GCC Hacks

Open Source Conference 北海道
さっぽろ産業振興センター
産業振興棟 2F ルーム C 16:00 ～
2008.06.28（土）
講師:若槻俊宏（@alohakun）

こんにちは

こんなニッチな
テーマのセミナー
に来てくださり

ありがとう
ございます

いきなりで
すいませんが
・・・

一応の想定客層（予告版より）

- プログラミング言語 C がそれなりにわかる！
  - Lisp（リスト操作）がちょっとわかるとモアベター
- オープンソースソフトウェア，あるいは
  自由なソフトウェア自体に興味がある！
  - 残念ながら，お金の匂いはあまりしません
- ネイティブコンパイラを書いてみたい！
  - 機械語は男の浪漫
- GCC を極めて起業する！
  - 素晴らしいですが，無理だと思います

とかいろいろ書いてはみたものの

やっぱり
客層が全然
絞り込めません
でした！

Lisp（笑）

というわけで
いっそのこと...

でもみんなさすがにCぐらいわかるよね大人だもん

今日はなぜか会場に
うちの未来嫁
（α版）
が来てることですし
［dn］

出たよ惚気

某国立H大学経済学部卒

普通の文系 0L にもわかる コンパイラ入門！

ゆるふわ愛され GCC Hacks

※ GNU Free Documentation License なので
いちおう節度ある範囲の改変は OK なはず

http://ja.wikipedia.org/wiki/%E7%94%BB%E5%83%8F:St-ignucius.jpg

WHAT'S YURUFUWA ?

空前のゆるふわブーム

http://kakutani.com/20080620.html#p02

Photo : http://gihyo.jp/assets/images/news/report/01/rubykaigi2008/0002/R0012105.jpg


YURUFUWA
http://ja.wikipedia.org/wiki/%E7%94%BB%E5%83%8F:St-ignucius.jpg

ふつうの
ネイティブコンパイラ開発

GCCで実現する
ゆるふわドンピシャ
愛されコンパイラ開発


いろいろ無理が
あるだろＪＫ
※ まぁまぁ，せっかくですんで
やれるだけやってみましょうよ
http://ja.wikipedia.org/wiki/%E7%94%BB%E5%83%8F:St-ignucius.jpg

ちなみに

スイーツ（笑）にもわかるソフトウェアリリース段階

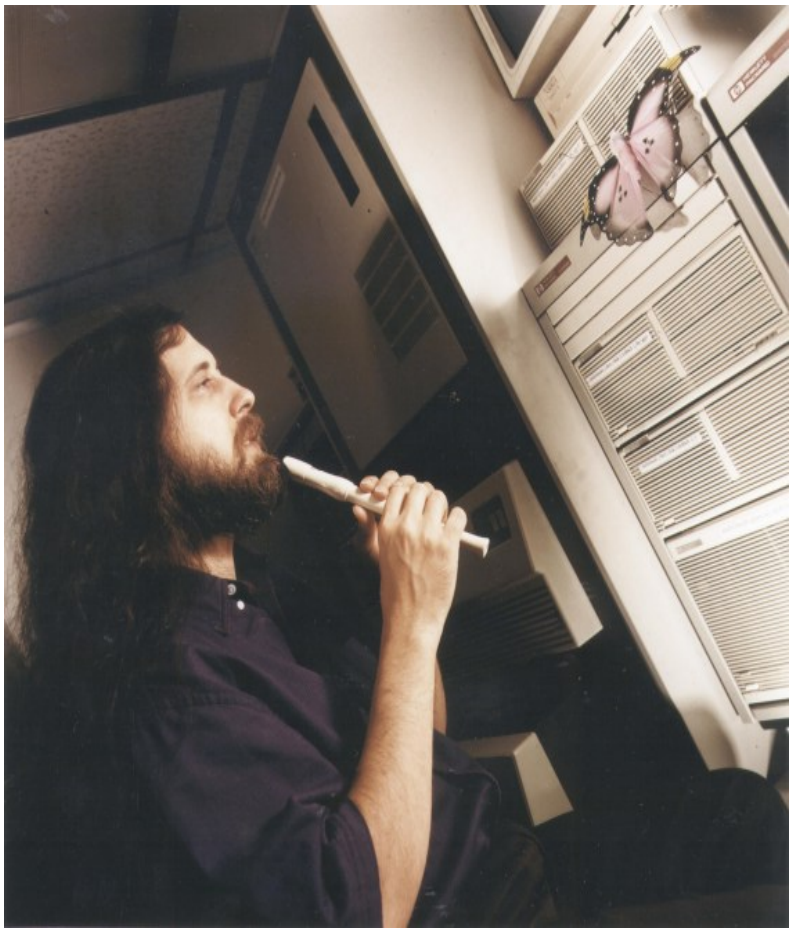
- α版 : 開発初期バージョン
  - 付き合い初めですね．まだいろいろと不具合がボロボロ出てくる段階
- β版 :テスト配布バージョン
  - ご両親に挨拶しにいく段階ですね（Web 2.0 的には永遠のβ版が今風）
- RC（Release Candidate）: 実環境テストバージョン
  - 同棲中ですね．試験運用中です
- 正式版 1.0
  - 入籍・披露宴も終わり，いよいよ結婚生活開始です
- 嫁 2.0（笑）
  - お子様が生まれたりとか，いろいろ（投げやり）
  - 以下，毎年バージョン番号が上がるたびに安定性が増していきます
  - 突然不安定になったり，開発中止になる場合もあるので油断はできない

ここで話者について(いまさら)

- 普通のゆるふわ愛されガチムチ男子大学院生
  - 北海道大学大学院情報科学研究科 博士後期課程
    - 複合情報学専攻 情報システム設計学研究室
  - プログラムの自動探索(生成)を研究してます
- GCC は，実システムのサーベイ兼趣味
  - id:w_o さんと id:shinichiro_h さんの影響大
    - 自分にとってネ申のような存在
  - 詳しくは GCC wikia に
    - http://tinyurl.com/5fe4ek

そもそも GCC って何なの？

- GNU Compiler Collection（フレームワーク）
  - 当初は GNU C Compiler
- 開発リーダー
  - Richard M. Stallman
    - 聖イグヌチウス
  - 1986年 Free Software Foundation (FSF) 設立
    gcc 開発開始
  - 自由なソフトウェア,
    自由な社会の象徴！

