バイナリ
だけが
出力じゃない

2008/08/23

MORITA Hajime <omo@dodgson.org>
http://steps.dodgson.org/

今日の話: LLVM のバックエンド

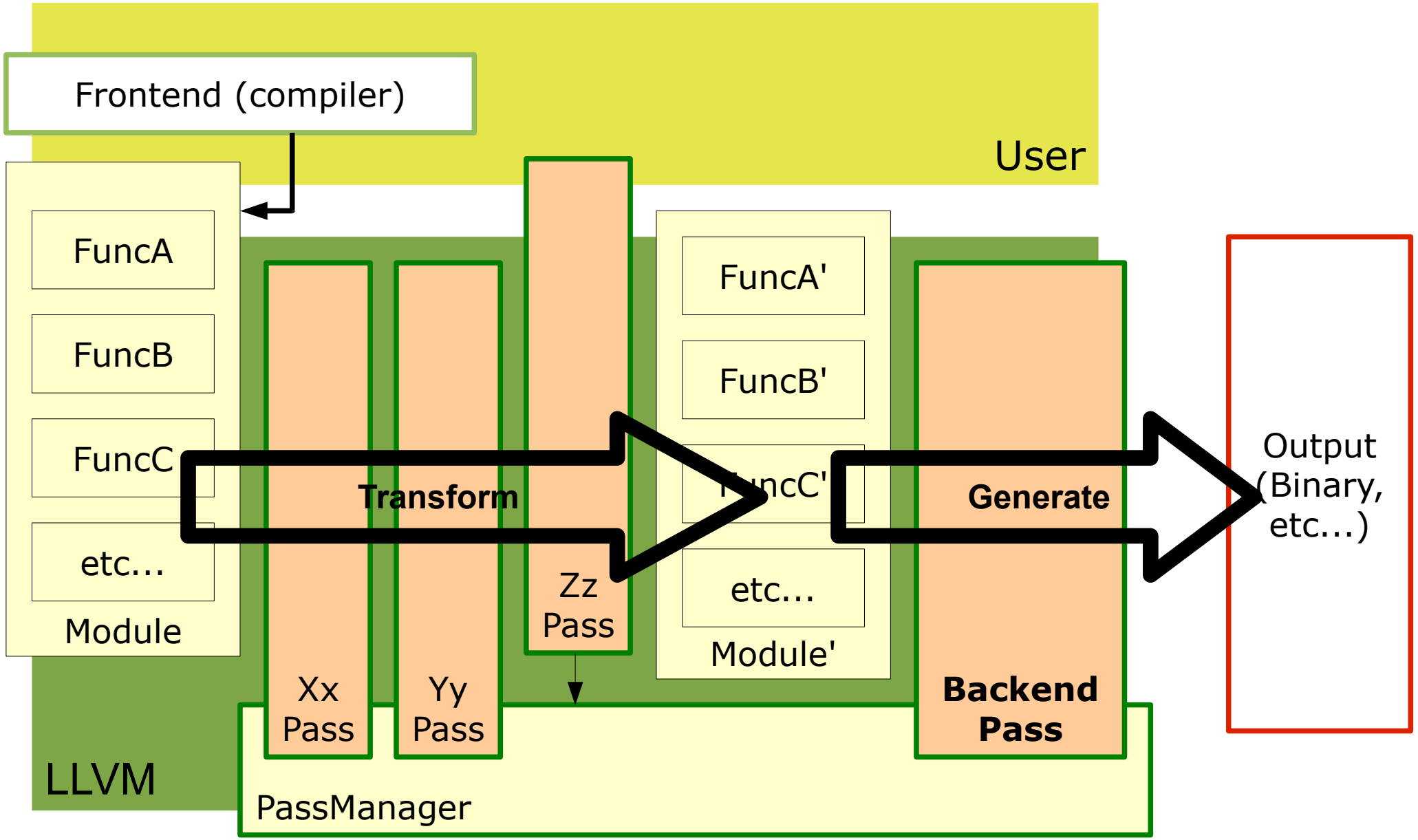
- Backend = 機械語を出力するモジュール <-> Frontend
- Pluggable になっている
  - CPU 色々
- 機械語以外も出力できる

binaries are not only output

Architecture

構成要素

- 言語モデル
  - モジュール(**Module**)、関数(**Function**)、命令、変数...
- モデル解析/変換アルゴリズム(**Pass**)
  - 最適化、セキュリティチェック、機械語生成
- ドライバ(**PassManager**), etc.
  - Pass のリストを管理する
  - モデルを Pass に順番どおり引き渡す


