Easily coding a GCC extension with MELT

Basile STARYNKEVITCH
basile@starynkevitch.net (or basile.starynkevitch@cea.fr)

October 26th, 2010, GCC Summit, Ottawa
slides' svn $Revision: 210 $
Contents

1. Introduction

2. Basic MELT usage and features
   - running MELT
   - MELT language syntax
   - MELT data \( (\text{things} = \text{values} + \text{stuff}) \)
   - connecting GCC to MELT

3. Pattern matching

4. Coding passes in MELT

5. Conclusion and future work
Introduction

Basic MELT usage and features
- running MELT
- MELT language syntax
- MELT data \( \text{things} = \text{values} + \text{stuff} \)
- connecting GCC to MELT

Pattern matching

Coding passes in MELT

Conclusion and future work
What is MELT?

- in Debian, `melt = command line media player and video editor.`
  [http://www.mltframework.org](http://www.mltframework.org)

- for LaTeX and Ocaml\(^1\) users, Melt (by Romain Bardou) allows you to program LaTeX documents using ocaml

- in GCC, MELT is an infrastructure and a domain specific language to ease development of specific GCC extensions

This talk is about GCC MELT 😊

MELT ≡ *Middle End Lisp Translator*

but MELT is perhaps not only for the middle-end!

\(^1\) [http://caml.inria.fr/](http://caml.inria.fr/)
The big GCC MELT picture

250 passes in GCC!

C front-end
bee.c

C++ front-end
foo.cc

Fortran front-end
bar.f90

erm representation(s)

Ada front-end
cat.adb
derm representation(s)

general tree

LTO "front-end"
dog.o

Link Time Optimizations

Melt runtime & translator
melt.so warmelt*.so yourpass.melt

yourpass.so

GCC MELT

October 26, 2010 – GCC Summit, Ottawa
About these slides

- All **opinions are only mine**, not of: my employer (CEA, LIST)$^2$, funding agency (DGCIS)$^3$, my colleagues or interns, or the GCC community.
- Some slides, in particular this one, are **extra slides** and may be skipped. They are pinky, and are **marked** with a ♠ at the bottom.
- These slides are made with Pdf $\LaTeX$ and beamer$^4$
- The **MELT examples are really run** when producing the slides.
- These slides are available (in PDF) on [www.gcc-melt.org](http://www.gcc-melt.org) site and attached to [http://gcc.gnu.org/wiki/MELT](http://gcc.gnu.org/wiki/MELT)
- **syntax colorization with** `pygments`, *minted* style, `contrib/pygmentize-melt`$^5$ **script.**

---

$^2$[www-list.cea.fr](http://www-list.cea.fr)

$^3$[www.industrie.gouv.fr/portail/une/dgcis.html](http://www.industrie.gouv.fr/portail/une/dgcis.html)

$^4$using `-shell-escape` but sadly without Romain Bardou’s Melt

$^5$Some bugs remain in that script, in particular for `$var`s in macro-strings, e.g. code chunks...
Expected audience

1. using GCC to compile important or big [free] software
2. able to compile newest GCC from its source code
3. interested in extending or hacking GCC
4. somehow familiar with GCC internals:
   - overall organization: driver, front-ends, middle-end, back-ends
   - major internal representations: Generic/Tree, Gimple, ...
   - the many (∼ 250) passes inside GCC
   - knowledgable of the GCC plugins machinery
5. pragmatically curious about domain specific or scripting languages
6. not necessarily familiar with lisp dialect, but not scared of parenthesis

---

6 MELT is probably overkill to compile hello-world.c
7 Or even wanting to contribute to GCC MELT
8 But small knoweldge of Scheme, Emacs-Lisp, or Common Lisp can help!
Questions to the audience

1. who coded a middle-end pass in GCC?
2. who knows some “functional / applicative” language, e.g. Ocaml, Haskell, Ruby, Python, Scala, Clojure . . . and used anonymous functions ($\lambda$-calculus)?
3. who knows (and did code) lispy languages, i.e. Scheme, Common Lisp, Emacs Lisp?
4. who knows about pattern matching?
5. who coded a GCC plugin?
To learn more about GCC...

- lots of web resources [gcc.gnu.org](http://gcc.gnu.org), and help of GCC community
- internal GCC documentations and source code availability
- major internal representations:
  - Generic/Tree (AST = abstract syntax tree), see gcc/tree.def and gcc/tree.h source files, “union” of \( \approx 180 \) cases.
  - Gimple mostly “3 address” instructions (with Tree operands), see gcc/gimple.def and gcc/gimple.h files, “union” of \( \approx 36 \) cases.
- etc...

organization of passes:
- used set of passes depends of optimization (and of source code)
- see gcc/tree-passes.c file
- run gcc -fdump-tree-all
- your extension usually add some [your own] passes\(^9\)

- GCC has a garbage collector Ggc [GTY annotations processed by gengtype] and MELT strongly depends upon it

---
\(^9\)Finding what pass to add and where is tricky!
When you shouldn’t use MELT

- if you cannot have and build plugins on your GCC compiler
- if your computer is too small to build GCC\(^{10}\)
- to develop core GCC (trunk) enhancements \((\text{to be pushed inside } \text{gcc})\)
  - you can’t use MELT since it is a plugin\(^{11}\) and a non-standard language!
  - but you can prototype your work with MELT and experiment new ideas with MELT
- to patch existing GCC passes:
  MELT provide extensions thru the plugins framework, which enable adding your new passes \(\text{(but not really modifying existing passes, unless you replace them)}\).
- to add new front-ends or new back-ends to GCC
  since it is impossible today \(\text{(gcc 4.5)}\) thru plugins
- for any GCC enhancement not doable with the plugin mechanism

\text{MELT uses extensively the plugin hooks}

\(^{10}\)You probably will need 2Gbytes of RAM and 3Gbytes of disk to compile MELT generated code

\(^{11}\)The MELT infrastructure is itself a plugin, unless you use the MELT branch; but even the MELT branch is using plugin hooks.
When and why use MELT

Assuming that you are able to code GCC plugins and learned a bit about MELT

1. to **experiment new ideas**\(^\text{12}\) inside GCC
   MELT should provide you increased productivity, since it is more expressive than C programming language

2. to **develop specific GCC extensions**
   MELT enables development of your application-, domain-, corporation-, project-, specific GCC extensions (which are not economically doable in C plugins)

With MELT, you can take advantage of the power of GCC for many source code related activities (and use GCC for source-code related tasks outside of ordinary compilation).

---

\(^{12}\) e.g. prototyping a new middle-end optimization pass in MELT!
MELT is available (but evolving) today!

```
svn co \
svn://gcc.gnu.org/svn/gcc/branches/melt-branch gcc-melt
```

also look on gcc-melt.org for releases

1. as a **plugin** to unchanged gcc-4.5\textsuperscript{13} or to the trunk
2. as a **branch** [executable program gcc-melt]
   code very close\textsuperscript{14} to gcc trunk (future 4.6). The main differences are shorter program options e.g. `gcc-melt -fmelt-mode=\mu` instead of `gcc -fplugin-arg-melt-mode=\mu`

Of course MELT is free software ©FSF and GPLv3 licensed.

Your MELT extensions are possible with the same conditions as GCC plugins. It is preferable to have free GPLv3 extensions. See http://www.gnu.org/licenses/gcc-exception.html for details.

\textsuperscript{13}Use the `contrib/build-melt-plugin.sh` script
\textsuperscript{14}merged with trunk more than weekly
MELT implementation details

MELT branch follows closely the trunk (svn repository files =):
- gcc/common.opt, gcc/toplevel.c: patched for -fmelt-α options
- gcc/Makefile.in: build and bootstrap\(^{15}\) MELT
- gcc/melt-runtime.[ch]: the MELT runtime (GC, module loading, low-level operations with values)
- gcc/melt-make.mk, gcc/melt-module.mk: Makefiles for MELT and for MELT modules
- gcc/melt-predef.ist: list of predefined MELT objects
- gcc/melt-run.proto.h: template generating melt-run.h included by every MELT generated C files.
- gcc/melt/warmelt-*.melt: the “bootstrapped” MELT system and translator
- gcc/melt/generated/warmelt-∗0.c: corresponding generated C files
- gcc/melt/xtramelt-*.melt: extra MELT files (e.g. MELT operations on gimple..., simple MELT passes)

Hints for building GCC MELT [branch]

- same advices as for GCC (in particular *build tree ≠ source tree*)
  See http://gcc.gnu.org/install/
- same dependencies as for GCC (trunk or 4.5)
- the Parma Polyhedra Library PPL (preferably 0.11) should be configured with `--enable-interfaces=c` at least\(^{16}\)
- after updating the GCC MELT branch source from svn repository by running `./contrib/gcc_update` in the source tree, don’t forget to `rm -f gcc/melt-runtime.o gcc/warmelt*.o` in the build tree!
- perhaps use **ccache** carefully from http://ccache.samba.org/
- parallel make `-j` useful for **ccl** but useless for **melt.encap** (since MELT is generating C files and compiling them just after)

