **current processors** (multi-core, out-of-order, pipelined, superscalar, branch prediction, many caches\(^5\)) are very complex, not like processors (68K, Sparc, i386) of 1980s, and increasingly far from the naive C computer model!

- **current languages standards** evolved too and “require” strong optimizations, e.g. in C++11

```cpp
#include <vector>
#include <algorithm>
int indexgreater(const std::vector<int>& a, int x) {
    return std::find_if (a.begin(), a.end(),
                          [&](int u){return u>x;})
             - a.begin(); }
```

is expected to be optimized without any calls. (the recent C++ standards are “impossible” without optimizations)

---

\(^5\) A cache miss requiring access to RAM lasts \(\approx 300\) cycles or machine instructions!
poor man’s (mark and sweep) garbage collector \texttt{ggc}
\,(does not handle local pointers! \textit{explicitly} triggered, e.g. between passes; some GC-ed data is \textit{explicitly} freed)

(a dozen of) specialized C++ code generators (e.g. \texttt{gengtype} for \texttt{ggc} generates marking routines from \texttt{GTY} annotations)

many (\approx 290) optimization passes (some very specialized, e.g. for \texttt{strlen}); see \texttt{gcc/passes.def}

\approx 2000 C++ \texttt{GTY}-ed data types inside the compiler, but...

\textbf{Generic Tree}-s = abstract syntax tree \approx S\text{-}expressions ; (\approx 223 \texttt{DEFTREECODE} in \texttt{gcc/tree.def})

\textbf{Gimple}-s = often 3 addresses instructions (like \texttt{x = y + z};) whose operands are trees : (41 \texttt{DEFGSCODE} in \texttt{gcc/gimple.def})

some “hooks” between compiler layers (front-end, middle-end, back-end)

code base growing by \approx 3\% each year

no introspection (à la GIRL in GTK)
Gcc customization (thru plugins in C++ or extensions in Melt) can be useful for:

- **validation of ad-hoc coding rules** like
  1. `pthread_mutex_lock` and `pthread_mutex_unlock` should be balanced and occur in the same function
  2. every call to `fork` should keep it result and test for $> 0$ or $= 0$ or $< 0$
  3. call to `fopen` should test against failure in the same routine

Such rules are API or industry **specific** (no free-software Coverity™-like tool)

- **fine-grained API or domain-specific typing**, e.g. of variadic routines: `Gcc` and `libc` already knows about `snprintf` thru some attribute(`(format(printf,3,4))`); But JANSSON library would like more type checks on its `json_pack` and GTK would be happy with a checked `g_object_set`

- **API or domain-specific optimizations**, e.g. `fprintf(stdout,...) ⇒ printf(...)`

- **profit of the hard work of the compiler** in other tools, e.g. `emacs` or `IDEs`

- **whole-project** metrics and (unsound or incomplete) analysis
Embedding an existing “scripting” language (Ocaml, Python\(^6\), Guile, Lua, Tcl, Javascript\(^7\) …) inside current Gcc is “impossible” and unrealistic:

- **hand-coding** the glue code is a **huge work**, incompatible with the **steady evolution** of Gcc
  (originally, I tried to glue Ocaml into Gcc for Frama-C, an LGPL static C source code prover and analyzer)

- **generating** the glue code automatically is **not achievable**
  (heterogeneity and legacy of coding styles inside Gcc)

- **difficult interaction between **Ggc** (the Gcc garbage collector) and the embedded language memory management**

But Gcc customization needs **expressivity**, notably **pattern matching** on Gcc internal representations, homoioiconic **meta-programming** and some **efficiency**

---

\(^6\)See D.Malcolm’s [GCC Python Plugin](https://git.fedorahosted.org/cgit/gcc-python-plugin.git) on

\(^7\)See Mozilla’s abandoned TreeHydra
Introduction  (Features of Melt)