Frontend (compiler)
User
FuncA
FuncB
FuncC
etc...
Module
Transform
Xx
Pass
Yy
Pass
Zz
Pass
FuncA'
FuncB'
FuncC'
etc...
Module'
Generate
Backend
Pass
Output
(Binary,
etc...)
LLVM
PassManager

Pass API (plug する側)

```
class Pass {
  virtual const char *getPassName() const;
  virtual PassManagerType getPotentialPassManagerType() const {
    return PMT_Unknown;
  }
  ...
};

class ModulePass : public Pass {
public:
  virtual bool runOnModule(Module &M) = 0;
  ...
};

class FunctionPass : public Pass {
  virtual bool doInitialization(Module &M) { return false; }
  virtual bool runOnFunction(Function &F) = 0;
  virtual bool doFinalization(Module &M) { return false; }
  ...
};

class BasicBlockPass : public Pass { .... };
```

Pass API (plug される側)

```
class PassManagerBase {
public:
  ....
  virtual void add(Pass *P) = 0;
  ....
};

class PassManager : public PassManagerBase {
public:
  ....
  void add(Pass *P);
  bool run(Module &M);
  ....
};

class FunctionPassManager : public PassManagerBase {
  ....
  void add(Pass *P);
  bool run(Function &F);
  ....
};
```

実例

```
// from llc.cpp
    ....
    // pass 登録
    FunctionPassManager Passes(&Provider);
    Passes.add(new TargetData(*Target.getTargetData()));
    ....
    Target.addPassesToEmitFile(Passes, *Out, FileType, Fast);
    ....
    AddELFWriter(Passes, *Out, Target);
    Target.addPassesToEmitFileFinish(Passes, MCE, Fast);

    // function への pass 適用
    Passes.doInitialization();
    for (Module::iterator I = mod.begin(), E = mod.end();
         I != E; ++I)
      if (!I->isDeclaration())
        Passes.run(*I);
    Passes.doFinalization();
    ....
```

標準添付の Pass 実装

- 最適化いろいろ
- 解析いろいろ
  - 最適化でつかうなど
- デバッグ支援すこし
  - テキスト形式 bitcode 出力など
- …

既存の Passes (CodeGen)

```
// include/llvm/CodeGen/Passes.h
namespace llvm {
  FunctionPass *createRegisterAllocator();
  FunctionPass *createSimpleRegisterAllocator();
  FunctionPass *createLocalRegisterAllocator();
  FunctionPass *createBigBlockRegisterAllocator();
  FunctionPass *createLinearScanRegisterAllocator();
  RegisterCoalescer *createSimpleRegisterCoalescer();
  FunctionPass *createPrologEpilogCodeInserter();
  FunctionPass *createLowerSubregsPass();
  FunctionPass *createPostRAScheduler();
  FunctionPass *createBranchFoldingPass(bool
DefaultEnableTailMerge);
  FunctionPass *createIfConverterPass();
  FunctionPass *createLoopAlignerPass();
  FunctionPass *createDebugLabelFoldingPass();
  FunctionPass *createMachineCodeDeleter();
  FunctionPass *getRegisterAllocator(TargetMachine &T);
  FunctionPass *createGCLoweringPass();
  ...
}
```