---

\(^{16}\)This is the issue if you get undefined symbols when linking **ccl** for symbols like `ppl_io_asprint_Polyhedron`
**about my system** Debian/Testing/AMD64 or Ubuntu/Maeverick/AMD64

**shell run 1**

```
echo $GCCMELTSOURCE $GCCMELTBUILD; gcc-melt -v |& grep 'gcc version'
⇒
/usr/src/Lang/basile-melt-gcc /usr/src/Lang/_Obj_Gcc_Melt
gcc version 4.6.0 20101019 (experimental) [melt-branch revision 165706] (GCC)
```

**shell run 2**

```
grep bash.*configure GCCMELTBUILD/ config.status0
⇒
/usr/src/Lang/_Obj_Gcc_Melt/config.status: set X '/bin/bash'
'/usr/src/Lang/basile-melt-gcc/configure' '--program-suffix=-melt'
'--libdir=/usr/local/lib/gcc-melt' '--libexecdir=/usr/local/libexec/gcc-melt'
'--with-gxx-include-dir=/usr/local/lib/gcc-melt/include/c++/
'--with-mpc-include=/usr/local/include' '--with-mpc-lib=/usr/local/lib'
'--enable-maintainer-mode' '--enable-checks=tree,gc' '--disable-bootstrap'
'--disable-multilib' '--enable-version-specific-runtime-libs' '--enable-plugin'
'CC=gcc' 'CFLAGS=-O -g' '--with-ppl-include-dir=/usr/local/include'
'--with-ppl-lib-dir=/usr/local/lib' '--enable-languages=c,c++,lto'
```
for developing significant MELT extensions, the GCC MELT branch is preferable (in particular because ENABLE_CHECKING gives you more debug related features, but is usually disabled in released GCC compilers)

it could be worthwhile to build your gcc-4.5 with --enable-checks for ease of development of your MELT extensions

running gengtype for plugins is very painful in GCC 4.5. Consider copying manually contrib/gt-melt-runtime-plugin-4.5.h to gcc/gt-melt-runtime.h in your plugin build tree.

read INSTALL/README-MELT-PLUGIN

read and run contrib/build-melt-plugin.sh accordingly

please report any bugs about MELT as plugin
Why use MELT for specific extensions?

MELT is mostly useful for (application-, domain-, industry-, project-) specific GCC extensions, e.g.

- particular warnings or checks, like:
  - warn when the result of `fopen` is not tested...
  - type variadic functions like `g_object_set` in Gtk
- specific optimizations, like `fprintf(stdout, . . .) ⇒ printf(. . .)`
- coding rule validation, like In C++, “ensure base classes common to more than one derived class are virtual” (HICPP 3.3.15). or Every call to `pthread_mutex_lock` should be followed by a similar call to `pthread_mutex_unlock` in the same block.
- source code processing (e.g. aspect oriented programming, retro-engineering, re-factoring tasks)

Notes:

1. most such extensions are specific and probably don’t belong inside GCC.
2. same arguments go for plugins coded in C; however, MELT is believed to increase your productivity while coding such extensions.
MELT extensions vs coding plugins in C

MELT (being based on the plugin machinery) permits the same extensions as GCC plugins coded in C.

- C is efficient at execution time, but difficult to use for custom middle-end processing (C is not an easy language to code compilers or static analyzers)
- specific extensions (to be coded in MELT) needed to be coded quickly
- MELT has many features tailored to processing of GCC internals
  1. automatic memory management (a powerful garbage collector)
  2. powerful pattern matching
  3. high-level programming styles: object-oriented, functional, applicative, reflexive abilities, dynamic typing, meta-programming...
  4. MELT is very tightly related to GCC internals
  5. MELT code is translated to C code\(^\text{17}\) suited for GCC

- gluing an existing scripting language implementation (e.g. Ruby, Python, Ocaml) into GCC is not realistic (because GCC API is not stable, huge, and not well defined by C functions.)

\(^\text{17}\) Often, the compilation of that generated C code to a \texttt{dlopen-ed} \*so is the bottleneck.
# MELT garbage collector versus Ggc

<table>
<thead>
<tr>
<th>Design:</th>
<th>MELT GC</th>
<th>Ggc(^{18})</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>designed from start in parallel with the MELT language</td>
<td>added as a crude hack</td>
</tr>
<tr>
<td>Manage:</td>
<td>MELT values</td>
<td>stuff = GTY-ed structures</td>
</tr>
<tr>
<td>Based upon:</td>
<td>Ggc →</td>
<td>system malloc</td>
</tr>
<tr>
<td>Time:</td>
<td>(O(\lambda) \quad \lambda = \text{size of live values})</td>
<td>(O(\sigma) \quad \sigma = \text{total memory size})</td>
</tr>
<tr>
<td>Type:</td>
<td>generational, copying</td>
<td>precise, mark &amp; sweep</td>
</tr>
<tr>
<td>Roots:</td>
<td>local variables and static</td>
<td>only static GTY-ed data</td>
</tr>
<tr>
<td>Invocation:</td>
<td>implicitly, when needed</td>
<td>between passes only</td>
</tr>
<tr>
<td>Implementation:</td>
<td>runtime suited for code generator</td>
<td>quick and dirty hack(^{19})</td>
</tr>
</tbody>
</table>

## Suit for:
- short-lived temporary values
- quasi-permanent data

## Allocation:
- very quick
- a non-trivial function call

## Usage:
- in generated C code
- in hand-written C

The **MELT** old generation is the **Ggc** heap, so **MELT** is compatible with **Ggc**\(^{20}\). A minor GC happens after each **MELT** pass.

---

\(^{18}\)Ggc is the “garbage collector” inside **GCC**. `gcc/ggc*.[ch]` and `gcc/gengtype*`

\(^{19}\)the `ggc` functions & GTY annotations preprocessed by `gengtype`

\(^{20}\)but **MELT** is not compatible with **Pch**
MELT code suitable for its garbage collector

MELT GC requires that every MELT call frame is explicited (as a C struct) in the generated code, and containing every local thing (e.g. values and GTY-ed data).

Typical C code looks almost like (in gcc/melt-runtime.c)

```c
melt_ptr_t meltgc_new_gimple (meltobject_ptr_t discr_p, gimple g) {
    MELT_ENTERFRAME (2, NULL);
    #define bgimplev meltfram__.mcfr_varptr[0]
    #define discrv meltfram__.mcfr_varptr[1]
    #define object_discrv ((meltobject_ptr_t)(discrv))
    discrv = (void *) discr_p ? : MELT_PREDEF (DISCR_GIMPLE);
    if (melt_magic_discr ((melt_ptr_t) discrv) != MELTOBMAG_OBJECT) goto end;
    if (object_discrv->object_magic != MELTOBMAG_GIMPLE) goto end;
    bgimplev = meltgc_allocate (sizeof (struct meltgimple_st), 0);
    ((struct meltgimple_st *) (bgimplev))->discr = discrv;
    ((struct meltgimple_st *) (bgimplev))->val = g;
    end:
    MELT_EXITFRAME ();
    return (melt_ptr_t) bgimplev;
}
```
Conventions expected by MELT garbage collector

- **MELT** call frames are declared as `struct ure meltframe__`, cleared and linked to the previous (with `MELT_ENTERFRAME`);
- every C **MELT** value formal argument should be copied into that frame;
- this call frame should be unlinked with `MELT_EXITFRAME`;
- **every MELT value** should be a local field in its `meltframe__`;
- → The generated code cannot have `xv=f (g (yv), zv)`; but should have a unique temporary `tmpv= g (yv); xv =f (tmpv, zv)`. This is easier to achieve in generated C code.
- the **MELT** GC can be triggered at every **MELT** allocation\(^{21}\)
- updates inside touched **MELT** values should be signaled (write barrier `meltgc_touch`)

---

\(^{21}\text{When the allocation birth zone is full, a copying minor GC is triggered. Every live value is copied out of it, and the birth zone (of e.g. a megaword) can be freed at once. Sometimes a full GC occurs by calling `ggc_collect` just after the minor GC. Minor GCs are forced before returning inside GCC code, e.g. at end of every GCC pass coded in MELT.}\)
Gory details about MELT garbage collection

- the MELT GC handles only MELT values; GCC stuff (gimple, tree, ...) is managed by Ggc
- “strongly” typed GC: MELT values are handled differently from GCC stuff
- explicit local frame allocation using MELT_ENTERFRAME and MELT_EXITFRAME macros
- local frame is meltfram__ with local values accessed thru #define meltfptr meltfram__.mcfr_varptr etc.
- write barrier: updated values should go thru meltgc_touch.
- coding in hand-written C for MELT is somehow painful
- but almost all the C code is generated
- the only way to allocate MELT values is meltgc_allocate, which may call melt_garbcoll (which sometimes calls ggc_collect)
- MELT GC is tunable thru parameters melt-minor-zone, melt-full-threshold & melt-full-period
Terminology

plugin: only melt.so as plugin to gcc-4.5 built from gcc/melt-runtime.[ch] and loaded when -fplugin=melt.so

MELT runtime: the functions from gcc/melt-runtime.[ch]

MELT language: our lispy domain specific language

MELT file: a .melt file in our lispy MELT language

source file: the *.c *.cc *.f90 ... files compiled by GCC

MELT generated file: the .c file generated from .melt by the MELT translator

MELT module: the .so shared object, dynamically loaded from the MELT runtime (calling conventions specific to MELT and unrelated to GCC plugins)

MELT mode: usually, a word or identifier ω passed with gcc-4.5 -fplugin=melt.so -fplugin-arg-melt-mode=ω or gcc-melt -fmelt-mode=ω. No extra processing happens without a mode.