NB: Melt was/is **incrementally designed and implemented**

- **free software** meta-plugin: GPLv3+ licensed, FSF copyrighted
- **Lisp-like** syntax and semantics (might have made it less attractive)
- efficient **generational copying garbage collector** above Ggc
  (values are born in a new region, later copied -when old enough- to Ggc heap)
- handle both first-class (Lisp-like) **values** and native **unboxed Gcc stuff**
  (like gimple, basic_block, tree, edge or long etc ...)
- evolves with **Gcc**\(^8\); in practice a release of Melt (1.1) can be built on two consecutive Gcc releases (e.g. Gcc 4.8 & 4.9)
- **pattern-matching** on both Gcc stuff and Melt values
- translated to (Gcc & Ggc friendly, dynamically compiled and dlopen-ed) **C++ code**
- can mix **C++** and Melt
- **meta-programming** thru Lisp-inspired **macros**
- **reflective**

---

\(^8\)Following and adapting Melt to Gcc is **labor-intensive**
1 Introduction

2 The MELT language

3 The MELT [meta-] plugin implementation

4 Conclusion
Hello World in MELT 😊

No display (à la Scheme), no format (à la Common Lisp), but shamefully 😊

```lisp
(let ( (two (+ 1 1)) ; a stuff
  
  (code_chunk hello_chk #{ // in $HELLO_CHK
    printf("hello world from $HELLO_CHK, two = %ld\n", $TWO);
  }#))
```

When running, you get something like

```
hello world from HELLO_CHK001, two = 2
```

C or C++ code chunks can be mixed with Melt. The “state symbol” hello_chk gets “gensym”-ed at code chunk expansion into C++ code. The locally let-bound variable two is a stuff (translated to some unboxed long C++ data), and in the code chunk $TWO is expanded to it.
**MELT values vs stuff**

MELT brings you **dynamically typed values** (à la Python, Scheme, Javascript):

- nil (is false), or **boxed** { strings, integers, *Tree*-s, *Gimples*, ...}, closures, tuples, lists, pairs, objects, homogeneous hash-tables . . .
- garbage collected by MELT using copying generational techniques (old generation is **GTY**-ed Ggc heap)
- quick allocation, favoring very temporary values
- **first class** citizens (every value has its discriminant - for objects their Melt class)

But **Gcc stuff** can be handled by MELT: **raw Gcc** tree-s, gimple-s, long-s, const char* strings, etc . . .

**Local data is garbage-collected**\(^9\) (values by MELT GC, stuff only by Ggc)

Type annotations like :**long**, :**cstring**, :**edge** or :**gimple** ... or :**value** may be needed in MELT code (but also :**auto** à la C++11)

\(^9\)Forwarding or marking routines for locals are generated!
Values in MELT

- list
  - descr
  - first
  - last

- pair
  - descr
  - hd
  - tl

- boxed gimple
  - descr
  - gimple

- GCC MELT values
  - hash 0x57de2f
  - 3 (length)
  - value 1
  - value 2
  - value 3
  - 3 (#fields)
  - 30017 (magic)

- 3-tuple
  - descr
  - first
  - last

- object
  - class
  - field 1
  - field 2
  - field 3
  - hash 0x57de2f
  - 3 (magic)
Some **MELT** language features

- **expression**-based language
- **local variable bindings** with `let` or `letrec`
- named `defun` and **anonymous** with `lambda functions` closures
- Smalltalk-like object system `defclass`, `defselector`, `instance` w. dynamic method dictionary (inside classes or discriminants)
- **iterative constructs** `forever`, `exit`, `again`, ... (but no tail-recursion)
- **pattern matching** with `match` (patterns with `?`, so `?_` is wildcard catch-all pattern)
- **dynamic evaluation** w. `eval`, quasi-quotation `backquote ≡ ` & `comma ≡ ,`
- **macros** with `defmacro` or local `:macro` binding in `let`
- **conditionals** with `if`, `cond`, `when`, `unless`, `or`, `and`, `gccif` (testing version of Gcc)
- multiple data results in function `return` and `multicall`
- many ways to mix C++ code with Melt code: `code_chunk`, `expr_chunk` and defining C++ generators `defprimitive`, `defcmatcher`, `defciterator`
- environment introspection `parent_module_environment` and `current_module_environment_reference`
The MELT language

the bizarre quote in MELT

As in every Lisp, ’2 is syntactic sugar for (quote 2)