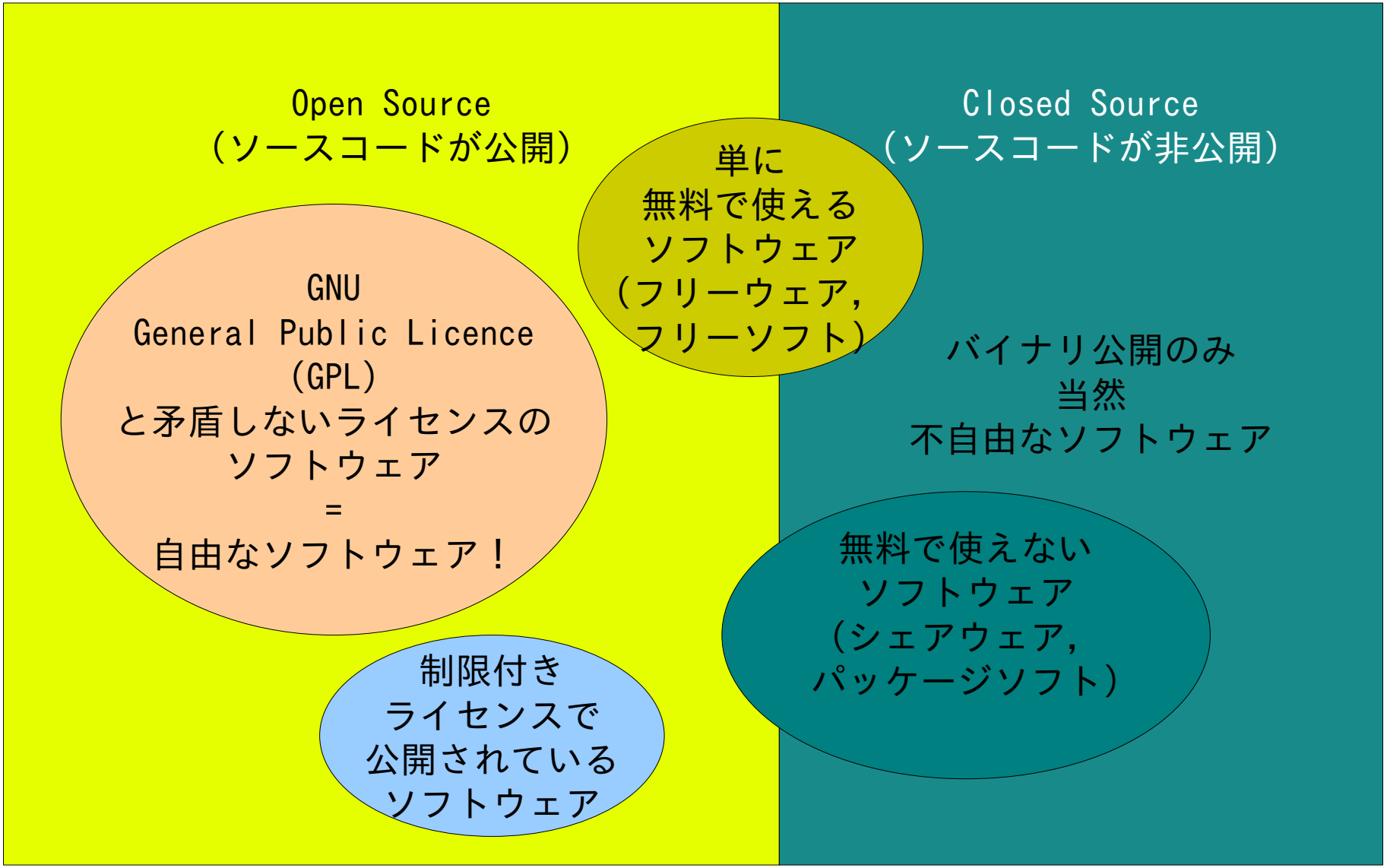


http://www.stallman.org/ より

現在の開発体制

- Redhat や Apple などが中心的に開発
  - チーフメンテナー 13人(伽藍モデル)
    - http://journal.mycom.co.jp/articles/2004/12/22/gcc/001.html
    - gcc/MAINTAINERS
  - そのうち 8 人が Redhat (↓1997 年当時)
    - http://www.icot.or.jp/FTS/REPORTS/H12-reports/H1303-AITEC-Report3/AITEC0103-R3-html/AITEC0103-R3-ch3_4_5.htm
    - 各部分の commit が認められるメンテナー 49 人
    - 上位メンテナーの承認が必要なメンテナー 64 人
  - cf) Linux kernel 約 3000 人 (バザールモデル)

自由なソフトウェアって何なの ？



ソースコードって何なの？

- プログラムはテキストファイルで表現される
- テキストファイル
  - 文字の並び（改行とかも，全部文字）
    - 数字列（バイト，8 bit（bit = 0/1，8桁の2進数））
    - 文字と数字の対応にはいろいろある（コードポイント）
      - ASCII とか JIS X 0208 とか Unicode とか
    - 数字の表現にもいろいろある（エンコーディング）
      - ASCII はそのまま 1 文字 1 バイト
      - JIS X 0208 は euc jp とか shift jis とか
      - Unicode は utf-8/16/32 とか
- 要するに，メモ帳で開けるファイル


お前の話は
つまらん
※ スイーツ（笑）にもわかるコンパイラセミナーです
http://ja.wikipedia.org/wiki/%E7%94%BB%E5%83%8F:St-ignucius.jpg

話をコンパイラに
戻すと．．．

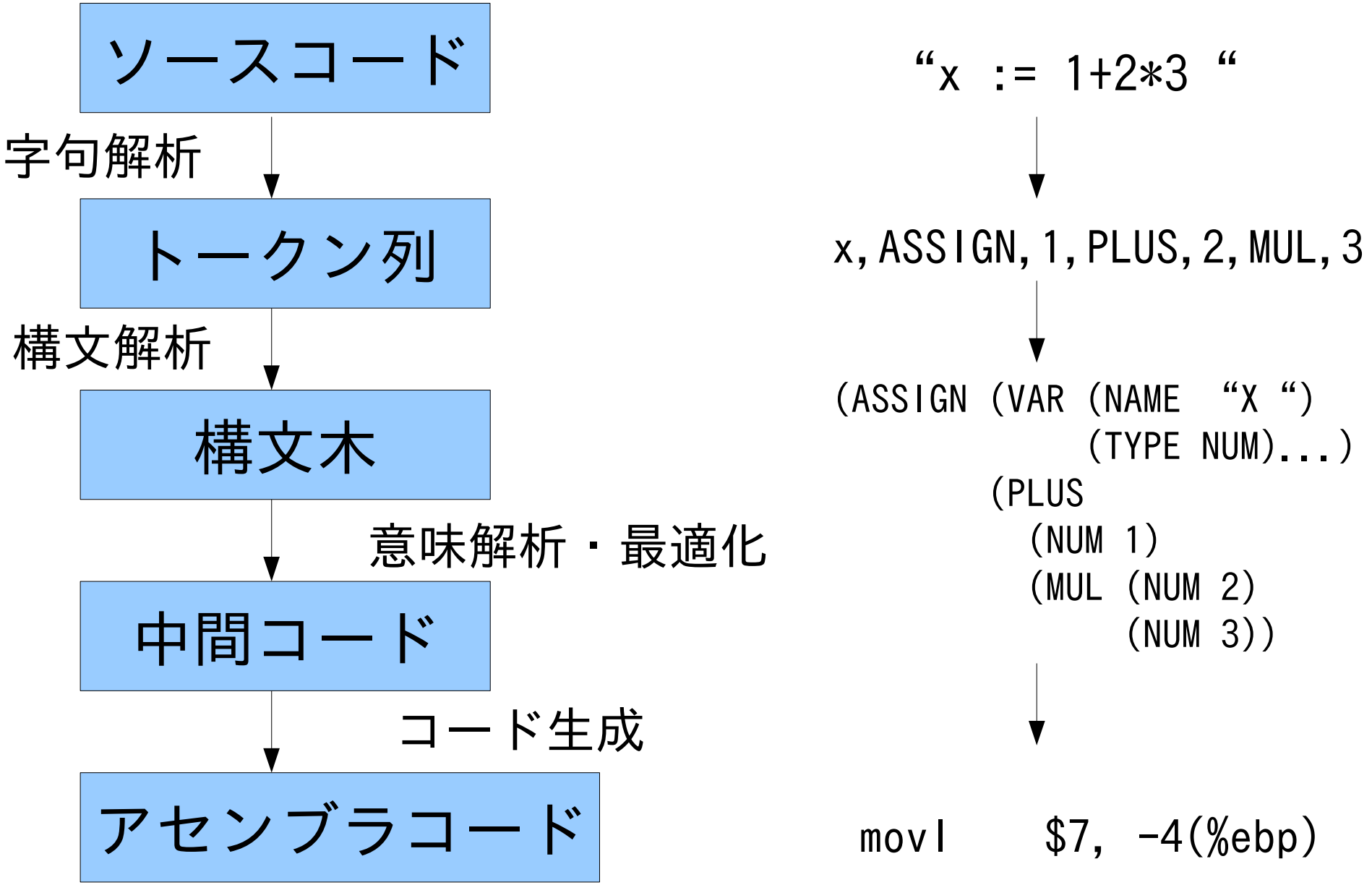
コンパイラって何なの？

- 翻訳系（システム）
  - コンパイラ本体意外にも，アセンブラとかリンカとかいろいろ
- テキストファイルをバイナリファイルに変換
  - 人間が読めるテキストファイルを，機械が読めるバイナリファイルに変換する （実行可能ファイル）
  - アセンブリプログラム
    - 一応人間も読み書きできる（煩雑で難しいけど）
  - バイナリファイル
    - 人間の目には，0と1の羅列にしか見えない
    - ちゃんとしたフォーマットに則ってる
      - a.out とか ELF とか COFF とか Mach-O とか