GCC stuff: any internal (often GTY-ed) data inside gcc-4.5 (or trunk), e.g. long or gimple or edge ...

---

22 Runtime provided by the melt.so plugin for gcc-4.5, or included in gcc-melt

23 A future infix form would be called *.milt
# Table of Contents

1. Introduction
2. Basic MELT usage and features
   - running MELT
   - MELT language syntax
   - MELT data \((things = values + stuff)\)
   - connecting GCC to MELT
3. Pattern matching
4. Coding passes in MELT
5. Conclusion and future work
Running MELT

Without any mode, [nothing more happens, so] running gcc-melt -O -c foo.c is the same as gcc-trunk -O -c foo.c or gcc-4.5 -fplugin=melt.so -O -c foo.c

The gcc compiler driver requires some source file. In some modes, it is useful to compile an empty.c file (simply to get cc1 started):

- translate your foo.melt to module foo.so:
  
gcc-melt -fmelt-mode=translateetomodule \ 
    -fmelt-arg=foo.melt -c empty.c

- translate your foo.melt and run it immediately when compiling bar.c:
  
gcc-melt -fmelt-mode=runfile -fmelt-arg=foo.melt \ 
    -O -c bar.c

Notice that in both cases, the cc1 started by gcc-melt is itself generating a C file and fork-ing its compilation\(^{24}\) into a MELT module.

\(^{24}\)Actually cc1 is pex_execute-ing a make command using melt-module.mk!
Running MELT as plugin

With `melt.so` as plugin to **GCC 4.5**, the commands are:

- **translate your `foo.melt` to module `foo.so`:**
  
  ```
  gcc-4.5 -fplugin=melt.so \
  -fplugin-arg-melt-mode=translatetomodule \
  -fplugin-arg-melt-arg=foo.melt -c empty.c
  ```

- **translate your `foo.melt` and run it immediately when compiling `bar.c`:**
  
  ```
  gcc-4.5 -fplugin=melt.so \\
  -fplugin-arg-melt-mode=runfile \\
  -fplugin-arg-melt-arg=foo.melt -O -c bar.c
  ```

**In general,** \( \forall \alpha \)

- `-fplugin-arg-melt-\alpha` for `gcc-4.5 -fplugin=melt.so`

\( \equiv -fmelt-\alpha `\) for `gcc-melt`
Existing MELT modes
they don’t fit all on the screen!

shell run

gcc-melt -fmelt-mode=help -c empty.c

⇒

* help : MELT help about available modes.
* justscan : install a pass scanning all the code
* makedoc : generate .texi documentation from .melt source files;

ARGLIST= input file, ...; OUTPUT= generated file
* makegreen : enable a pass finding fprintf to stdout...
* nop : a mode doing nothing.
* rundebug : translate and run a .melt file for debug;

ARGUMENT= input file; [OUTPUT=generated C]
* runfile : translate and run a .melt file.

ARGUMENT= input file; [OUTPUT=generated C].
* smallana : install a small analysis pass
* translateddebug : translate a .melt file to .so module for debug;

ARGUMENT= input file; OUTPUT= generated module *.so;

generates also *.c and no MELT line number;

Useful for running gdb on the module.
MELT files, directories, paths

- **MELT builtin source directory**  for “MELT system” files
  *warmelt-*.melt* (the MELT infrastructure and translator) and
  *xtramelt-*.melt* (extra MELT passes and functions). Also contains the
  corresponding translation *warmelt-*.c*.

- **MELT builtin module directory** for the “MELT system”, contains the
  executable modules *warmelt-*.so and *xtramelt-*.so* and their
  default list *melt-default-modules.modlis*

- colon separated **MELT source path** of directories, used to find MELT
  source files *.melt & *.c*, from program argument
  `-fmelt-source-path` or environment variable `GCCMELT_SOURCE_PATH`.

- **MELT module path** of directories, used to find MELT module files *
  .so* (or
  their *
  .modlis* list), from `-fmelt-module-path` or `GCCMELT_MODULE_PATH`.

- **MELT colon separated initial module list** `-fmelt-init` is a list of
  modules or @ module lists. Defaults to @@ for @melt-default-modules

---

25 e.g. `/usr/local/libexec/gcc/gcc-melt/x86_64-unknown-linux-gnu/4.6.0/melt-source` which is always sought.

26 e.g. `/usr/local/libexec/gcc/gcc-melt/x86_64-unknown-linux-gnu/4.6.0/melt-module`
other significant MELT program options

- a single argument \(-\text{fmelt-arg}=\alpha\) (e.g. a MELT input file to translate);
- a secondary argument \(-\text{fmelt-secondarg}=\alpha\);
- an output argument \(-\text{fmelt-output}=\omega\) (e.g. a MELT generated C file);
- an option argument \(-\text{fmelt-option}=\alpha\);
- a temporary directory \(-\text{fmelt-tempdir}=\delta\);
- a comma separated argument list \(-\text{fmelt-arglist}=\alpha_1, \alpha_2\);
- the makefile to build MELT modules from MELT generated C files
  \(-\text{fmelt-module-makefile}\)
- the make command to built them \(-\text{fmelt-module-make-command}\)

\[27\] the builtin default is 
/usr/local/libexec/gcc-melt/gcc/x86_64-unknown-linux-gnu/4.6.0/melt-module.mk
MELT lispy dialect

Why a Lispy syntax? Because I am lazy\(^{28}\), and because of Emacs-Lisp and Guile!

As usual:

- Lisp $\equiv$ Lots of Insipid Stupid Parenthesis
- Parenthesis are very important: $\phi \neq (\phi)$ and always matched
- MELT syntactic constructs are always prefix\(^{29}\) like $(\Omega\ \alpha_1\ \ldots\ \alpha_n)$ where $\Omega$ is the operator and the $\alpha_i$ are the $n \geq 0$ operands or arguments.

---

\(^{28}\)no time to create a sexy syntax and an Emacs mode for it

\(^{29}\)Except syntactic sugar like ’ or ? etc.
MELT vs other lisps

- MELT is **lexically scoped** (à la Scheme) with a single namespace.
- MELT can handle non-value data called **stuff**
- **nil**, noted (), is the only **false** value
- several major syntactic constructs are noted nearly like in other Lisps: `let` `if` `lambda` `cond` `defun` `definstance` `setq` ...
- symbols (e.g. `if` or `foo`) are objects of **class_symbol**
- keywords (e.g. `:else` or `:gimple`) are objects of **class_keyword**
- source s-expressions are parsed as instances of **class_sexpr** and know their location
- lists are not just simply linked pairs
- no familiar operations: there is no `car`, `cons`, or `+` in MELT
- `'2 ≠ 2`, because `'2 is a [boxed long] **value** but 2 is some `:long` **stuff**

---

30 Every non-nil value is true. But the stuff 0 is false!
31 The first element of a list is gotten with `list_first_element`
32 The + primitive binary operation handles unboxed long stuff, not values!
identifiers or **symbols** are case insensitive, so `let` is the same as `LeT`
symbols may contain a few special characters, e.g. `<i` or `+i`  
**“keywords”** start with a colon like `:long` or `:else`  
comments start with a semi-colon `;` up to end of line
strings are nearly like in C (so "a\nb" has three characters...).
**macro-strings** (for C code chunks with MELT “hole” variables) with `#{ ... }#` so

```c
#{/*$P*/printf("a=%ld\n", $A);}#
≡
("/*" p "*/printf("a=%ld\n", " a ");")
```

`é` is **syntactic sugar** for `(quote é)`, e.g. `'"ab"` ≡ `(quote "ab")` and
`'if` ≡ `(quote if)`

`?é` is **syntactic sugar** for `(question é)`, so `?x` ≡ `(question if)`

33 “keyword” is lisp parlance, not as syntactically important as in C or Ada.
Hello world in MELT

```plaintext
;;; file hello.melt

;;; a comment for the generated C code
(comment "hello world is public domain")

;;; a code chunk containing C
(code_chunk say-hello-chunk

    #\{printf("hello from MELT %s:%d\n", __FILE__, __LINE__);\}#)

;;; eof hello.melt
```

the **comment** is translated into a C comment. The **code_chunk** adds some C code chunk in the generated C file. **MELT** source line numbers are preserved in the generated C file thru **#line** directives.