Nobody codes like that ’2 in Lisp, but I do code like that in MELT

Remember: stuff ≠ values (but both are Melt “things”), hence the evaluations

- 2 → the stuff 2 (in C++, a raw unboxed (long)2)
- ’2 → the value 2 (in C++, a pointer to an allocated struct meltvalue_t....) of discriminant discr_constant_integer managed by the Melt garbage collector, so can be forwarded, when old enough, to the Ggc heap!
- "hello" → the stuff C-string (in C++, a raw unboxed (const char*)"hello")
- ’"hello" → the allocated value string hello of discr_string
- ’if → an interned symbol value, of discriminant class_symbol
- ’(f x) → an s-expr value of discriminant class_sexpr (with two fields :loca_location -some source file location- and :sexp_contents -a list of 2 pairs-)

So in MELT ’2 ≠ 2, unlike in every other Lisp
Defining primitives in *MELT*

A “primitive” is defined by giving the formals (with their types) and the type of the result, then the macro-string giving its C++ equivalent:

```plaintext
;; primitive to compute the length of a cstring
(defprimitive cstring_length (:cstring cstr) :long
  :doc #{Compute safely the length a C-string $CSTR. Gives 0 if null.}#
  #{(($CSTR)?strlen($CSTR):0)}#)
```

Don’t forget to be safe in primitives, code chunks, etc...

Notice the “keyword” annotations like `:cstring` for typing things. A documentation is generated using `:doc` annotations.

In formal argument lists, a `ctype` annotation applies to further formals. Initial formal ctype is of course `:value`. Default `let` binding ctype is `:auto`

*MELT* is **statically typed** for **stuff** and **dynamically typed** for **values**
How $+$ is defined in $MELT$?

(defprimitive +i (:long a b) :long
   :doc #{Integer binary addition of $a$ and $b.}#
   #{((a) + (b))}#)

Then $+$ is a variadic macro expanded to invoking $+i$
(in fact it is a bit more complex).
Defining functions in MELT

*Common Lisp* like syntax:

```
(defun multiple_every (tup f)  
  :doc #{Apply to every component of tuple $TUP$ and its index 
       the given function $F$. Return nil.}#  
  (if (is_multiple tup)  
      (if (is_closure f)  
         (foreach_in_multiple ;; a C-iterator 
                               (tup)  
                               (comp :long ix)  
                               (f comp ix))))))
```

*MELT* also accepts a *Scheme* like syntax to define functions

```
(define (multiple_every tup f) ...)
```

Anonymous functions with *lambda*
Call protocol for fixed-arity functions

- application of non-closure (e.g. objects) values (even reified primitives) gives nil
- function applications give a **primary result value** and perhaps **secondary results** (stuff or values)
- **first formal** (if given) should be a **value**
- **first** (actual) **argument** should also be a **value** or missing
- other formals and arguments should have the same c-type
- otherwise, all **remaining formals are cleared**
- missing arguments bind their formals to a cleared thing

So, with

```
((lambda (v :long i j k) some-body)
 :true 2 "not-a-long" 3)
```

inside **some-body** v is :true, i is 2, but both j and k are 0
variadic functions and loops

Use (:rest) in formals, and variadic form to dispatch and bind variadic arguments by type. Often with forever loops.

(defun add2out (out :rest)
  :doc #{Variadic function to add to an output $OUT various things. ...}
  (if (not (is_out out))
    (return))
  (forever argloop
    (variadic
      ( () (exit argloop))
      ( (:value v)
        (if (is_closure v)
          (variadic
            ((:value vv) (v out vv))
            ((:long ll) (v out ll))
            ;; etc...