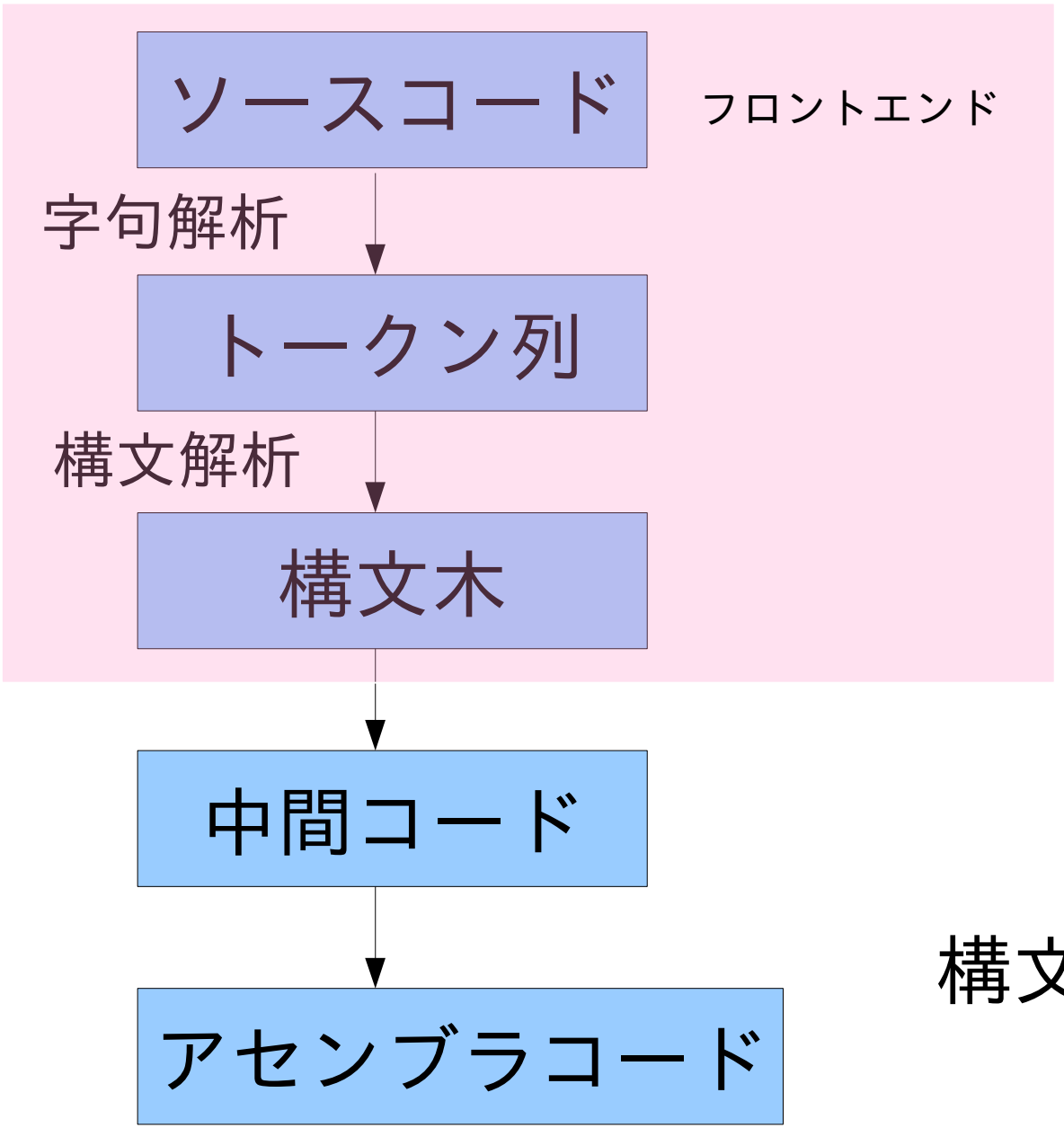
プログラミング言語は星の数ほどあるけど

- コンパイラの構成自体はほとんど変わらない
- フレームワーク化できる
  - 再利用性
  - 言語間の連携
  - 一つ一つ手作りするよりも，いろいろお得
- 言語非依存の設計が重要
  - 何が言語に依存/非依存なのか?
    - 実はかなり難しい問題（アカデミック！）
  - ある程度の割り切りが必要（GCC は，基本的に命令型の静的コンパイル型言語を対象としている）

コンパイラの大まかな構成



フロントエンド部分



“x := 1+2*3 “

↓

x,ASSIGN,1,PLUS,2,MUL,3

↓

```
(ASSIGN (VAR (NAME  “X “)
             (TYPE NUM)...)
        (PLUS
          (NUM 1)
          (MUL (NUM 2)
               (NUM 3))
```

構文木作るところまで

他のコンパイラフレームワーク

- COINS
  - http://www.coins-project.org/
  - 並列化コンパイラ向け共通インフラストラクチャ
  - SCK (COINS の流れ)
    - http://www14.plala.or.jp/gazico/sck/
    - GNU Emacs 上のマルチソース、マルチターゲットなコンパイラ作成支援環境(IPA 未踏プロジェクト)
- LLVM
  - http://llvm.org/
  - コンパイラというよりも，コード生成と実行全体
  - より高速なコンパイル，BSD ライセンス，モダンな最適化

最初からハックを前提とした作り

- アカデミックで綺麗な作りになっている
  - GCC を参考にしている（後発の強み）
- 中間表現がダンプ & 読み込み可能（COINS）
    - 高水準中間表現 (HIR: High level intermediate representation)
    - 低水準中間表現 (LIR: Low level intermediate representation)
  - GCC は意図的に不可能（フリーライド防止）
    - http://journal.mycom.co.jp/articles/2004/12/22/gcc/003.html
- ドキュメントもちゃんと存在（日本語も）
  - 厳密な表示的意味論が存在している

では，なぜ GCC ？

- 多くのプログラミング言語に対応 & 相互運用可能
  - C, C++, Objective-C, Fortran, Java, Ada, ...
- 豊富なターゲット CPU
  - x86, PPC, SPARC, Alpha, SH, ARM, MIPS
- クロスプラットフォーム
  - Windows, Linux, BSD, Mac OS, Plan 9 (?), ...
- それなりの最適化
  - さすがに icc 等，特定環境専用コンパイラには負ける
- 完成度と長年 FOSS 界を支えてきた実績
- 現在も活発に開発が続いている


まえおき
ながいよ
※ 見た目はゆるふわでも，辛口
http://ja.wikipedia.org/wiki/%E7%94%BB%E5%83%8F:St-ignucius.jpg

やっぱり
無理でした！
ごめんなさい

人間だもの

ちょっと
ここからは
普通の OL さん
には難しいかも

てか，まだその一発ネタ引っ張ってたの．．．

このセミナーの目的（いまさら）

- GCC のフロントエンドを書けるようになる
  - 自分独自のプログラミング言語のコンパイラを書けるようになる
  - コンパイラの一般的な技術解説は行わない
  - 字句解析/構文解析などは別の専門書を参考にせよ
- 構文解析時のセマンティックアクション
  - どうやって GCC の構文木を作れば良いのか ？
  - ソースコードから構文木を作るやり方さえわかればコンパイラが作れる！