gcc-melt -fmelt-mode=runfile -fmelt-arg=hello.melt -c empty.c

⇒

```plaintext
hello from MELT hello.melt:8
```
running MELT

building the hello module and running it

shell run 5

gcc-melt -fmelt-mode=translatetomodule -fmelt-arg=hello.melt -c empty.c

⇒

shell run 6

ls -lt hello.*

⇒

-rw-r--r-- 1 meltuser meltuser 10992 Oct 26 03:09 hello.c
-rwxr-xr-x 1 meltuser meltuser 135408 Oct 26 03:09 hello.so
-rw-r--r-- 1 meltuser meltuser 10989 Oct 26 03:09 hello.c%
-rw-r--r-- 1 meltuser meltuser 286 Oct 22 16:31 hello.melt
Basic MELT usage and features

running MELT

shell run 7

gcc-melt -fmelt-init=@@:hello -c empty.c
⇒

Nothing happened, since there is no mode.

shell run 8

gcc-melt -fmelt-module-path=. -fmelt-init=@@:hello -fmelt-mode=nop -c empty.c
⇒

hello from MELT hello.melt:8
MELT files and modules

Very Scheme inspired:
- a MELT source file $\phi\cdot\text{melt}$ contains a sequence of expressions and is compiled into a melt module $\phi\cdot\text{so}$ (with a big initialization function for the evaluation of each expression).
- the melt runtime `dlopen`-s the $\phi\cdot\text{so}$ module
- modules are loaded in sequence, and the MELT system has an initial sequence (containing the translator and initial environments). User modules are loaded after these.
- a module consumes an environment and produces a new one, as a translated side effect of evaluating the expressions in $\phi\cdot\text{melt}$. Only exported bindings are visible outside of the module.

In practical terms, a MELT source file contains defining and/or side-effecting expressions. It can install new modes and new GCC passes etc. Translating $\phi\cdot\text{melt}$ to $\phi\cdot\text{c}$ is much faster than compiling the generated $\phi\cdot\text{c}$ to $\phi\cdot\text{so}$.

---

34 It is also possible to compile a sequence of expressions from the MELT heap into a module!
35 the modules `warmelt-*\cdot\text{so}` and `xtramelt-*\cdot\text{so}` listed in `melt-default-modules` abreviated by `@@`
### main MELT syntactic constructs

<table>
<thead>
<tr>
<th>expressions where $n \geq 0$ and $p \geq 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>application</strong></td>
</tr>
<tr>
<td><strong>assignment</strong></td>
</tr>
<tr>
<td><strong>message send</strong></td>
</tr>
<tr>
<td><strong>let expression</strong></td>
</tr>
<tr>
<td><strong>sequence</strong></td>
</tr>
<tr>
<td><strong>abstraction</strong>(^{37})</td>
</tr>
<tr>
<td><strong>pattern matching</strong></td>
</tr>
</tbody>
</table>

---

\(^{36}\)So the **let** of MELT is like the **let**\(^*\) of Scheme!

\(^{37}\)abstractions are constructive expressions and may appear in letrec bindings
A cleared thing\(^{38}\) (represented by all zero bits) is nil, or the long 0 stuff, or the null gimple or tree ... stuff. It is false.

### Conditional Expressions

<table>
<thead>
<tr>
<th>Test</th>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional</td>
<td>(\text{cond} \ k_1 \ldots k_n)</td>
<td>Evaluate conditions (k_i) until one is satisfied</td>
</tr>
<tr>
<td>Conjunction</td>
<td>(\text{and} \ k_1 \ldots k_n k')</td>
<td>If (k_1) and then (k_2) ... and then (k_n) is “true” (non nil or non zero) then (k') else the cleared thing of same type</td>
</tr>
<tr>
<td>Disjunction</td>
<td>(\text{or} \ \delta_1 \ldots \delta_n)</td>
<td>The first of the (\delta_i) which is “true” (non nil, or zero, ...)</td>
</tr>
</tbody>
</table>

In a cond conditional expression, every condition \(k_i\) -except perhaps the last- is like \((\gamma_i \ \epsilon_{i,1} \ldots \epsilon_{i,p_i} \ \epsilon')\) with \(p_i \geq 0\). The first such condition for which \(\gamma_i\) is “true” gets its sub-expressions \(\epsilon_{i,j}\) evaluated sequentially for their side-effects and gives its \(\epsilon'\). The last condition can be \((\text{else} \ \epsilon_1 \ldots \epsilon_n \ \epsilon')\), is triggered if all previous conditions failed, and (with the sub-expressions \(\epsilon_j\) evaluated sequentially for their side-effects) gives its \(\epsilon'\)

---

\(^{38}\) Every local thing (value, stuff ...) is cleared at start of its containing MELT function.
### more expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>loop</strong></td>
<td>(forever λ α₁ ... αₙ)</td>
</tr>
<tr>
<td></td>
<td>loop indefinitely on the αᵢ which may exit</td>
</tr>
<tr>
<td><strong>exit</strong></td>
<td>(exit λ ε₁ ... εₙ ε')</td>
</tr>
<tr>
<td></td>
<td>exit enclosing loop λ after side-effects of εᵢ and result of ε'</td>
</tr>
<tr>
<td><strong>return</strong></td>
<td>(return ε ε₁ ... εₙ)</td>
</tr>
<tr>
<td></td>
<td>return ε as the main result, and the εᵢ as secondary results</td>
</tr>
<tr>
<td><strong>multiple call</strong></td>
<td>(multicall φ κ ε₁...εₙ ε′)</td>
</tr>
<tr>
<td></td>
<td>locally bind formals φ to main and secondary result[s] of application or</td>
</tr>
<tr>
<td></td>
<td>send κ and evaluate the εᵢ for side-effects and ε’ for result</td>
</tr>
<tr>
<td><strong>recursive let</strong></td>
<td>(letrec (β₁...βₙ) ε₁...εₚ)</td>
</tr>
<tr>
<td></td>
<td>with co-recursive <em>constructive</em> bindings βᵢ evaluate sub-expressions εⱼ</td>
</tr>
<tr>
<td><strong>field access</strong></td>
<td>(get_field :Φ ε)</td>
</tr>
<tr>
<td></td>
<td>if ε gives an appropriate object retrieves its field Φ, otherwise nil</td>
</tr>
<tr>
<td><strong>unsafe field access</strong></td>
<td>(unsafe_get_field :Φ ε)</td>
</tr>
<tr>
<td></td>
<td>unsafe access without check like above</td>
</tr>
<tr>
<td><strong>object update</strong></td>
<td>(put_fields ε :Φ₁ ε₁ ... :Φₙ εₙ)</td>
</tr>
<tr>
<td></td>
<td>safely update (if appropriate) in object given by ε each field Φᵢ by value of εᵢ</td>
</tr>
</tbody>
</table>

---

39 i.e. if the value ω of ε is an object which is a direct or indirect instance of the class defining field Φ.

40 Only for MELT gurus, since it may crash!

41 i.e. update object ω only if the value ω of ε is an object which is a direct or indirect instance of the class defining each field Φᵢ.
constructive expressions

<table>
<thead>
<tr>
<th>list</th>
<th>(list α₁ ... αₙ)</th>
<th>make a list of n values αᵢ</th>
</tr>
</thead>
<tbody>
<tr>
<td>tuple</td>
<td>(tuple α₁ ... αₙ)</td>
<td>make a tuple of n values αᵢ</td>
</tr>
<tr>
<td>instance</td>
<td>(instance κ :Φ₁ ε₁ ... :Φₙ εₙ)</td>
<td>make an instance of class κ and n fields Φᵢ set to value εᵢ</td>
</tr>
</tbody>
</table>

Abstractions (lambda expressions) are also constructive.

Constructive expressions may be recursively bound in letrec:

```
(letrec (
  (a (list b c))
  (b (tuple a b))
  (c (lambda (x y) (if (== x a) b y)))
  (d (instance class_container :container_value a))
)
(c d bar))
```

Note: contrarily to Scheme, **MELT has no tail recursive calls.**
Every [recursive] MELT call grows the stack (because it is translated to a C call).
# Basic MELT usage and features

MELT language syntax

## Expressions about names

| for functions | (defun \( \nu \ \phi \ \epsilon_1 \ldots \ \epsilon_n \ \epsilon' \)) | define function \( \nu \) with formal arguments \( \phi \) and body \( \epsilon_1 \ldots \ \epsilon_n \ \epsilon' \) |
| for primitives | (defprimitive \( \nu \ \phi \ : \theta \ \eta \)) | define primitive \( \nu \) with formal arguments \( \phi \) and result c-type \( \theta \) by macro-string expansion \( \eta \) |
| for c-iterators | (defciterator \( \nu \ \Phi \ \sigma \ \Psi \ \eta \ \eta' \)) | define c-iterator \( \nu \) with input formals \( \Phi \), state symbol \( \sigma \), local formals \( \Psi \), start expansion \( \eta \), end expansion \( \eta' \) |
| for c-matchers | (defcmatcher \( \nu \ \Phi \ \Psi \ \sigma \ \eta \ \eta' \)) | define c-matcher \( \nu \) with input formals \( \Phi \) [the matched thing, then other inputs], output formals \( \Psi \), state symbol \( \sigma \), test expansion \( \eta \), fill expansion \( \eta' \) |
| for fun-matchers | (defunmatcher \( \nu \ \Phi \ \Psi \ \epsilon \)) | define funmatcher \( \nu \) with input formals \( \Phi \), output formals \( \Psi \), with function \( \epsilon \) |