No way (yet) to accumulate variadic arguments or to apply them elsewhere!
antiquotations

**syntactic sugar**: `$\alpha \equiv \text{backquote} \alpha$` and `$\epsilon \equiv \text{comma} \epsilon$` so is analogue to `' for quote.

Build a value, instance of `class_sexp` nearly like `' (f \ x) did.

```
(let ( (qfx `(f x))
      (onetwo (tuple '1 '2)) )
  `(g ,qfx ,onetwo))
```

$\rightarrow$ s-expr for `(g (f x) 1 2)

Notice that in antiquotations `(comma \epsilon)` may give *several* -or none- expressions if $\epsilon$ is some sequence. So no need of `, @ eta

(antiquotations are useful for macros)
defining c-iterators

A c-iterator expands into an iterative construct (à la for in C or C++). We give head and tail macro-string expansions.

(defciterator foreach_in_multiple
  (tup) ;start formal
eachtup ;state symbol
  (comp :long ix) ;local formals
  :doc #{Iterate in the given tuple $TUP for each component $COMP at index $IX}#

  ;; head or starting macrostring
  #{ /* start foreach_in_multiple $EACHTUP */
    long $EACHTUP#_ln = melt_multiple_length((melt_ptr_t)$TUP);
    for ($IX = 0;
      ($IX >= 0) && ($ix < $EACHTUP#_ln);
      $IX++) {
      $comp = melt_multiple_nth((melt_ptr_t)($TUP), $IX);
    }#

  ;; tail or ending macrostring
  #{ if ($IX<0) break;
    } /* end foreach_in_multiple $EACHTUP */ }
)
The MELT language

pattern-matching example

Deciding if a C function should be processed by some analysis pass.

**syntactic sugar**: \( ?\pi \equiv (\text{question } \pi) \) for patterns

```lisp
(defun meltframe_gate (pass)
  (with_cfun_decl ()
    (:tree cundec)
    (match cundec
      ( ?(tree_function_decl_named ?(cstring_containing "meltgc_") ?_)
        (return :true)
      )
      ( ?(tree_function_decl_named ?(cstring_prefixed "meltrout_") ?_)
        (return :true)
      )
      ( ?_ (return ()))))
)
```

Notice that \(?_\) is the **wildcard pattern** or joker.

Patterns occur in **match** expressions. The syntax separates expressions, patterns, **let**-bindings, formals, ...
defining a C-matcher

(defcmatcher tree_function_decl_named
  (:tree tr) ;matched
  ;; output
  (:cstring funame :tree trresult)
treefunam ;state symbol
:doc #{$TREE_FUNCTION_DECL_NAMED match a function declaration extracting
  its name $FUNAME and result tree decl $TRRESULT#}

;; test expansion
#{ /* tree_function_decl_named $TREEFUNAM ? */
  (($TR) && TREE_CODE($TR) == FUNCTION_DECL) }#

;; fill expansion
#{ /* tree_function_decl_named $TREEFUNAM ! */
  $FUNAME = NULL;
  $TRRESULT = NULL;
  if (DECL_NAME($tr))
    $FUNAME = IDENTIFIER_POINTER(DECL_NAME($TR));
  $TRRESULT = DECL_RESULT($TR); }#
)

Matching means testing if something fits, then destructuring it (filling step).

Matchers can also be defined with MELT functions using defunmatcher.
### cmatcher for a cstring starting with a given prefix

(defcmatcher cstring_prefixed
  (:cstring str cstr)
  ()
  strprefixed
  :doc #{The $CSTRING_PREFIX c-matcher matches a string $STR and test if it starts with the constant string $CSTR. The match fails if $STR is a null string or not prefixed by $CSTR.}#
  ;; test
  #{/* cstring_prefixed $STRPREFIXED test*/
    ($STR && $CSTR && !strncmp($STR, $CSTR, strlen ($CSTR))) }#
  ;; no fill
)
Defining a **MELT** hook