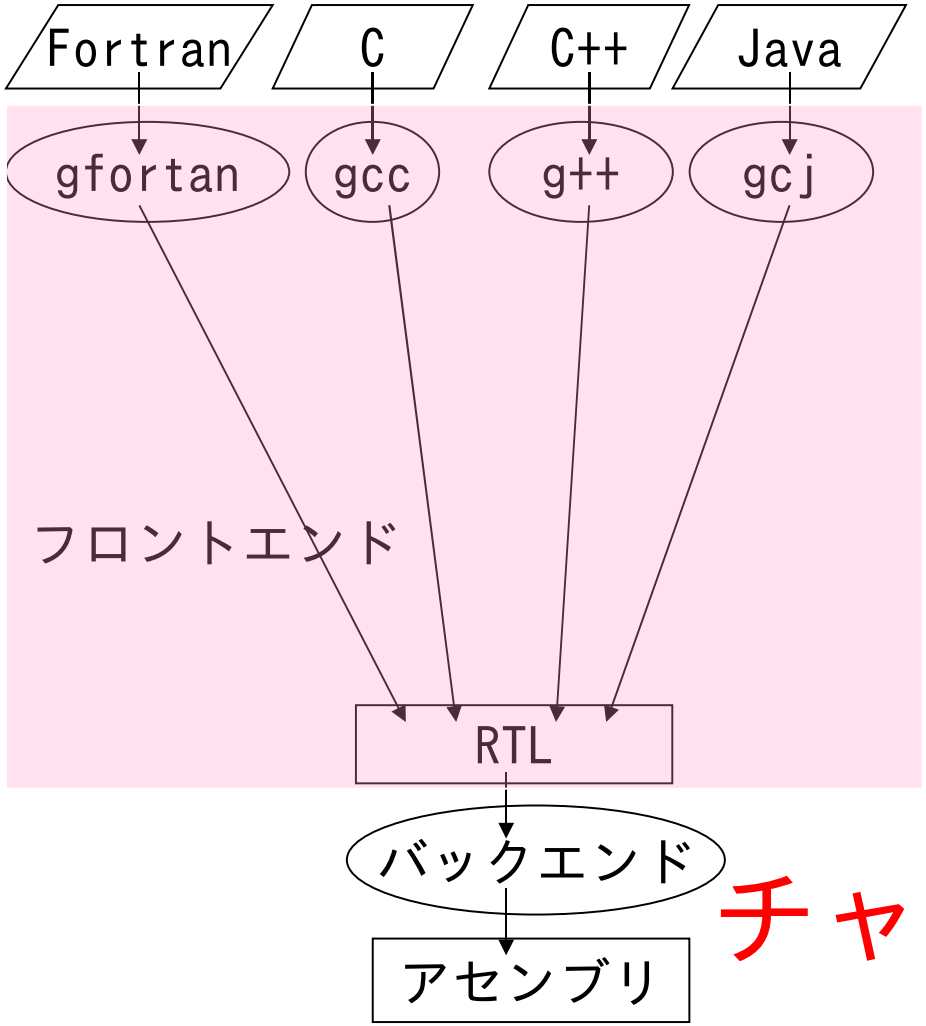
要するに

- 川合さんの 30 日 OS 自作入門みたいなノリ
  - 私だけのこだわりコンパイラ作り（笑）
- ネイティブコンパイラが作れる
  - = 自分好みのプログラミング言語を作れる（かもしれない）
  - 学習用，研究用，もしかしたら仕事にも（なるかもしれない）
- うちの彼女の名（迷？）言集 （メッセンジャーより勝手に抜粋）
  - つまりマイ筆記用具なんでしょ？
  - 改造好きには堪らないね
  - 俺言語が作れるって，何でもできるっこと ？
  - 彼女作ってメイド服着せたりチャイナ着せたりナース着せたりみたいな？
  - もうやり放題って感じっすか！概念的に言えば（´＾ω＾｀）

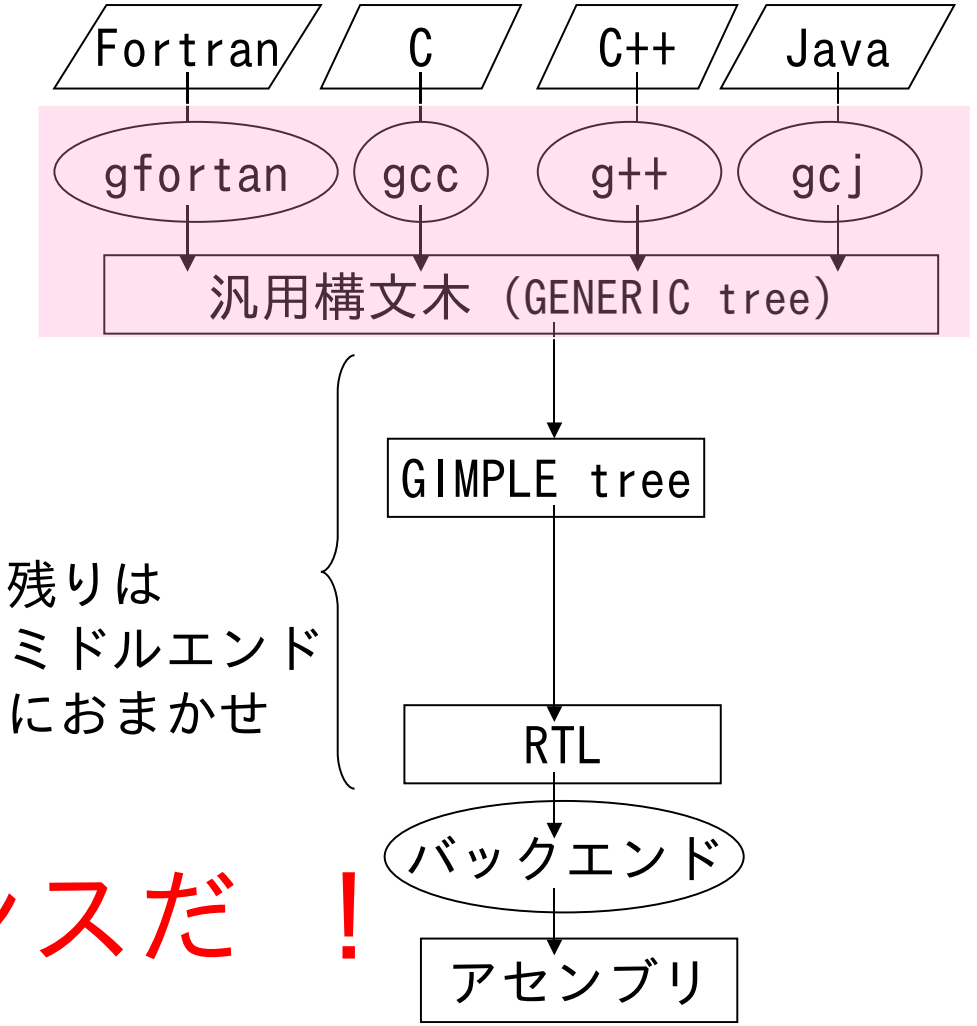
※ そうです（たぶん）... あなたが神です

実は以前よりもフロントエンドは書きやすくなってる

- GCC 3.x 以前



- GCC 4.x 以降

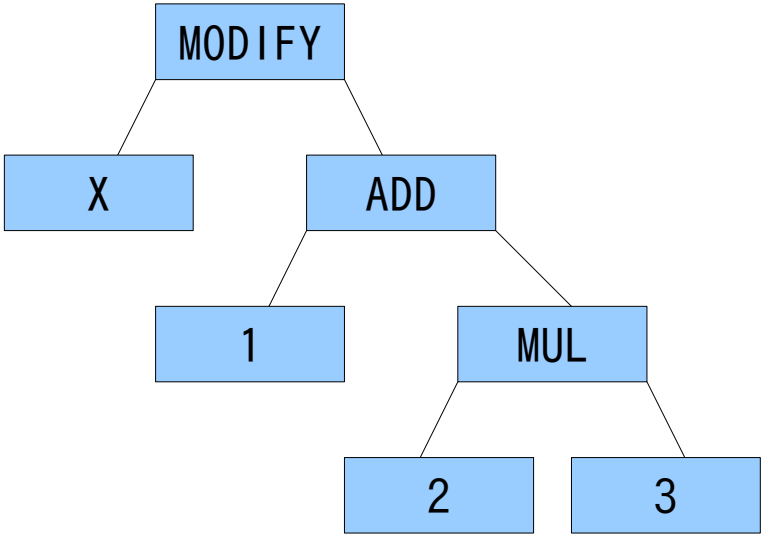


チャンスだ！

要するに

- 文字列のパターンと，構文木を対応させる
  - ここらへんを真面目に数学的に定式化すると，表示的意味論とかになる（Scheme とか）
- 入力文字列に応じて，構文木作る
- あとは GCC におまかせ ♪

X := 1 + 2 * 3 ⟶



しかし問題が

- GCC の開発スピードは非常に速い
  - あっという間にドキュメントが古くなる
- GCC 4.x 以降の資料はほとんど存在しな
  - 事実上ソースがドキュメント
  - tree まわりの API 資料はほとんど無い...
    - GCC internals も空ページ
- 歴史があるぶん，複雑で巨大
  - 本質と非本質の切り分けが. . .
  - ビルドを通すだけで大変