## Expressions defining names

| of values | (export_value \( \nu_1 \ldots \)) | export the names \( \nu_i \) as bindings of value (e.g. of functions, objects, matcher) |
| of macros | (export_macro \( \nu \ \epsilon \)) | export name \( \nu \) as a binding of a macro (expanded by the \( \epsilon \) function) |
| of classes | (export_class \( \nu_1 \ldots \)) | export every class name \( \nu \) and all their fields (as value bindings) |
| as synonym | (export_synonym \( \nu \ \nu' \)) | export the new name \( \nu \) as a synonym of the existing name \( \nu' \) |
### Basic MELT usage and features

### MELT language syntax

#### miscellaneous expressions

<table>
<thead>
<tr>
<th>For all:</th>
<th>expressions for debugging</th>
<th>meta-conditionals</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>debug message</strong></td>
<td><code>(debug_msg \(\epsilon \ \mu\))</code></td>
<td>conditional on a preprocessor symbol: emitted C code is <code>#if \(\sigma\) code for \(\epsilon\)</code> <code>#else</code> <code>code for \(\epsilon'\)</code> <code>#endif</code></td>
</tr>
<tr>
<td><strong>assert check</strong></td>
<td><code>(assert_msg \(\mu \ \tau\))</code></td>
<td>the (\epsilon_i) are translated only if the GCC translating them has version prefix string (\beta)</td>
</tr>
<tr>
<td><strong>warning</strong></td>
<td><code>(compile_warning \(\mu \ \epsilon\))</code></td>
<td></td>
</tr>
<tr>
<td><strong>Cpp test</strong></td>
<td><code>(cppif \(\sigma \ \epsilon \ \epsilon'\))</code></td>
<td></td>
</tr>
<tr>
<td><strong>Version test</strong></td>
<td><code>(gccif \(\beta \ \epsilon_1 \ \ldots\))</code></td>
<td></td>
</tr>
</tbody>
</table>

#### For gurus:

**introspective expressions**

| **Parent environment**       | `(parent_module_environment)`                                           | gives the previous module environment                                                                                                                         |
| **Current environment**      | `(current_module_environment_container)`                                | gives the container of the current module’s environment                                                                                                       |
MELT values [and stuff] from inside

An example: **boxed** gimple (values containing a raw gimple pointer) from melt-runtime.h

```c
struct GTY (()) meltgimple_st {
    meltobject_ptr_t descr;
    gimple val;
};
```

Every MELT value has a **discriminant**. Such a discriminant is a non-null MELT object (not all values are objects).

MELT values are **first-class**. Non-value stuff (e.g. raw gimple or long) is second-class.

<table>
<thead>
<tr>
<th>Things</th>
<th>Values</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>any data relevant to MELT</td>
<td>first class, with discriminant</td>
<td>second class, like gimple or long</td>
</tr>
</tbody>
</table>
stuff handled by MELT

Potentially, any data inside GCC. In practice

- **long raw integers**
- **cstring**, i.e. **constant array of chars** like "mutable" - the const char [] data! These are not heap-allocated!
- **Major GCC data types** like **gimple, gimple_seq, tree, basic_block, edge, loop, rtx, rtvec & bitmap**
- **PPL data** like **ppl_coefficient** etc.
- **void** (like in C, for absence of result)
- **MELT value-s** are not really stuff...

Any **GTY-ed type** could be handled as some stuff.  

Stuff are second-class⁴³ in **MELT**. Handling first-class values is simpler.

---

⁴² Adding extra **GTY-ed data types** requires additional code in the **MELT** runtime.

⁴³ They can only be secondary arguments or results.
Most stuff can be boxed as a simple value, e.g.

- boxed *gimple*
- boxed *tree*
- boxed *long*
- etc.
- the nil value

Strings are immutable (heap-allocated) MELT values. Several discriminants are possible e.g. `discr_verbatim_string` and `discr_string`. String buffers contain a growable sequence of characters.

---

44One could also have several kinds or colors of boxed *gimple* by making several discriminants for them.
MELT aggregate values

1. **tuples**, i.e. fixed sequence of values
2. pairs (head is any value, tail is a pair or nil)
3. **lists** (it knows its first and last pair)
4. **MELT objects** (described below)
5. **closures** (a MELT function with values)
6. associative **hash-maps** associating keys to non-nil values:
   - string maps (= dictionary); keys are string values.
   - object maps; keys are MELT objects
   - `gimple, tree, edge` ... maps; key are non nil stuff (all of the same C type).

**Hash-maps** are quite important, it is the only way to **associate MELT values to major GCC data types** like `gimple` or `basic_block`.

---

45 So **MELT** lists are **not** like in Scheme or Lisp!
46 But we don’t have maps associating non-object values -like closures, boxed `edge`s, or tuples- to other values.
aggregate values (figure)

 GCC MELT values

 3-tuple
  object

 boxed gimple

 pair

 pair

 pair

 list

 MELT data \( (\text{things} = \text{values} + \text{stuff}) \)
MELT objects representation

```c
typedef struct meltobject_st* meltobject_ptr_t;
struct GTY((variable_size)) meltobject_st {
  /* for objects, the discriminant is their class */
  meltobject_ptr_t obj_class;
  unsigned obj_hash;     /* hash code of the object */
  unsigned short obj_num;
  /* discriminate the melt_un containing it as discr */
  #define object_magic obj_num
  unsigned short obj_len;
  melt_ptr_t GTY((length ("%h.obj_len")))
    obj_vartab[FLEXIBLE_DIM];
};
```

The `obj_num (≡ object_magic)` field is set at most once to a non-zero short.

The `object_magic` field of discriminants (starting every MELT value) is describing in the union `melt_un` the GTY-ed field.
The MELT values in `melt-runtime.h`

typedef union melt_un* melt_ptr_t;
union GTY ((desc ("%0.u_discr->object_magic"))) melt_un {
    meltobject_ptr_t GTY ((skip)) u_discr;
    struct meltmultiple_st
        GTY ((tag ("MELTOBMAG_MULTIPLE"))) u_multiple;
    struct meltobject_st
        GTY ((tag ("MELTOBMAG_OBJECT"))) u_object;
    struct meltlist_st
        GTY ((tag ("MELTOBMAG_LIST"))) u_list;
    struct meltclosure_st
        GTY ((tag ("MELTOBMAG_CLOSURE"))) u_closure;
    struct meltgimple_st
        GTY ((tag ("MELTOBMAG_GIMPLE"))) u_gimple;
    struct meltmapobjects_st
        GTY ((tag ("MELTOBMAG_MAPOBJECTS"))) u_mapobjects;
    struct meltmapgimples_st
        GTY ((tag ("MELTOBMAG_MAPGIMPLES"))) u_mapgimples;
    // etc ...
};
MELT classes and object creation

e.g. MELT mode for GCC pass

Dynamic object creation with instance (from gcc/melt/xtramelt-ana-simple.melt)

(defun makegreen_docmd (cmd moduldata) ;; unused formals
  (let ((greenpass
    (instance class_gcc_gimple_pass
      :named_name "melt_greenpass"
      :gccpass_gate makegreenpass_gate
      :gccpass_exec makegreenpass_exec
      :gccpass_data
        (make_maptree descr_map_trees 100)
      :gccpass_properties_required ()
    ))
  ;; register our pass after the "phiopt" pass
  (install_melt_gcc_pass greenpass "after" "phiopt" 0)
  ;; return non-nil, e.g. our greenpass, to accept the mode
  greenpass))

In the above makegreen_docmd function -called from the makegreen MELT mode-, an instance of class_gcc_gimple_pass is created and registered as a GCC pass.
static creation with `definstance` which defines a MELT variable statically bound to an instance:
definition and installation of a MELT mode

```
(definstance makegreen_mode
class_melt_mode
  :named_name "makegreen"
  :meltmode_help
    "enable a pass finding fprintf to stdout..."
  :meltmode_fun makegreen_docmd
)
(install_melt_mode makegreen_mode)
```
defining a MELT class

MELT classes are defined with `defclass`, by giving the super-class and the sequence of fields (instance variables).

```lisp
;; class describing MELT options
(defclass class_option_descriptor
  :doc #\{A class describing MELT options for -fmelt\-option=\}#
  :super class_root
  :fields (optdesc_name
            optdesc_fun
            optdesc_help)
)
```

There is a tree of classes. The topmost class (without any proper fields) is `class_root`. Conventionally, the class name starts with `class_`. Often, the field names share a common prefix.