- Factory method： 実装は `lib/CodeGen/*.cpp` に
- 最適化アルゴリズムの実装など

既存の Passes (Analysis)

```
// include/llvm/Analysis/Passes.h
namespace llvm {
  ...
  Pass *createGlobalsModRefPass();
  ModulePass *createAliasAnalysisCounterPass();
  FunctionPass *createAAEvalPass();
  ImmutablePass *createNoAAPass();
  ImmutablePass *createBasicAliasAnalysisPass();
  FunctionPass *createLibCallAliasAnalysisPass(LibCallInfo *LCI);
  ModulePass *createAndersensPass();
  ImmutablePass *createBasicVNPass();
  ModulePass *createProfileLoaderPass();
  ImmutablePass *createNoProfileInfoPass();
  ModulePass *createDSAAPass();
  ModulePass *createDSOptPass();
  ModulePass *createSteensgaardPass();
  FunctionPass *createInstCountPass();
  ...
}
```

- 補助データ構造の構築など

本題: Backend API / Passes

- **Backend も Pass の一種**
  - 構成は各実装次第
  - 複数 Pass なのがふつう
- Backend 固有のインターフェイスすこし
  - オブジェクトファイルの生成支援
  - バックエンドドライバに対する共通API
「必要なパス一式を登録」など

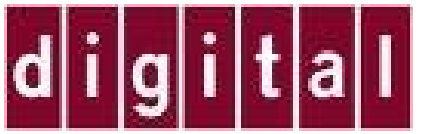
Backend API (plug する側)

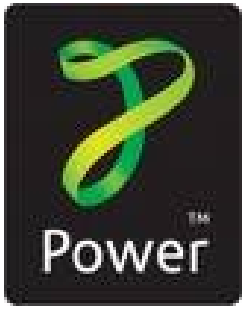
```
// llvm-2.3/include/llvm/Target/TargetMachine.h

class TargetMachine {
  virtual const TargetAsmInfo *createTargetAsmInfo() const ...
  ...
  virtual FileModel::Model
  addPassesToEmitFile(PassManagerBase &PM,
                      std::ostream &Out,
                      CodeGenFileType FileType,
                      bool Fast) ...
  ...
};
```

What's Available?

標準添付

- ARM
- Alpha
- MIPS
- CELL
- PowerPC
- X86
- IA64
- Sparc
- MSIL
- ...
- できはまちまち？





実験的な実装 by Adobe

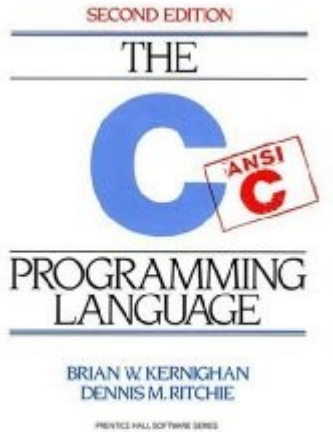
- LLVM -> ABC (ActionScript Bytecode)
  - 中の人が実験的につくったもの
  - 実物はなし
  - 資料はあり
    “Flash C Compiler: Compiling C code to the Adobe Flash Virtual Machine”
    http://llvm.org/devmtg/2008-08/Petersen_FlashCCompiler.pdf
  - Doom が動くらしい (Adobe MAX でデモ)
    “Flash on C/C++ Sneak Peek”
    http://jp.youtube.com/watch?v=0hX-Uh3oTcE
    - 割とさくさくうごいてびびる