ミドルエンド以降の資料は比較的存在

- 本家 GCC Internals
  - http://gcc.gnu.org/onlinedocs/gccint/
- Diego Novillo, *GCC Internals.*, CGO 2007 Workshops and Tutorials (International Symposium on Code Generation and Optimization), March 2007
  - http://www.airs.com/dnovillo/Papers/cgo2007-gcc-internals.pdf
- The GCC Architecture Documents
  - http://www.cfdvs.iitb.ac.in/~amv/gcc-int-docs/

サンプルフロントエンドが必要

- とりあえず，何をすれば良いのか ？
  - 最小例が欲しい（あとはみんな大人なんだから...）
- toy language frontend : リンク切れ
- hello-world frontend : obsolete
  - http://svn.gna.org/viewcvs/gsc/branches/hello-world/
    - 2 年以上前に更新停止 (May 2006)
- 唯一 4.3.0 に追従 : gcc-4.3.0/gcc/treelang
  - 比較的複雑な構造なので初心者には厳しい
    - yacc/lex などの非本質にも惑わされる
  - そもそも GCC 4.4.0 以降では削除されそう

BL sample GCC frontend project

- http://sourceforge.jp/projects/blanguage
- 公開物
  - Minimal frontend
    - ビルドが通る最小構成
    - int main() {return 0;} の構文木を決め打ち
  - FizzBuzz frontend
    - FizzBuzz が通るぐらいの言語機能をサポート
    - GCC ビルトイン関数サポート（puts, printf, ...）
  - コア開発者 : 自分の他に @QLeap（佐藤大介）さん
    - GCC 4.3.0 以降のフロントエンド初期化部分対応

BL is not Boys Love.

- ボーイズラブとは、日本における男性同士の同性愛を題材とした女性向けの小説や漫画のジャンルのことである。10代の少年（特に美少年）同士の間での恋愛を指す言葉であり、大人の男性同士の作品はメンズラブと呼ばれる場合があったが、最近では広い範囲で「女性向けの男性間同性愛」を指す。(ja.wikipedia 調べ)
  - Basic Language ?
  - Buggy Language ?
  - C 未満言語 ?

Minimal frontend

- 最小構成のフロントエンド（250 行程度）
- gcc/bl/
  - Make-lang.in　　: 50 行強
  - bl1.c　　: 150 行強
  - config-lang.in　　: 2 行
  - lang-specs.h　　: 5 行
  - lang.opt　　: 10 行弱（無駄がある）
  - spec.c　　: 30 行弱

Minimal な bl1.c（コアファイル）

- ビルド通すだけ

```
static void insert_block (tree block) {}
static int global_bindings_p (void) {}
static tree pushdecl(tree decl) {}
static tree getdecls (void) {}
tree convert(tree type, tree expr) {}
static tree bl_type_for_size(unsigned precision, int unsignedp) {}
static tree bl_type_for_mode(enum machine_mode mode, int unsignedp) {}
static void bl_finish(void) {}
static unsigned int bl_init_options (unsigned int argc, const char **argv) {}
static int bl_handle_option(size_t scode, const char  *arg, int value) {}
static bool bl_mark_addressable(tree exp) {}
. . .
struct lang_type GTY(()) {
  char dummy;
};
struct lang_decl GTY(()) {
  char dummy;
};
struct language_function GTY(()) {
  char dummy;
};
```

動かすためには

- gcc-4.3.0.tar.bz2 を持ってきて展開
- bl-minimal-4.3.0.tar.bz2 あるいは bl-0.0.1-for-gcc-4.3.0.tar.bz2 を同様に展開
- bl というディレクトリを gcc-4.3.0/gcc/ 以下に
- ビルドディレクトリの中で configure

```
$ ../gcc-4.3.0/configure
 --prefix=/usr/local/build-gcc/build-bl
 --enable-languages=bl --enable-checking=all --
 disable-nls --disable-bootstrap
```

- make && make install

--disable-bootstrap

- 通常 gcc は 3 回ビルド（bootstrap）する
  - Stage1 : ホスト環境の gcc で gcc をビルド
    - 一緒に gcc runtime やライブラリもビルドされる
  - Stage2 : Stage1 gcc で gcc をビルド
  - Stage3 : Stage2 gcc で gcc をビルド
  - Final : Stage2 gcc と Stage3 gcc を比較
- 面倒なので 1 回のビルドで済ませるオプション

FizzBuzz frontend

```
puts   :: String -> Int
printf :: String -> Int
main = fun Int _ -> Int
  fizzbuzz = fun Int n -> Int
    if n % (100 + 1)
      if n % (3 * 5)
        if n % 3
          if n % 5
            _ <- printf(" %d ", n)
            return fizzbuzz(n + 1)
          else
            _ <- printf(" Buzz ")
            return fizzbuzz(n + 1)
          fi
        else
          _ <- printf(" Fizz ")
          return fizzbuzz(n + 1)
        fi
      else
        _ <- printf(" FizzBuzz ")
        return fizzbuzz(n + 1)
      fi
    else
      return puts("")
    fi
  nuf
  return fizzbuzz(1)
nuf
```

- 型宣言
- 関数定義
  - nested functions
- GCC ビルトイン関数
  - puts, printf
- 加減乗除, 剰余
- If 文
- return 文
- ループは末尾再帰で

実行例

```
$ ../bin/gcc fizzbuzz.bl
$ ./a.out
 1  2  Fizz  4  Buzz  Fizz  7  8  Fizz  Buzz  11
  Fizz  13  14  FizzBuzz  16  17  Fizz  19  Buzz  Fizz
   22  23  Fizz  Buzz  26  Fizz  28  29  FizzBuzz  31
  32  Fizz  34  Buzz  Fizz  37  38  Fizz  Buzz  41
  Fizz  43  44  FizzBuzz  46  47  Fizz  49  Buzz  Fizz
   52  53  Fizz  Buzz  56  Fizz  58  59  FizzBuzz  61
  62  Fizz  64  Buzz  Fizz  67  68  Fizz  Buzz  71
  Fizz  73  74  FizzBuzz  76  77  Fizz  79  Buzz  Fizz
   82  83  Fizz  Buzz  86  Fizz  88  89  FizzBuzz  91
  92  Fizz  94  Buzz  Fizz  97  98  Fizz  Buzz
```

本丸（LANGHOOKS_PARSE_FILE）

```
static void bl_parse_file(int debug ATTRIBUTE_UNUSED) {
  tree main_type   = build_function_type(integer_type_node, NULL_TREE);
  tree main_fndecl = build_fn_decl("main", main_type);
  DECL_EXTERNAL(main_fndecl) = false;
  DECL_ARTIFICIAL(main_fndecl) = false;
  TREE_STATIC(main_fndecl) = true;
  TREE_PUBLIC(main_fndecl) = true;
  DECL_CONTEXT(main_fndecl) = NULL_TREE;
  tree stmts = alloc_stmt_list();
  tree  block = build_block(NULL_TREE, NULL_TREE, NULL_TREE, NULL_TREE);
  DECL_INITIAL(main_fndecl) = block;
  DECL_SAVED_TREE (main_fndecl) = build3(BIND_EXPR, void_type_node, BLOCK_VARS(block), stmts, block);
  tree resdecl = build_decl(RESULT_DECL, NULL_TREE, integer_type_node);
  DECL_CONTEXT(resdecl) = main_fndecl;
  DECL_RESULT(main_fndecl) = resdecl;
  tree main_setret = fold_build2(MODIFY_EXPR, TREE_TYPE(main_fndecl), DECL_RESULT(main_fndecl), integer_zero_node);
  TREE_SIDE_EFFECTS(main_setret) = true;
  TREE_USED(main_setret) = true;
  tree main_ret = fold_build1(RETURN_EXPR, void_type_node, main_setret);
  append_to_statement_list(main_ret, &stmts);
  allocate_struct_function(main_fndecl, false);
  dump_function (TDI_original, main_fndecl);
  gimplify_function_tree(main_fndecl);
  dump_function (TDI_generic, main_fndecl);
  cgraph_node(main_fndecl);
  cgraph_finalize_function(main_fndecl, /* nested function ? */false);
  cgraph_finalize_compilation_unit();
  cgraph_optimize();
}
```

http://noplans.org/product/t-shirt/syo.png


一つ一つ
理解しよう
※ 初期化して
GCC の API 呼び出して
構文木作って
ミドルエンド
にブチ込めば
アセンブリが出るッ！
そんだけだッッ！！
http://ja.wikipedia.org/wiki/%E7%94%BB%E5%83%8F:St-ignucius.jpg