**Discriminants, classes and fields are reified**: they are objects (of class `class_discriminant`, `class_class`, `class_field` respectively).
globally unique field names

The field names should be **globally** unique. This enables the safe `get_field` contract, which tests that the accessed value is an object of the right class:

```
(get_field :container_value cont)
```

≡

```
(if (is_a cont class_container)
    (unsafe_get_field :container_value cont)
()
)
```

The `is_a` primitive tests that `cont` is an object, and is an instance of `class_container` or a sub-class. The discriminant of a value (e.g. the class of an instance) can be accessed with the `discrim` primitive. The discriminant of nil is conventionally `discr_null_reciever`. The subclassing relation is tested with `subclass_of` or `subclass_or_eq` primitives.
selector, methods and message sending

Selectors are “method names”. They are defined with defselector, similar to definstance.

```
;; selector for debugging output
(defselector dbg_output class_selector
  :formals (recv dbginfo :long depth)
)
```

Once a selector is defined, every use of it as operator is a message send. The optional :formals given in a selector enable checking the signature of message sends. Almost always, the class of a selector is class_selector.\(^\text{47}\)

Selectors exist independently of the discriminants (or classes) understanding them. Sending a message with a selector \(\sigma\) to a receiving value \(v\) whose discriminant \(\delta\) don’t know about \(\sigma\) is a no-op and gives nil.

\(^\text{47}\)But it could be a sub-class of class_selector.
message sending and method installation

The sending of a message is \((\sigma \ \rho \ \alpha_1 \ ... \ \alpha_n)\) syntactically like an application\(^{48}\), it has a selector \(\sigma\), a receiving value \(\rho\) and secondary arguments \(\alpha_i\):

\[
\text{(dbg_output curval dbg_i (+i depth 1))}
\]

Sending a message in MELT is more similar to Smalltalk sends than to methods calls in C++ or Java.

A method is just a function installed\(^{49}\) with \text{install_method}. There is no special name (unlike \text{this} in C++) for the formal receiver.

---

\[\begin{align*}
1 & \text{;; null debug output} \\
2 & (\text{defun dbgout_null_method} \ (\text{self dbg_i :long depth}) \\
3 & \quad (\text{let} \ (\text{out} \ (\text{unsafe_get_field :dbgi_out dbg_i}))) \\
4 & \quad (\text{add2out_strconst out "()"})) \\
5 & \quad (\text{install_method discr_null_receiver dbg_output} \\
6 & \quad \quad \text{dbgout_null_method})
\end{align*}\]

\(^{48}\)However, the selector in a message send is always a constant selector name. In contrast, in function applications, the applied function can be computed with a complex expression.

\(^{49}\)Methods can be installed and removed dynamically at any time, independently of their discriminants and selectors.
message sending machinery

Every discriminant (i.e. class) has a method dictionary, and a super-discriminant (i.e. the super-class of a class). To send a message with selector $\sigma$ to receiver $\rho$ and extra arguments $\alpha_i$ of discriminant $\delta$

1. look in the method dictionary $^5_1$ of $\delta$ for a closure associated to $\sigma$
2. if a closure $\kappa$ is found, use it as the method function and apply it (i.e. $\kappa$) to $\rho$ and the $\alpha_i$ ...
3. otherwise, look in the super-discriminant $\delta'$ (e.g. super-class) of $\delta$, and repeat by replacing $\delta$ with $\delta'$.

The topmost discriminant is `discr_any_reciever`. It is the ultimate super-discriminant of every other discriminant or class.

Message sending is a bit slower than function application.

---

$^5_0$ Messages can be sent to any MELT value, even nil, closures, boxed gimples, maps...

$^5_1$ The field `:disc_methodict` in discriminants is an object hash map whose keys are selectors, associated to method closures.
In formal argument lists, keywords like `:long` indicate the ctype of next formals, so the formal argument list `(x y :long n p :value z)` means: formals `x y` are values, `n p` are long stuff, then `z` is value. Likewise, `let` binds sequentially\(^{52}\) local variables

```scheme
(let ( (:long n 0)
       (:gimple g (gimple_content gb))
 )
  (f a n g))
```

Each stuff type has its ctype keyword, like `:gimple :gimple_seq :basic_block :long :cstring etc.`

\(^{52}\) So MELT’s `let` is like `let*` in Scheme!
MELT closures = functional values

A MELT function can take both values and stuff as arguments. First argument (if any) should be a value\(^{53}\). A function application\(^{54}\) returns a value\(^{55}\). Only values can be closed. Closures can be passed as arguments. Named functions are defined with \texttt{defun} (as in Emacs Lisp)

\begin{verbatim}
;; apply \texttt{f} to each boxed gimple in a gimple seq \texttt{gseq}
(defun do_each_gimpleseq (f :gimple_seq gseq)
  (each_in_gimpleseq
   (gseq) (:gimple g)
   (let ( (gplval (make_gimple discr_gimple g)) )
     (f gplval)))
)
\end{verbatim}

\(^{53}\) It is the receiver in methods
\(^{54}\) And hence message sendings
\(^{55}\) Secondary results can be stuff.
application mechanism

- preferably, formals arguments and actual parameters should be similar (in number, in ctype). First formal and primary parameter should be a `:value`.
- no variable arity MELT functions exist.
- (in C code) secondary arguments (and secondary results if any) are passed thru arrays of `union meltparam_un` described by a constant string (produced by the MELT translator).
- if a formal argument mismatch its actual parameter, it is cleared with the rest of the formals
- primary result is a `:value`
- secondary actual results are handled similarly.

⇒ mismatched arguments or results are cleared.

Applying a non-closure value gives nil.
connecting the GCC API to MELT

Several linguistic devices exist:

- `code_chunk` to add C code inside MELT code (like `asm` adds assembly code inside C code).
- `defprimitive` to define primitive operations (by a translation “template” to C).
- `defciterator` to define iterative constructs
- `defcmatcher` to define pattern-matching constructs

Each take advantage of `macro-strings` (mixing strings for C code fragments with MELT symbols for “holes”).
C code in MELT with `code_chunk-S`

Useful to include unique C code. For example (with `modnamestr` bound to a string value):

```c
(code_chunk
  checkerrorsaftercompilation
  #{ /*$checkerrorsaftercompilation*/
    if (melt_error_counter>0)
      melt_fatal_error ("MELT translation of %s halted: got %ld MELT errors",
                         melt_string_str($modnamestr),
                         melt_error_counter);
  }#)
```

Becomes translated to:

```c
/*CHECKERRORSAFTERCOMPILATION__1*/
if (melt_error_counter>0)
  melt_fatal_error ("MELT translation of %s halted: got %ld MELT errors",
                   melt_string_str(/*__.MODNAMSTR__V3*/ meltfptr[2]),
                   melt_error_counter);
```

Notice the substitution of `$-names`: the state symbol `checkerrorsaftercompilation` with the unique `CHECKERRORSAFTERCOMPILATION__1` and `modnamestr` with `/*__.MODNAMSTR__V3*/ meltfptr[2]`
primitives with their C “template”

```
(defprimitive basicblock_nb_succ (:basic_block bb) :long
  #{((BB)?EDGE_COUNT(BB->succs):0)}#)
```

The ctype of primitive application actual parameters is checked\(^{56}\).

A primitive is translated into a C block or instruction if its resulting ctype is :void.

Otherwise, it is translated into a C expression.

When defining your primitives, make them safer by checking for null pointers!

\(^{56}\)MELT gives an error at translation time if basicblock_nb_succ is given a thing which is not a :basic_block, (like a value or a long stuff).
defining iterative constructs with \texttt{defciterator}

The \texttt{each\_in\_gimpleseq} iterates for every gimple \( g \) inside a \texttt{gimple\_seq} \( gseq \):

\begin{verbatim}
;;; iterate on a gimpleseq
(defciterator each_in_gimpleseq
  (:gimple_seq gseq) ;; start formals
  eachgimplseq
  (:gimple g) ;; state symbol
  ;;; local formals
  ;;; before expansion
  #{
gimple_stmt_iterator gsi_${eachgimplseq};
  if ($gseq)
    for (gsi_${eachgimplseq} = gsi_start ($gseq);
      !gsi_end_p (gsi_${eachgimplseq});
      gsi_next (&gsi_${eachgimplseq}) { 
    $g = gsi_stmt (gsi_${eachgimplseq});
  }#
  ;;; after expansion
  #{ /*end $eachgimplseq*/ } }# )
\end{verbatim}
Translation (partly) in C

```c
/*citerblock EACH_IN_GIMPLESEQ*/ {
    gimple_stmt_iterator gsi_cit1__EACHGIMPLESEQ;
    if (/*_?*/ meltfram__.loc_GIMPLE_SEQ__o0)
        for (gsi_cit1__EACHGIMPLESEQ = gsi_start (/*_?*/ meltfram__.loc_GIMPLE_SEQ__o0);
            !gsi_end_p (gsi_cit1__EACHGIMPLESEQ);
            gsi_next (&gsi_cit1__EACHGIMPLESEQ)) {
            /*_?*/ meltfram__.loc_GIMPLE__o1 = gsi_stmt (gsi_cit1__EACHGIMPLESEQ);
            MELT_LOCATION("xtramelt-ana-base.melt:2502:/ quasiblock");
            #ifndef MELTGCC_NOLINENUMBERING
            #line 2502 ".../xtramelt-ana-base.melt"
            #endif /*MELTGCC_NOLINENUMBERING*/
            /*_.GPLVAL__V4*/ meltfptr[3] =
                (meltgc_new_gimple((meltobject_ptr_t)((/*!DISCR_GIMPLE*/ meltfrout->tabval[0])),(/*_?*/ meltfram__.loc_GIMPLE__o1)));
            MELT_LOCATION("xtramelt-ana-base.melt:2503:/ apply");
            /*apply*/{
                /*_.F__V5*/ meltfptr[4] = melt_apply ((meltclosure_ptr_t)(/*_.F__V2*/ meltfptr[1]), (melt_ptr_t)(/*_.GPLVAL__V4*/ meltfptr[3]), (""), (union meltparam_un*)0, "", (union meltparam_un*)0);
            }
            /*_.LET___V3*/ meltfptr[2] = /*_.F__V5*/ meltfptr[4];
            MELT_LOCATION("xtramelt-ana-base.melt:2502:/ clear");
            /*clear*/ /*_.GPLVAL__V4*/ meltfptr[3] = 0 ;
            /*clear*/ /*_.F__V5*/ meltfptr[4] = 0 ;
        }
    MELT_LOCATION("xtramelt-ana-base.melt:2500:/ clear");
    /*clear*/ /*_?*/ meltfram__.loc_GIMPLE__o1 = 0 ;
    /*clear*/ /*_.LET___V3*/ meltfptr[2] = 0 ;
}
```
Table of Contents