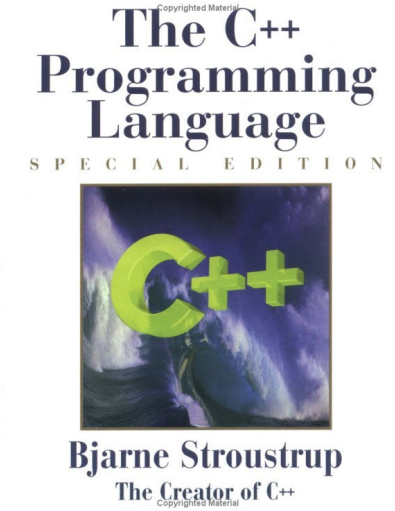
Flash C Compiler

- スライドみること > 自分
- 時間がなければあとで

標準添付つづき

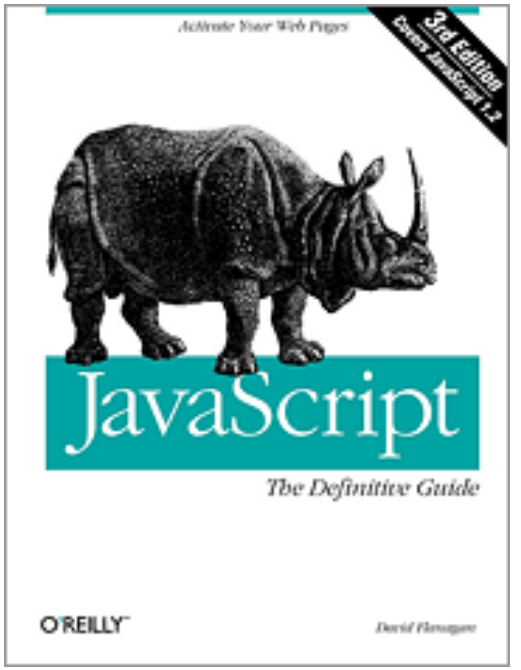
- CBackend
  `llvm-2.3/lib/Target/CBackend/`
  - C のコードを出力
  - 割とちゃんと動くらしい (syoyo 談)
- CppBackend
  `lib/Target/CppBackend/`
  - C++ のコードを出力(たぶん)