第一の壁 : TREE API

- そもそも API で構文木を作るって？
- 例えば，数の並びの表現（リスト）を作る

Lisp : '(1 2 3)

= (cons 1 (cons 2 (cons 3 nil)))

GCC :

```
tree_cons(NULL_TREE, build_int_cst(NULL_TREE, 1),
          tree_cons(NULL_TREE, build_int_cst(NULL_TREE, 2),
                    tree_cons(NULL_TREE, build_int_cst(NULL_TREE, 3),
                    NULL_TREE))))
```

- 煩雑だけど，大して変わらない
  - と思う

例 ： 配列を作る

- C ソース ： int iarr[2] = {1, 2, 3};
- 対応する構文木を作る parse_file() 関数

```
static void bl_parse_file(int debug ATTRIBUTE_UNUSED) {

tree index_type = build_index_type(build_int_cst(NULL_TREE, 2));
tree array_type = build_array_type(integer_type_node, index_type);
tree array_decl = build_decl(VAR_DECL, get_identifier ("iarr"), array_type);
DECL_EXTERNAL(array_decl) = false;
TREE_STATIC(array_decl) = true;
TREE_PUBLIC(array_decl) = true;
TREE_USED(array_decl) = true;
tree init = build_constructor_from_list(
                    array_type,
                    さっきの数のリスト);
DECL_INITIAL(array_decl) = init;
rest_of_decl_compilation(array_decl, /* TOP LEVEL ? */0, /* AT END ? */0);

}
```

実際に試してみる

- FizzBuzz frontend の bl/bl1.c bl_parse_file() {} の中身を置き換える
  - Minimal frontend だとちゃんと初期化されてないので SEGV る. 注意
- make && make install
- /file/to/install/gcc -S test.bl
  - 何食わせても同じ
    - ただし空ファイルだと : Input file null.bl is empty
  - 当然リンクはできないので, アセンブリ出すだけ

なんかこういうアセンブリが出る

```
        .file   "hello.bl"
.globl iarr
        .data
        .align 4
        .type   iarr, @object
        .size   iarr, 12
iarr:
        .long   1
        .long   2
        .long   3
        .ident  "GCC: (GNU) 4.3.0"
        .section        .note.GNU-stack,"",@progbits
```

コラム : GCC-4.3.0 からの変更点

- GMP/MPFR が必要になった
  - Makelang.in に $(GMPLIBS) が必要になった
- ビルトイン型と関数の初期化が必要になった
  - Minimal で puts や printf を使うと SEGV
  - ちょっと複雑なことをやっても SEGV
  - (少なくとも) 4.2.1 までは何もしなくても使えた
- FizzBuzz frontend は初期化をちゃんとやってるので，これを改造すれば良い
  - ほぼ全てQLeapさんの貢献
  - bl_parse_file() を，独自のパース関数に差し替える

main 関数の構文木を作る

- 関数の型を作る
  - 要するに public static int main();
  - ソースコード内で実体が定義されてる（非 extern）
  - 人工的にコンパイラによって生成された関数ではない
  - 関数内関数では無い　（外部コンテキストを持たない）

```
static void bl_parse_file(int debug) {
  tree main_type   = build_function_type(integer_type_node, NULL_TREE);
  tree main_fndecl = build_fn_decl(”main”, main_type);
  DECL_EXTERNAL(main_fndecl) = false;
  DECL_ARTIFICIAL(main_fndecl) = false;
  TREE_STATIC(main_fndecl) = true;
  TREE_PUBLIC(main_fndecl) = true;
  DECL_CONTEXT(main_fndecl) = NULL_TREE;
```

main 関数の構文木を作る

- main() の中身（式のリスト）を作って初期化
  - main_fndecl にブロック（スコープの構文木）を BIND
  - 構文木を作って，他の構文木に突っ込むってのが基本
  - 取り合えず NULL_TREE で初期化しておいて，後から段階的に構文木を作りつつ，突っ込んでいくことになる
- 今回は {return 0;} のみ （決め打ち）

```
tree stmts = alloc_stmt_list();
tree  block = build_block(NULL_TREE,  /* variables */
                          NULL_TREE,  /* subblocks */
                          NULL_TREE,  /* supercontext */
                          NULL_TREE); /* next same level block */
DECL_INITIAL(main_fndecl) = block;
DECL_SAVED_TREE (main_fndecl) = build3(BIND_EXPR, void_type_node,
 BLOCK_VARS(block), stmts, block);
```

main 関数の構文木を作る

- {return 0;}
- Pascal の result 変数みたいなもの
- 返したい値を RESULT_DECL 宣言した変数に代入する
  - つまり {result := 0;} で {return 0;} に相当する

```
tree resdecl = build_decl(RESULT_DECL, NULL_TREE, integer_type_node);
DECL_CONTEXT(resdecl) = main_fndecl;
DECL_RESULT(main_fndecl) = resdecl;
tree main_setret = fold_build2(MODIFY_EXPR, TREE_TYPE(main_fndecl),
 DECL_RESULT(main_fndecl), integer_zero_node);
TREE_SIDE_EFFECTS(main_setret) = true;
TREE_USED(main_setret) = true;
tree main_ret = fold_build1(RETURN_EXPR, void_type_node, main_setret);
append_to_statement_list(main_ret, &stmts);
```

main 関数の構文木を作る

- しあげ
- 関数を作って，ミドルエンドに渡す
  - もしネストした関数を許す場合，cgraph_node() を再帰的に摘要して，全ての関数を finalize する
  - 詳しくは BL フロントエンドを

```
  allocate_struct_function(main_fndecl, false);
  dump_function (TDI_original, main_fndecl); // デバッグダンプ
  gimplify_function_tree(main_fndecl);       // GENERIC から GIMPLE に変換
  dump_function (TDI_generic, main_fndecl);  // デバッグダンプ
  cgraph_node(main_fndecl);                  // call グラフ
  cgraph_finalize_function(main_fndecl, /* nested function ? */false);
  cgraph_finalize_compilation_unit();        // コンパイル単位終了
  cgraph_optimize();                         // ここからミドルエンド
}
```

ここまでして GCC を学ぶメリット

- 実用コンパイラについての理解が深まる
  - 教科書だけではなかなか学べない
  - 普段使っているものの中身を知りたい
    - プログラマとして自然な感想(のはず)
- C についての理解が深まる
  - C の構文がどういう構文木になるのか
- GCC は最高の生きた教材である
  - 20 年の歴史と実績（究極の実用アプリケーション）
  - 様々なプログラミングテクニックの宝庫
  - 世界最高峰の（変態）プログラマ集団の技に触れる

関数型プログラミング

- GCC の中身は C で書かれた Lisp
  - RMS は筋金入りの Lisper（MIT AI lab. 出身）
- Lisp のプログラミングテクニック
  - メタプログラミング
  - 内部/外部 DSL の塊
    - Lisp like API + プリプロセッサマジック（内部 DSL）
    - S 式からのコード生成（外部 DSL）
    - C コンパイラの中に独自 C サブセットのコンパイラが！
  - 実用アプリケーションでの貴重な実例

豊富なプログラミングテクニック

- クラス指向 OOP ライク(ユーザ定義データ型構築)
  - 構造体 + インタフェース(関数ポインタ)
  - C でクラス階層構築（継承）みたいなこと
  - function override (機能上書き)みたいなこと
- メタプログラミング(プログラムを生成するプログラム)
  - プリプロセッサマジック（全体的）
    - #include “*.def“ → 巨大な定義生成など
  - GGC (GCC Garbage Collection) メモリ管理
    - GC 用マーク付き C ソース → GNU C ソース生成
  - MD (Machine Description) → バックエンド生成
    - *.md (S 式) → *.c, *.h, *.mk, *.md