1 Introduction

2 Basic MELT usage and features
   ● running MELT
   ● MELT language syntax
   ● MELT data \((\text{things} = \text{values} + \text{stuff})\)
   ● connecting GCC to MELT

3 Pattern matching

4 Coding passes in MELT

5 Conclusion and future work
Patterns in MELT

- patterns are major nested syntactic constructs (like expressions are) in MELT.
- patterns appear in the `match` expression, with the `?` notation.
- patterns are a **destructuring mechanism**. They consume some thing (the matched data) $\mu$, and extract several data-s from it (transmitted to sub-patterns).
- a pattern may **match** or **fail**.
- pattern variables are instanciated by the matching.
- a pattern matching involves a test (of the matched data, does it match?) and then a fill (of the transmitted data to sub-patterns)
- MELT patterns can be non-linear: the same pattern variable can appear more than once.

Similarity of patterns in the unix world: `sed regexpr`-s (or in Posix regular expressions in the `regexec` function): Regexpr-s consume strings, and extra matched substrings (thru `regmatch_t` in Posix `regexec`).

MELT patterns inspired by Ocaml’s patterns mostly (and also Wadler’s views, bananas, etc.).
an example of pattern use

(defun makegreen_transform (grdata :tree decl :basic_block bb)
  (debug_msg grdata "makegreen_transform start grdata")
  (debugbasicblock "makegreen_transform bb" bb)
  (eachgimple_in_basicblock
    (bb) (:gimple g)
    (match g
      (? (gimple_assign_cast ?lhs ?rhs)
        (debugtree "makegreen_transform assign cast lhs" lhs)
        ;; etc
      )
      (? (gimple_assign_single
           ?lhs ?(and ?rhs
              (? (tree_var_decl
                   _ ?(cstring_same "stdout") _)))
        (debugtree "makegreen_transform assign stdout lhs" lhs)
        ;; etc
      )
      (? _
        (debuggimple "makegreen_transform unmatched g" g)
      )))
)
**match expressions**

**syntax**

Syntax: \((\text{match } \epsilon \ \chi_1 \ \ldots \ \chi_n)\) where \(\epsilon\) is the matched sub-expression and the \(\chi_i\) are **match clauses**

Each match clause \(\chi_i\) starts with a pattern \(\pi_i\) followed by one or more sub-expressions: \(\chi_i \equiv (\pi_i \ \epsilon_{i,1} \ \ldots \ \epsilon_{i,n_i} \ \epsilon'_i)\)

Patterns (usually starting with \(?\) - a question mark) may in particular be

- constants,
- **pattern variables** like \(?x\) or **jokers** like \(?_\)
- composite patterns with matchers, e.g. \(?(@(\text{gimple\_assign\_cast} \ ?\text{lhs} \ ?\text{rhs})\)
- patterns made with “patmacros” like
  \(?(@(\text{and} \ ?(@(\text{gimple\_cond\_less} \ ?\text{lhs} \ ?\text{rhs}) \ ?(@(\text{gimple\_cond\_with\_edges} \ ?\text{iftrue} \ ?\text{iffalse}))\)
- etc...
Pattern matching

**match expressions**

**informal semantics**

First, the matched $\epsilon$ is evaluated to some matched thing $\mu$ (a value or a stuff).

The matched thing $\mu$ is matched with each pattern $\pi_1 \ldots \pi_n$ in turn. When a matching pattern $\pi_i$ is found, its pattern variables bound by the match are visible in the clause\(^\text{57}\), and the sub-expressions $\epsilon_{i,1} \ldots \epsilon_{i,n_i}$ are evaluated for their side effects and the last sub-expression $\epsilon_i'$ gives the result of the entire match.

Usually the sub-expressions in a match clause contains the pattern variables.

If no clause matches, the entire *match* expression gives a cleared result (nil value, 0 long stuff, $(\text{gimple})0$ stuff, ...).

The resulting ctype of the *match* expression is the common ctype of the $\epsilon_i'$ (or else :\text{void})

\(^{57}\)E.g. if $?x$ appears inside $\pi_2$ and $\epsilon'_2 \equiv x$, the result of the entire match expression is the thing matching $?x$ when $\mu$ matches $\pi_2$.
expressions $\epsilon$ (e.g. constant literals) are (degenerated) patterns. They match the matched data $\mu$ iff $\epsilon == \mu$ (for the C sense of $==$).

The joker noted $？_\nu$ matches every thing and never fails.

a pattern variable $？\nu$ matches $\mu$ if it was unset (by a previous [sub-]matching of the same $？\nu$). In addition, it is then set to $\mu$. If the pattern variable was previously set, it is tested for identity with $==$ in the C sense.

most patterns are matcher patterns $？(m \ \epsilon_1 \ldots \ \epsilon_n \ \pi_1 \ldots \ \pi_p)$ where the $n \geq 0$ expressions $\epsilon_i$ are input parameters to the matcher $m$ and the $\pi_j$ sub-patterns are passed extracted data.

instance patterns are $？(instance \ \kappa : \Phi_1 \ \pi_1 \ldots : \Phi_n \ \pi_n)$; matched $\mu$ is an object of [a sub-] class $\kappa$ whose field $\Phi_i$ matches sub-pattern $\pi_j$.

conjunctive patterns are $？(and \ \pi_1 \ldots \ \pi_n)$ and they match $\mu$ iff every $\pi_i$ in sequence matches $\mu$

disjunctive patterns are $？(or \ \pi_1 \ldots \ \pi_n)$ and they match $\mu$ if one of the $\pi_i$ matches $\mu$
defining C-matchers with defcmatcher

A c-matcher gives C code template for testing the matched data and for filling the extracted sub-data.

```c
;; match a gimple cast assign
(defcmatcher gimple_assign_cast
  (:gimple ga) ;match
  (:tree lhs ;left hand side
   :tree rhs ;first right operand
  ) ;outs
  gimpascs
  ;; test expansion
  #{/*$gimpascs test*/($ga && gimple_assign_cast_p ($ga))}#
  ;; fill expansion
  #{/*$gimpascs fill*/
    $lhs = gimple_assign_lhs($ga);
    $rhs = gimple_assign_rhs1($ga);
  }#
)
```

When defining your c-matcher, be cautious: the matched data μ can be cleared!
You can define your fun-matcher with a MELT function, which returns primarily nil on failure and a non-nil value\textsuperscript{58} on success. When the match succeeds, the filled data is given thru secondary results.

```
(defun matchbiggereven (fmat :long m :long n)
  (if (==i (%iraw m 2) 0)
    (if (>i m n)
      (let ( (:long h (/iraw m 2)) )
        (return fmat h) ;; succeed, gives h )))
    ;; fails
    (return))
)
```

Then ?(matchbiggereven 5 ?n) would match some long stuff $m$, if it is even $m = 2n$ and $m > 5$ and bind $n$ to $n$.

\textsuperscript{58}E.g. :true or here the matcher fmat
Pattern matching

A first version of the pattern matching translator is ad-hoc (but try to share sub-pattern matching).

A second version makes a graph of tests

```lisp
(defun testnameofsymbol (symb f g)
  (match symb
    (? (instance class_symbol :named_name ?synam)
       (f synam))
    (? (instance class_container :container_value ?cval)
       (g cval))))
```

gets translated into the following graph

59 almost done, very experimental
Your passes in MELT

- define your own MELT modes (using `class_melt_mode` and `install_melt_mode`).
- define and install your GCC passes coded in MELT
- if needed, define missing glue (primitives, c-matchers, c-iterators, ...)
- use pattern matching extensively
- MELT values can be shared between GCC passes coded in MELT (thru instances defined by `definstance`, by closing values, etc.). For instance, you could have a first pass filling some MELT object with a basic block hash map associating MELT closures to `basic_block`s, and a later pass choosing some of the basic block and applying the associated closure.