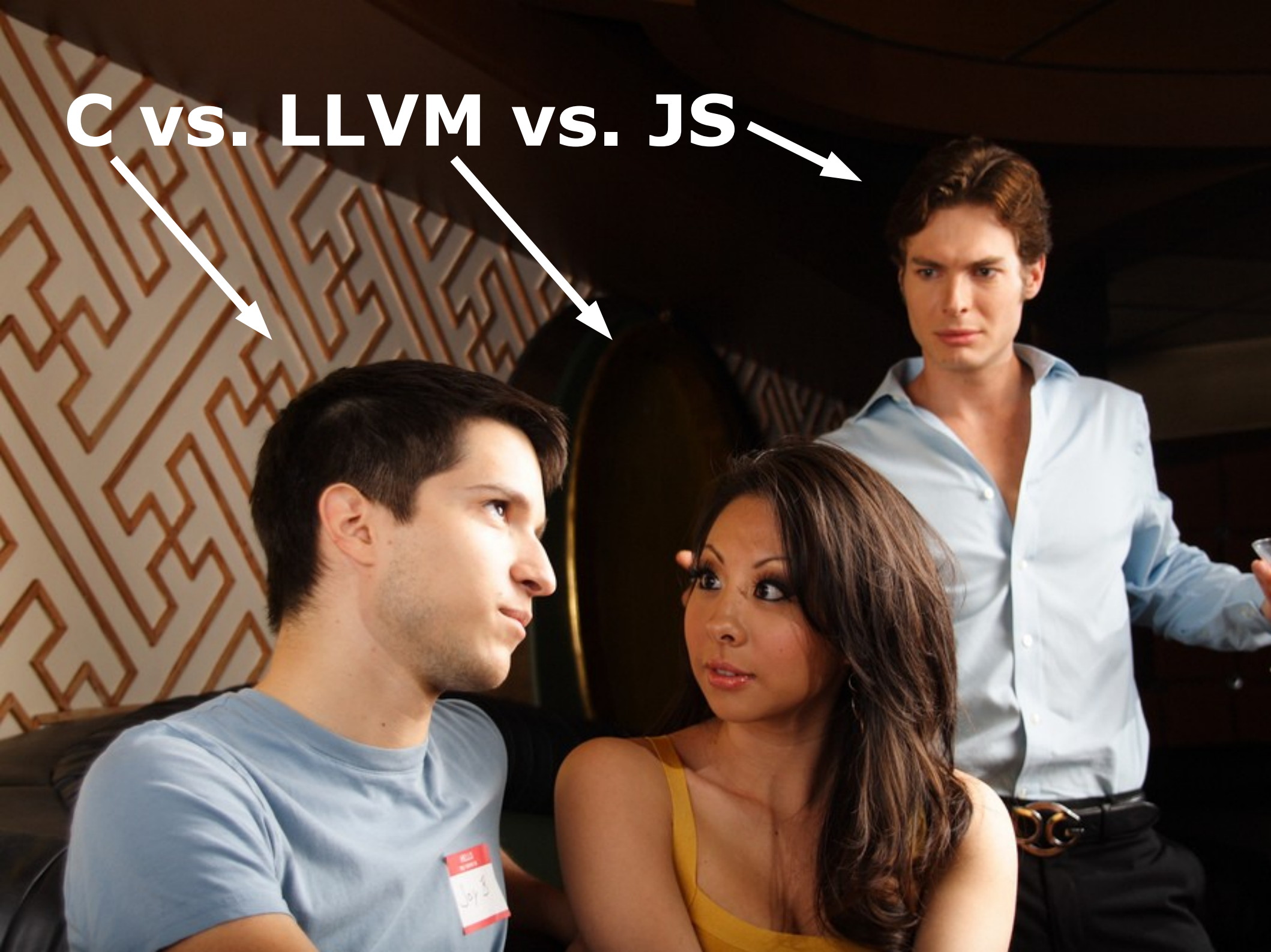
JSBackend

- つくってみた
- CBackend ぱくり
- ぜんぜん動かない
  - キャストしちゃだめ
  - 構造体の値渡しダメ
  - 他にもいろいろダメだが詳細忘れ



試しかた

```
$clang -emit-llvm-bc hello.c
# generates hello.bc
## -emit-llvm for hello.ll (text)
$jsllc hello.bc -o hello.js
# generates hello.js
```


C vs. LLVM vs. JS

比較の題材

- 関数/変数定義
- 制御構造
- ポインタ演算

比較(1/3)：関数/変数 (C)

```
int add(int a, int b) {
 int x = a + b;
 return x;
}
```

比較(1/3)：関数/変数 (C->**LL**)

```
define i32 @add(i32 %a, i32 %b) nounwind {
entry:
        %a.addr = alloca i32
        %b.addr = alloca i32
        %x = alloca i32, align 4
        store i32 %a, i32* %a.addr
        store i32 %b, i32* %b.addr
        %tmp = load i32* %a.addr
        %tmp1 = load i32* %b.addr
        %add = add i32 %tmp, %tmp1
        store i32 %add, i32* %x
        %tmp2 = load i32* %x
        ret i32 %tmp2
}
```

比較(1/3)：関数/変数 (C->LL->**JS**)

```
function add(a, b)
{
      var a_addr = __JSBE.make_memory();
      var b_addr = __JSBE.make_memory();
      var x = __JSBE.make_memory();
      a_addr.set_ref(a);
      b_addr.set_ref(b);
      var tmp = a_addr.ref();
      var tmp1 = b_addr.ref();
      var add = tmp + tmp1;
      x.set_ref(add);
      var tmp2 = x.ref();
      return tmp2;
}
```

JSBackend Runtime

```
__JSBE = {
 ...
 MemoryArray: function(arr, off) {
   this.arr = arr || [];
   this.off = off || 0;
   ...
   this.ref = function() {
     var toret = this.arr[this.off];
     return undefined == toret ? 0 : toret;
   };
   this.set_ref = function(ch) {
     this.arr[this.off] = ch;
   };
   ...
 },
 make_memory: function() {
   return new __JSBE.MemoryArray();
 },
 ...
};
```