謎の呪文(1)

```
#undef  LANG_HOOKS_NAME
#define LANG_HOOKS_NAME                  "bl"
#undef  LANG_HOOKS_INIT
#define LANG_HOOKS_INIT                  bl_init
#undef  LANG_HOOKS_INIT_OPTIONS
#define LANG_HOOKS_INIT_OPTIONS          bl_init_options
#undef LANG_HOOKS_FINISH
#define LANG_HOOKS_FINISH                bl_finish
#undef  LANG_HOOKS_HANDLE_OPTION
#define LANG_HOOKS_HANDLE_OPTION         bl_handle_option
#undef  LANG_HOOKS_PARSE_FILE
#define LANG_HOOKS_PARSE_FILE            bl_parse_file
#undef  LANG_HOOKS_MARK_ADDRESSABLE
#define LANG_HOOKS_MARK_ADDRESSABLE      bl_mark_addressable
#undef  LANG_HOOKS_TYPE_FOR_MODE
#define LANG_HOOKS_TYPE_FOR_MODE         bl_type_for_mode
#undef  LANG_HOOKS_TYPE_FOR_SIZE
#define LANG_HOOKS_TYPE_FOR_SIZE         bl_type_for_size
#undef LANG_HOOKS_COMMON_ATTRIBUTE_TABLE
#define LANG_HOOKS_COMMON_ATTRIBUTE_TABLE    bl_attribute_table
#undef LANG_HOOKS_FORMAT_ATTRIBUTE_TABLE
#define LANG_HOOKS_FORMAT_ATTRIBUTE_TABLE    bl_format_attribute_table
#undef LANG_HOOKS_CALLGRAPH_EXPAND_FUNCTION
#define LANG_HOOKS_CALLGRAPH_EXPAND_FUNCTION tree_rest_of_compilation
```

なんぞこれー

わーん
こわいよー ＞_＜

謎の呪文(2)

```
const struct lang_hooks lang_hooks = LANG_HOOKS_INITIALIZER;

#define DEFTREECODE(SYM, NAME, TYPE, LENGTH) TYPE,
const enum tree_code_class tree_code_type[] = {
#include "tree.def"
  tcc_exceptional
};
#undef DEFTREECODE

#define DEFTREECODE(SYM, NAME, TYPE, LENGTH) LENGTH,
const unsigned char tree_code_length[] = {
#include "tree.def"
  0
};
#undef DEFTREECODE

#define DEFTREECODE(SYM, NAME, TYPE, LEN) NAME,
const char *const tree_code_name[] = {
#include "tree.def"
  "@@dummy"
};
#undef DEFTREECODE
```

やっぱり無理だよ...

お前はもう十分がんばった. . .

さぁ, 森へ帰ろう. . .

λ . .

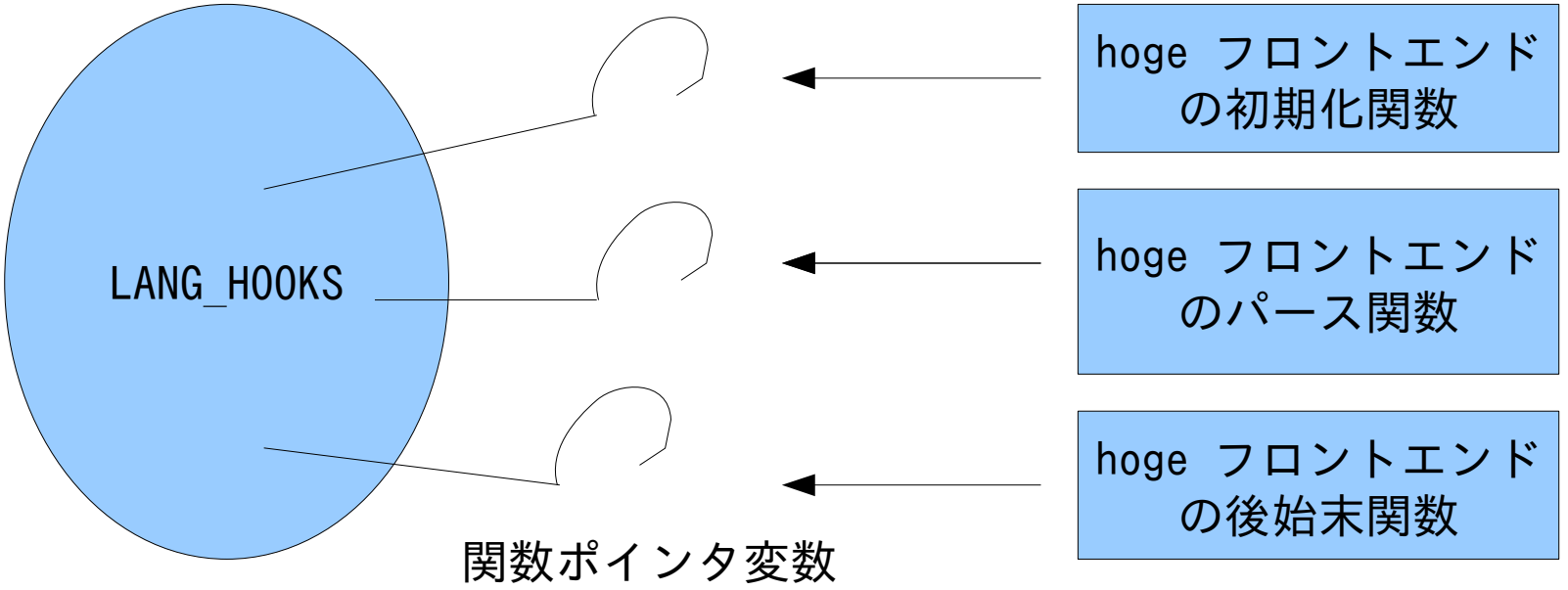
λ. .

λ . . .


おまじない
いっぱいだな
※ GCC は魔法がいっぱいの夢の国です
http://ja.wikipedia.org/wiki/%E7%94%BB%E5%83%8F:St-ignucius.jpg

第二の壁 ： LANG_HOOKS

- フロントエンドが実装しなければいけないインタフェース郡を定義
  - フック ： 機能を引っ掛けるところ
  - 機能の差し替えが可能（function overwriting）
  - 様々なフロントエンドを実現するためのしくみ



他にもミドルエンドなどにも

- 最適化パスの追加・削除
  - QLeap さん談
- passes.c の init_optimization_passes()
  - struct tree_opt_pass **p;
  - #define NEXT_PASS(PASS) (p = next_pass_1 (p, &PASS))
  - NEXT_PASS (pass_remove_useless_stmts);
  - NEXT_PASS (pass_mudflap_1);
  - NEXT_PASS (pass_lower_omp);
  - NEXT_PASS (pass_lower_cf);
  - NEXT_PASS (pass_refactor_eh);
  - NEXT_PASS (pass_lower_eh);
  - NEXT_PASS (pass_build_cfg);
  - NEXT_PASS (pass_lower_complex_OO);
  - NEXT_PASS (pass_lower_vector);
  - . . .

プリプロセッサマジック（1）
LANG_HOOKS

- 巨大なマクロの塊

```
#undef  LANG_HOOKS_NAME
#define LANG_HOOKS_NAME                 "bl"
#undef  LANG_HOOKS_INIT
#define LANG_HOOKS_INIT                 bl_init
#undef  LANG_HOOKS_INIT_OPTIONS
#define LANG_HOOKS_INIT_OPTIONS         bl_init_options
#undef LANG_HOOKS_FINISH
#define LANG_HOOKS_FINISH               bl_finish
#undef  LANG_HOOKS_HANDLE_OPTION
#define LANG_HOOKS_HANDLE_OPTION        bl_handle_option
#undef  LANG_HOOKS_PARSE_FILE
#define LANG_HOOKS_PARSE_FILE           bl_parse_file
...

const struct lang_hooks lang_hooks = LANG_HOOKS_INITIALIZER;
```

lang_hooks 構造体の初期化マクロ

```
•#define LANG_HOOKS_INITIALIZER { ¥
          LANG_HOOKS_NAME, ¥ (例 : ”GNU C”)
          . . .
          LANG_HOOKS_INIT, ¥ (例 : c_objc_common_init)
          LANG_HOOKS_FINISH, ¥
          LANG_HOOKS_PARSE_FILE, ¥
          . . .
          LANG_HOOKS_DECLS, ¥
          LANG_HOOKS_FOR_TYPES_INITIALIZER, ¥
          LANG_HOOKS_GIMPLIFY_EXPR, ¥
          LANG_HOOKS_FOLD_OBJ_TYPE_REF, ¥
          LANG_HOOKS_BUILTIN_FUNCTION, ¥
          LANG_HOOKS_INIT_TS, ¥
          LANG_HOOKS_EXPR_TO_DECL, ¥
        }
```