Such hooks are not *Gcc* hooks, but just functions compiled as ordinary C++ functions callable from C++ code.

```
(defhook hook_handle_attribute
  (:tree tr_in_node tr_name tr_args :long flags)
  (:tree tr_out_node :long out_no_add_attrs)
  :tree
  :predef HOOK_HANDLE_ATTRIBUTE
  (debug "hook_handle_attribute" " tr_in_node=" tr_in_node
    "; tr_name=" tr_name "; tr_args=" tr_args
    "; flags=" flags )
  (let (  
    (attrv ( ))  
  )
  (code_chunk getname_chk # /* hook_handle_attribute $GETNAME_CHK start */
    melt_assertmsg ("check good name",
      $TR_NAME
      && TREE_CODE($TR_NAME) == IDENTIFIER_NODE) ;
    $ATTRV = melt_get_mapstrings
      ((meltmapstrings_st*) $GCC_ATTRIBUTE_DICT,
        IDENTIFIER_POINTER($TR_NAME)) ;
    /* hook_handle_attribute $GETNAME_CHK end */
    #)
  (debug "hook_handle_attribute " "attrv=" attrv)
  ;; etc ......
```
1 Introduction

2 The MELT language

3 The MELT [meta-] plugin implementation

4 Conclusion
MELT implementation overview (> 100KLOC)

- runtime system:
  1. melt-runtime.h: 3795 lines, common header, included in
  2. melt-run.proto.h: includes Gcc plugin headers
  3. melt-runtime.cc: 13260 lines

- Melt generated parts of the runtime system:
  1. melt/generated/meltrunsup.h: 2800 lines the various data structures
  2. melt/generated/meltrunsup-inc.cc: 4638 lines, forwarding, copying, etc...

- the MELT (to C++) translator (63KLOC) in several phases:
  1. parsing into S-exprs of class_sexpr
  2. macro-expansion into AST, subclasses of class_source
  3. normalization in A-normal form\(^\text{10}\), so \((f (g x) y)\) is becoming almost like \((\text{let } (\theta (g x)) (f \theta y))\)
  4. generation of C++-like AST, subclasses of class_generated_c_code
  5. emission of C++ code

- C++ generated for the translator (1737KLOC melt/generated/warmelt*.cc)
- misc. (shell scripts and their generator)

\(^{10}\text{required by the copying Melt GC}\)
a big lot of C++ generated code

Melt is designed so that every value (even closures) is *computed* at runtime. (no “core image”\(^\text{11}\) à la sbcl.core like in most Lisp-s or in Ocaml)

A MELT “translation unit” or module is conceptually compiled into a C++ routine which takes a starting environment and returns a new environment. The starting environment is accessible with (parent_environment). The new current environment is contained in (current_module_environment_reference). Both are instances of class_environment defined as

```
(defclass class_environment
  :predef CLASS_ENVIRONMENT
  :super class_root
  :fields (env_bind ;the map of bindings
            env_prev ;the previous environment
            env_proc ;the procedure of this environment)
```

The compilation time of generated C++ code is the bottleneck

\(^{11}\)This could be improved, using Gcc “PCH” techniques
During the translation from MELT to C++, and in environments, symbols may have various bindings of different sub-classes of class_any_binding. The bound symbol is its :binder field.

- class_value_binding, exported with export_values
- class_primitive_binding for handling defprimitive, exported with export_values
- class_citerator_binding for handling defciterator, exported with export_values
- class_patmacro_binding for handling pattern-macros exported with export_patmacro
- class_macro_binding for macros (e.g. defined with defmacro), exported with export_macro, or inside a let annotated with :macro
- etc...

The handling of a symbol in operator position depends upon its bindings. Symbols have lexical-scoped bindings.
metaprogramming (e.g. in `defmacro`s and their invocation) is done “semilazily”, like `eval`: each dynamic evaluation is done by generating C++ code and `dlopen`-ing it.