比較(1/3)：関数/変数 (C->LL->**JS**)

```
function add(a, b)
{
      var a_addr = __JSBE.make_memory();
      var b_addr = __JSBE.make_memory();
      var x = __JSBE.make_memory();
      a_addr.set_ref(a);
      b_addr.set_ref(b);
      var tmp = a_addr.ref();
      var tmp1 = b_addr.ref();
      var add = tmp + tmp1;
      x.set_ref(add);
      var tmp2 = x.ref();
      return tmp2;
}
```

比較(1/3)：関数/変数 (C->LL->**JS**)

```
function add(a, b)
{
      var
      var
      var
      a_
      b_
      v
      va
      var add =
      x.set_ref(add);
      var tmp2 = x.ref();
      return tmp2;
}
```

```
make_memory();
emory();
```

Toooo Slow!!

opt: LLVM Optimizer

```
$opt -std-compile-opts hello.bc
      -o=- > hello_opt.bc
```

比較(1b/3)：関数/変数 (C->**LL**)

```
define i32 @add(i32 %a, i32 %b) nounwind {
entry:
        %add = add i32 %b, %a
        ret i32 %add
}
```

比較(1b/3)：関数/変数 (C->LL->**JS**)

```
function add(a, b)
{
     var add = b + a;
     return add;
}
```

この先の例はすべて optimized です

比較(2/3)：制御構造 (C)

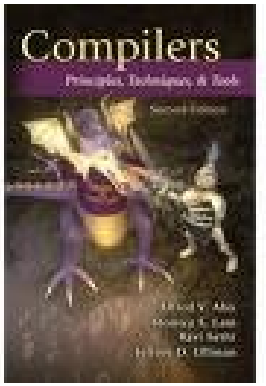
```
int fac(int n) {
    int x = 1, i = 0;
    for (i=0; i<n; i++) { x *= (i+1); }
    return x;
}
```

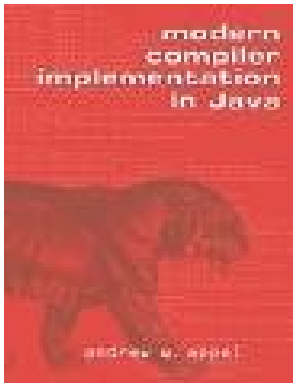
比較(2/3)：制御構造 (C->**LL**)

```
define i32 @fac(i32 %n) nounwind {
entry:
  %cmp3 = icmp sgt i32 %n, 0
  br i1 %cmp3, label %forbody, label %afterfor

forbody:
  %i.02 = phi i32 [ 0, %entry ], [ %add, %forbody ]
  %x.01 = phi i32 [ 1, %entry ], [ %mul, %forbody ]
  %add = add i32 %i.02, 1
  %mul = mul i32 %add, %x.01
  %exitcond = icmp eq i32 %add, %n
  br i1 %exitcond, label %afterfor, label %forbody

afterfor:
  %x.0.lcssa = phi i32 [ 1, %entry ], [ %mul, %forbody ]
  ret i32 %x.0.lcssa
}
```

phi function

- 遷移前のBBによって値が変わる魔法の変数
  - SSA用語
  - 詳しくは教科書を
- ハマりがち（？）






```
   ...
entry:
   ...
forbody:
   %i.02 = phi i32 [ 0, %entry ], [ %add, %forbody ]
   %x.01 = phi i32 [ 1, %entry ], [ %mul, %forbody ]
   ...
afterfor:
   ...
```

比較(2/3)：制御構造 (C->LL->**JS**)

- **`while-switch`** で **`goto`** 代替
  - 機械語ではしない苦労
- phi function は遷移前のBBで変数代入
  - これは機械語でも同じ
- 現物は次ページ (ながいです)

```
function fac(n)
{
 var i_02; // phi var
 var x_01; // phi var
 var x_0_lcssa; // phi var
 var __bb = 0;
 while (-1 != __bb) {
   switch(__bb) {
   case 0: // %entry
     var cmp3 = n > 0;
     if (cmp3) {
       // %i.02 = phi i32 [0, %entry ], [%add, %forbody]
       i_02 = 0;
       x_01 = 1;
       __bb = 1;
     } else {
       x_0_lcssa = 1;
       __bb = 2;
     }
     break;
```

(後半につづく...)