There are lots of existing MELT names gluing many GCC API names. Look into documentation or `grep` the `*.melt` code.
useful MELT features for adding GCC passes

```
(defclass class_gcc_pass
  :predef CLASS_GCC_PASS
  :super class_named
  :fields (gccpass_gate ; closure for gate
             gccpass_exec ; closure for execution
             gccpass_data ; extra data
             gccpass_properties_required
             gccpass_properties_provided
             gccpass_properties_destroyed
             gccpass_todo_flags_start
             gccpass_todo_flags_finish)
)
(defclass class_gcc_gimple_pass
  :predef CLASS_GCC_GIMPLE_PASS
  :super class_gcc_pass)
(defclass class_gcc_rtl_pass
  :predef CLASS_GCC_RTL_PASS
  :super class_gcc_pass)
(defclass class_gcc_simple_ipa_pass
  :predef CLASS_GCC_SIMPLE_IPA_PASS
  :super class_gcc_pass)
```

and the `install_gcc_pass` primitive.
debugging help

You usually don’t want to debug the MELT generated C code under *gdb*.

The `-fmelt-debug` program argument to `gcc-melt` gives lot of (uniquely numbered) debugging output messages.

The `-fmelt-debugskip=1234` program argument skips the first 1234 debugging messages.

To get your own debugging messages

- Debug display a value with `debug_msg`:
  
  ```(debug_msg curval "this is curval")```

- Debug display a stuff, e.g. a raw `gimple` or `tree`, with `debuggimple`, `debugtree` primitives, etc...

Such debug messages also show their source location in MELT source file.

---

60 However, your MELT module knows about the MELT source code location because MELT generates lots of `#line` directives, which can be disabled.

61 The debugged stuff is the first argument, the message string the second one!
The `assert_msg` syntax checks an assertion at runtime:

\[(\text{assert_msg } \mu \; \tau) \Rightarrow \text{When ENABLE_CHECKING, if the test } \tau \text{ is false, display the message } \mu, \text{ the MELT call stack, and fatal error.}\]

```lisp
;; -*- lisp -*- ; public domain file assertex.melt
(defun chokeme (x :long t)
  (debug_msg x "chokeme got x")
  (assert_msg "choke me got a zero t" t))
(defun dobad (x y)
  (debug_msg y "has x")
  (if (= x y) (chokeme x 0)))
(dobad 'b (multiple_nth (tuple 'a 'b 'c) 1))```
debugging aid helps!

shell run 9

gcc-melt -fmelt-mode=runfile -fmelt-arg=assertex.melt -fmelt-debug -c empty.c

⇒

#4:<RUNFILE_DOCMD @warmelt-outobj.melt:4002> warmelt-outobj.melt:4030:/ clear
#5:_ melt-runtime.c:9546 <do_initial_mode before apply>
#6:_ melt-runtime.c:9760 <load_initial_melt_modules before do_initial_mode> .

@assertex.melt:42: start initialize_module_meltdata_assertex iniframp__=0x7fff03afa120

!!!*****####1743#^6:assertex.melt:6:has x !1: ‘B|CLASS_SYMBOL/bad961

!!!*****####1744#^7:assertex.melt:3:chokeme got x !1: ‘B|CLASS_SYMBOL/bad961

cc1: error: MELT fatal failure from assertex.melt:4 [MELT built Oct 19 2010]

SHORT BACKTRACE[#1744] MELT fatal failure;
#1:<CHOKEME @assertex.melt:2> assertex.melt:4:/ cond.else
#2:<DOBAD @assertex.melt:5> assertex.melt:7:/ cond
#3:_ assertex.melt:8:/ apply
#4:_ melt-runtime.c:7155:meltgc_make_load_melt_module before calling module asserte
#5:<RUNFILE_DOCMD @warmelt-outobj.melt:4002> warmelt-outobj.melt:4030:/ clear
#6:_ melt-runtime.c:9546 <do_initial_mode before apply>
#7:_ melt-runtime.c:9760 <load_initial_melt_modules before do_initial_mode> .

cc1: error: MELT failure with loaded module #1: /usr/local/libexec/gcc-melt/gcc/x86_64-unknown-linux-gnu/4.6.0/melt-module//warmelt-first.so
cc1: error: MELT failure with loaded module #2: /usr/local/libexec/gcc-melt/gcc/x86_64-unknown-linux-gnu/4.6.0/melt-module//warmelt-base.so
cc1: error: MELT failure with loaded module #3: /usr/local/libexec/gcc-melt/gcc/x86_64-unknown-linux-gnu/4.6.0/melt-module//warmelt-debug.so
cc1: error: MELT failure with loaded module #4: /usr/local/libexec/gcc-melt/gcc/x86_64-unknown-linux-gnu/4.6.0/melt-module//warmelt-macro.so

(partial output shown)
## Table of Contents

1. Introduction

2. Basic MELT usage and features
   - running MELT
   - MELT language syntax
   - MELT data (*things = values + stuff*)
   - connecting GCC to MELT

3. Pattern matching

4. Coding passes in MELT

5. Conclusion and future work
Try to use MELT by yourself

- most of the MELT reference documentation\(^{62}\) is generated\(^{63}\) from MELT source code (using :doc annotations)
- Learn more about MELT by looking into generated documentation... make pdf in the MELT branch

Generating documentation from code brings “interesting” issues\(^{64}\). Perhaps MELT reference documentation (mostly generated) will be GPL-ed?

- ask me for help if needed in english on gcc@gcc.gnu.org, in french on gcc-melt-french@googlegroups.com
- patch the MELT branch (e.g. if some primitives are missing)

---

\(^{62}\) Which is very incomplete on october 2010
\(^{63}\) In file gcc/meltgendoc.texi in the build tree
\(^{64}\) See http://gcc.gnu.org/ml/gcc/2010-10/msg00242.html
Who uses MELT?

- (my former intern) Jérémie Salvucci jeremie.salvucci@free.fr:
  1. added several needed glues (cmatchers, primitives)
  2. coded a translator\(^{65}\) in MELT from Gimple to low-level C for the free Frama-C http://frama-c.com/ static analyzer\(^{66}\)
  3. coded (with me) the gengtype patches for plugins\(^{67}\)

- Marie Krumpe (intern of Emmanuel Chailloux, LIP6 [Paris 6 Univ.]) explored generation of low-level C++ code from Gimple for the Cadna http://www.lip6.fr/cadna free software library\(^{68}\)

- Alexandre Pelissy (Mandriva and PhD student at Univ.Tours) - with Pierre Vittet are starting to use MELT for analysis of the linux kernel code

- I am continuing to enhance MELT and will use for generation of OpenCL code www.opengpu.net\(^{69}\)

- you are welcome to use it!

---

\(^{65}\)Partially working
\(^{66}\)Developed in Ocaml by my CEA collegues and INRIA people; free software LGPL licensed
\(^{67}\)Work being submitted to the trunk in october 2010
\(^{68}\)estimating accuracy of IEEE 754 floating computations
\(^{69}\)Perhaps using Graphite-OpenCL?
future work

- bug corrections
- (in progress) improving the translation of `match` expressions
- minor MELT language and runtime improvements
e.g. simpler program arguments
- good LTO support in MELT
- more MELT language features (rewriting MELT macro-s à la Scheme, patterns in `let`, . . .)
- memoizing method lookup
- adding more c-types when needed
- real GCC passes coded in MELT
technical points

LTO $\Rightarrow$ serialization of **MELT** values
- sharing **MELT** values between several *.o*
- serialization of closures (but **MELT** closures know the C name of their routines!)
- ...

Take profit of better **gengtype**
(generate some runtime code from **gtype.state**?)
MELT can be used to experiment new middle-end passes in GCC. It can also be used for specific GCC extensions. For library- or software-specific MELT extensions to GCC, the best experts are from that software community.

- GTK would need several MELT extensions to:
  1. type-check `g_object_set`
  2. check coding conventions
  3. etc
- The Linux kernel will benefit from GCC extensions coded in MELT.
- Major free software\(^{70}\) compiled with GCC could benefit from MELT.

Long term dream: perhaps pushing MELT into the trunk, e.g. as a help to developers? MELT as a tree-browser on steroids?\(^ {71}\) Or maybe, MELT should stay as a GCC plugin example?

\(^{70}\)Intuitively, any large enough \(\geq 1\)MLOC free software could develop its own GCC extensions in MELT for its own use, and using MELT should be less hard than coding plugins in C!

\(^{71}\)The `gcc/tree-browser.c` is useful for GCC developers. Perhaps MELT may be useful for them also?
To help me...

- compile and use MELT
- explain me LTO streaming
- explain me how to make test-cases (dejagnu scripts for MELT?)
- explain me why MELT didn’t exist before? GCC has a lot of code generators already! (why not in the middle-end?)
- suggest better building scheme for MELT (Makefile-s)?

Thanks! Questions?