C++ or C compilers are fast enough to be compatible with a read-eval-print-loop.

But meta-error handling is bad; some meta-errors are fatal. Could be improved.

```
(eval expr [env]) is working well enough.
```
"signal" handling and "asynchronous" I/O

Notice that Gcc is absolutely **not re-entrant**; however, *MELT* provides `register_paragraph_input_channel_handler`, `register_raw_input_channel_handler` **and** `register_alarm_timer` etc...: a file descriptor (e.g. socket or pipe) may recieve s-expressionong which will be apparently processed asynchronously. Actually, we are using `SIGIO` which sets a volatile flag tested using `MELT_CHECK_SIGNAL()` emitted at many places\(^{12}\)

In previous versions of *MELT* (1.0), we had a graphical GTK probe, but this is too inconvenient (stops gcc).

Today: JSON RPC [server and] client abilities (e.g. `do_blocking_jsonrpc2_call` & `json_parser_input_processor`)

Still missing: an external daemon and web interface, interacting with Gcc using Melt, to keep (e.g. in some database?) extracted properties of the compiled source code.

\(^{12}\)generating C++ code makes that reasonably easy, like the support of a copying GC
Some meta-data is kept in C code (files `*+meltdesc.c`) like:

```c
/* hash of preprocessed melt-run.h generating this */
const char melt_prepromd5meltrun[]="5bfc178c40b000dfbd23bbcb66857e91";

/* hexmd5checksum of primary C++ file */
const char melt_primaryhexmd5[]="b9b57cd8da15c812a5d8027af64166ee";

/* hexmd5checksum of secondary C++ files */
const char* const melt_secondaryhexmd5tab[]=
    /*nosecfile*/ (const char*)0,
    /*sechexmd5checksum warmelt-modes+01.cc #1 */ "c51b07cca977373ea3bc2a1f5ecbc1d3",
    /*sechexmd5checksum warmelt-modes+02.cc #2 */ "10ef7730cb92c4d26656bc7cef0b748c",
    /*sechexmd5checksum warmelt-modes+03.cc #3 */ "31ca48fea5dfba35b5e79ffa7ca5ea0e",
    (const char*)0;
```

These files are **compiled and parsed**\(^{13}\) to check consistency of `dlopen`-ed shared objects with their C++ counterparts.

---

\(^{13}\)The parsing of these C files happens in the Melt runtime - some `ccache` flavor
various flavors of Melt binary modules

The same *MELT* is translated into C++ code (with lots of `#line` directives in emitted C++) which is then compiled into binary module.

- **optimized modules**: compiled with `g++ -O2 -fPIC, (debug) ...` and `(assert_msg ...)` expressions are disabled.
- **quickly built modules**: compiled with `g++ -O0 -fPIC -DMELT_HAVE_DEBUG`, so `(assert_msg ...)` expressions are enabled.
- **debug noline modules**: compiled with `g++ -g -fPIC -DMELT_HAVE_DEBUG -DMELTGCC_NOLINENUMBERING` so skipping `#line`

*Melt* is internally running some `make` to compile the generated C++ code.

(Actually, bootstrapping has N to M dependencies, with complex generated shell scripts).
showing some code, etc...

Show code from `xtramelt-ana-simple.melt`

Complementary slides (much more Gcc focused):

**GCC plugins thru the MELT example** at Linux Foundation, march 2014
Taking advantage of compilers for doing more

Both free software and the general software industry need more “static analysis” tools which leverage on existing compilers.

- we need (several) free-software source code analyzers
- we need to formalize some coding rules
- compilers and their extensibility can be tremendously useful for more than compilation.
- free software cannot use only Coverity thru Github, it needs better free software tools
- special compilation mode "gcc -O∞" could profit from (slow) static analysis

I am interested in getting more work funded with Melt (industrial contracts, European collaborative research projects with DSL needs, etc...), or in similar approaches in other compilers (e.g. adding some DSL in LLVM?)
questions? Thanks!