(...前半のつづき)

```
    case 1: // %forbody
      var add = i_02 + 1;
      var mul = add * x_01;
      var exitcond = add == n;
      if (exitcond) {
        x_0_lcssa = mul;
        __bb = 2;
      } else {
        // %i.02 = phi i32 [0, %entry ], [%add, %forbody]
        i_02 = add;
        x_01 = mul;
        __bb = 1;
      }
      break;
    case 2: // %afterfor
      return x_0_lcssa;
      break;
    default: break;
    }
  }
}
```

Have a break...

比較(3/3)：ポインタ (C)

```
void puts(const char* str) {
  while (*str) {
    putc(*(str++)); // putc() は 外で定義
  }
}
```

比較(3/3)：ポインタ操作 (C->LL)

```
define void @puts(i8* %str) nounwind {
entry:
  %tmp12 = load i8* %str ;; char tmp12 = *str;
  ...
whilebody:
  %str.addr.01.rec = phi i32 [ 0, %entry ], ...
  ;; char* str_addr_1 = str + str_addr_01_rec;
  %str.addr.1 = getelementptr i8* %str, \
                              i32 %str.addr.01.rec
  %tmp3 = load i8* %str.addr.01
  tail call void @putc( i8 signext %tmp3 )
  %ptrincdec.rec = add i32 %str.addr.1.rec, 1
  %ptrincdec = getelementptr i8* %str, \
                             i32 %ptrincdec.rec
  %tmp1 = load i8* %ptrincdec
....
}
```

GetElementPtr

- ポインタ演算
- 参照はしない
  - load/store が参照
- 割とややこしい(可変長引数など)

```
;; char* str_addr_1 = str + str_addr_01_rec;
%str.addr.1 = getelementptr i8* %str, \
                            i32 %str.addr.01.rec
%tmp3 = load i8* %str.addr.01
```

比較(3/3)：ポインタ (C->LL->**JS**)

```
function puts(str) {
  ...
  switch(__bb) {
  case 0:
    var tmp12 = str.ref();
    ...
    break;
  case 1:
    var str_addr_01 = \
      str.element_at(str_addr_1_rec);
    var tmp3 = str_addr_01.ref();
    putc(tmp3);
    var ptrincdec_rec = str_addr_01_rec + 1;
    var ptrincdec = str.element_at(ptrincdec_rec);
    ...
 }
}
```

余談: JSBackend Runtime (2)

```
MemoryArray: function(arr, off) {
  ...
  this.element_at = function() {
    var sum = 0;
    for (var i=0; i<arguments.length; i++) {
       sum += arguments[i];
    }
    return new __JSBE.MemoryArray(this.arr, sum);
  };
  ...
},
```

Demo

- 料理番組方式
- とても地味です

扱っていない話題

- 言語機能いろいろ
  - 構造体
    - 型の混じった配列扱い
  - 値渡し
  - キャスト
  - グローバル変数、リテラル
- 最適化いろいろ
  - インライン展開、アンローリング、....
- などなど.....

バックエンドをつくるのは...

- Pros
  - LLVM の命令セットに詳しくなる
  - Clang の出力に詳しくなる
  - CBackend に詳しくなる
- Cons
  - 役に立たない
  - JS よくない: ネイティブのバイト列がない
    - ruby の方がマシかも: String, pack/unpack

まとめ

- LLVM はバックエンドが pluggable
- ソースコードを書き出すのもあり

バイナリ
だけが
出力じゃない

写真たち (CC-NA 2.0)

http://flickr.com/photos/genista/6262747/

http://flickr.com/photos/seandreilinger/321723071/

http://flickr.com/photos/docman/358331914/

http://www.thewvsr.com/adsvsreality.htm

http://flickr.com/photos/sgw/2378431809/

http://flickr.com/photos/thisisawakeupcall/1216243808/

http://flickr.com/photos/philippeleroyer/495960750/