プリプロセッサマジック(2)
DEFTREECODE

- 巨大な定義生成

```
#define DEFTREECODE(SYM, NAME, TYPE, LENGTH) TYPE,

const enum tree_code_class tree_code_type[] = {

#include ”tree.def”

  tcc_exceptional

};

#undef DEFTREECODE
```

- tree.def は巨大な定義ファイル

tree.def

```
DEFTREECODE (ERROR_MARK, "error_mark", tcc_exceptional, 0)
DEFTREECODE (IDENTIFIER_NODE, "identifier_node", tcc_exceptional, 0)
DEFTREECODE (TREE_LIST, "tree_list", tcc_exceptional, 0)
DEFTREECODE (TREE_VEC, "tree_vec", tcc_exceptional, 0)
DEFTREECODE (BLOCK, "block", tcc_exceptional, 0)
DEFTREECODE (OFFSET_TYPE, "offset_type", tcc_type, 0)
DEFTREECODE (ENUMERAL_TYPE, "enumeral_type", tcc_type, 0)
DEFTREECODE (BOOLEAN_TYPE, "boolean_type", tcc_type, 0)
DEFTREECODE (INTEGER_TYPE, "integer_type", tcc_type, 0)
DEFTREECODE (REAL_TYPE, "real_type", tcc_type, 0)
DEFTREECODE (POINTER_TYPE, "pointer_type", tcc_type, 0)
DEFTREECODE (FIXED_POINT_TYPE, "fixed_point_type", tcc_type, 0)
DEFTREECODE (REFERENCE_TYPE, "reference_type", tcc_type, 0)
DEFTREECODE (COMPLEX_TYPE, "complex_type", tcc_type, 0)
DEFTREECODE (VECTOR_TYPE, "vector_type", tcc_type, 0)
DEFTREECODE (ARRAY_TYPE, "array_type", tcc_type, 0)
DEFTREECODE (RECORD_TYPE, "record_type", tcc_type, 0)
DEFTREECODE (UNION_TYPE, "union_type", tcc_type, 0)
DEFTREECODE (QUAL_UNION_TYPE, "qual_union_type", tcc_type, 0)
DEFTREECODE (VOID_TYPE, "void_type", tcc_type, 0)
DEFTREECODE (FUNCTION_TYPE, "function_type", tcc_type, 0)
DEFTREECODE (METHOD_TYPE, "method_type", tcc_type, 0)
DEFTREECODE (LANG_TYPE, "lang_type", tcc_type, 0)
DEFTREECODE (INTEGER_CST, "integer_cst", tcc_constant, 0)
DEFTREECODE (REAL_CST, "real_cst", tcc_constant, 0)
...
```

cpp 展開後

- #define DEFTREECODE(SYM, NAME, TYPE, LENGTH) TYPE,
- const enum tree_code_class tree_code_type[] = {
- #include ”tree.def”

↓

```
const enum tree_code_class tree_code_type[] = {
tcc_exceptional,
tcc_exceptional,
tcc_exceptional,
tcc_exceptional,
tcc_exceptional,
tcc_type,
...
  tcc_exceptional
};
```

← もし手で書こうとしたらすごく大変
（数や対応関係を間違えそう．．．保守性最悪）

謎の呪文(3)

- あれ？ プログラミング言語 C にこんな構文あったっけ ???

```
union lang_tree_node  GTY((desc ("TREE_CODE (&%h.generic) == IDENTIFIER_NODE"))) {
  union tree_node GTY ((tag ("0"), desc ("tree_node_structure (&%h)"))) generic;
  struct lang_identifier GTY ((tag ("1"))) identifier;
};
```

- しかも，こんなファイル作った覚え無いぞ ？
  - うちの子じゃありません ！ (o^⌐^o)

#include "gt-bl-bl1.h" (gt-*filepath*.h)

#include "gtype-bl.h"

(gtype-*言語フロントエンドサブディレクトリ名*.h)

第三の壁 : GGC (GCC Garbage Collection)

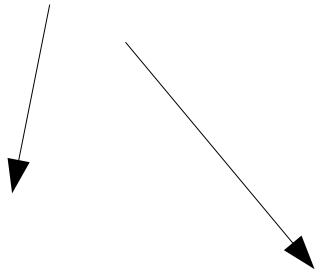
- 独自拡張された C で GCC は書かれる
  - 様々な情報を付加する GTY タグが付いてる
- GC 機構のコードが付加された C コードが自動生成される（メタプログラミング！）
  - そのためのコンパイラを独自に持ってる
  - コンパイラの中にコンパイラがある！
  - GCC developer は頭おかしい （褒め言葉）
- 独創的なメモリ管理が実用化済み
- あんまり僕もわかってないので，詳しくはドキュメントを...

怪しい GGC の解説

- GTY (([option] [(param)], [option] [(param)] ...))
  - 構造体/共用体定義の { の前
  - グローバル変数宣言の static/extern の後ろ
  - 構造体フィールド宣言の名前の前
- メモリ管理のための様々な情報を付加する
  - The Inside of a GTY (GCC internals の拙訳)
    - http://tinyurl.com/5aljar
  - GGC パーサは C サブセットを理解 (typedef)
- config-lang.in の gtfiles 変数に, GTY マークを含むファイルの名前を全て追加する

minimal frontend から引っ張ってきた例

%h : 対象の構造体自身



```
union lang_tree_node  GTY((desc ("TREE_CODE (&%h.generic) == IDENTIFIER_NODE"))) {
  union tree_node GTY ((tag ("0"), desc ("tree_node_structure (&%h)"))) generic;
  struct lang_identifier GTY ((tag ("1"))) identifier;
};
```

desc(“expression)

tag(“constant)

→ 共用体のどのフィールドが，実際に有効になっているのかという情報

今後の展望
（妄想）

ちょっと楽しい可能性

- 要するに bl_parse_file() {...}
- ここだけ切り出して，動的リンクできるように
  - dl でも LD_PRELOAD でも
- GCC 全体をビルドしなくても OK に
- 別に C で書かなくても良い ？
  - Pascal でも Java でも，GCC のフロントエンドが存在する言語ならばなんでも良い ？

さらに楽しい可能性

- yacc/lex 面倒 → Boost::spirit とか使う ？
  - C++ で extern C で parse_file 書けば良いだけ
  - （だと思う）
  - Write GCC in C++ なんて話も
    - http://lwn.net/Articles/286539/
  - GGC とか無茶なハックとか不要になるかも ？
- GCC の API を Ruby とかから使えるように
  - スクリプト言語でコンパイラ書きたい，とか
  - Haskell の parsec とか使いたい，とか

もっと楽しい可能性

- bl_parse_file() から，スクリプト言語のインタプリタを呼び出す
- スクリプト言語側から GCC API 呼び出して
- 構文木作ってミドルエンドに渡す
- GCC 本体は一切再ビルド不要
- スクリプト言語でプログラム書けば，GCC のフロントエンドが書ける！
  - iLogScript（id:w_o さんが 3.x 時代に通った道）

読み物リンク

- GCC のフロントエンド作る日記（1）～（26）
  - 私のブログです
  - 全然まとまってませんが，情報はあるかも...
  - http://alohakun.blog7.fc2.com/?all
- ソースからバイナリへ: GCCの内部機構
  - http://www.jp.redhat.com/magazine/N05/
- GCCに匹敵するコンパイラ?! LLVM – BSDCan2008
  - http://journal.mycom.co.jp/articles/2008/06/03/bsdcan6/index.html

まとめ

- GCCって僕でもハックできそう！
  - という夢を子供たちと大きなお友達に与えたい！
- コンパイラ作りの面白さ，奥深さを共有したい！
- 叩き台を提供した
  - あとはやりたい放題！！
- 最後に ...

やっぱり35分では
無理でした！
ごめんなさい

俺たちの GCC Hacks はまだ始まったばかりだ！

応援ありがとうございました
あろは先生の次回セミナーにご期待ください

ご清聴
ありがとう
ございました