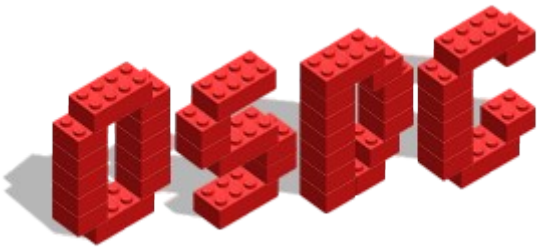
Build Programming Language Runtime with LLVM

Jim Huang (黃敬群)

Developer & Co-founder, 0xlab

jserv@0xlab.org

@ 1500 / March 27, 2011



$1500_{10} = 05DC_{16}$

- OSDC !
  - Open Source Developer Conference
  - 0x05DC
- About 0xlab
  - The meaning of open
  - $0x1ab_{16} = 427_{10}$ → 2009/04/27
- About me
  - http://about.me/jserv
  - 最大的專長就是培養興趣

Rights to copy

http://0xlab.org/

contact@0xlab.org

creative commons
COMMONS DEED

Attribution – ShareAlike 3.0

Corrections, suggestions, contributions and translations are welcome!

You are free

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Latest update:March 27, 2011

Under the following conditions

BY: **Attribution**. You must give the original author credit.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

License text: http://creativecommons.org/licenses/by-sa/3.0/legalcode

姊妹議程：
〈窮得只剩下 Compiler〉
OSDC.tw 2009

http://www.slideshare.net/jserv/what-can-compilers-do-for-us

姊妹議程：
〈身騎 LLVM ，過三關：
淺談編譯器技術的嶄新應用〉
TOSSUG 2009
http://www.slideshare.net/jserv/llvm-introduction

姊妹議程：
〈 Applied Computer Science Concepts in Android 〉
台大資訊系 2010
http://www.slideshare.net/jserv/applied-computer-science-concepts-in-android

姊妹議程：
〈 from Source to Binary -- How GNU Toolchain Works 〉
臺北科技大學資訊工程所 2011/03/31

提綱

(1) Compilers on Rails 的時代

(2) 探索 LLVM

(3) 程式語言的變遷 (Low-Level 觀點)

傳統 -> 動態 -> 移動運算

(4) LLVM 實例

Compilers on Rails 的時代

Compilers on Rails 的時代

[詞彙] on the rails: 正常運行；在正常軌道

[啓發] Ruby on Rails:

(1) **convention** over configuration

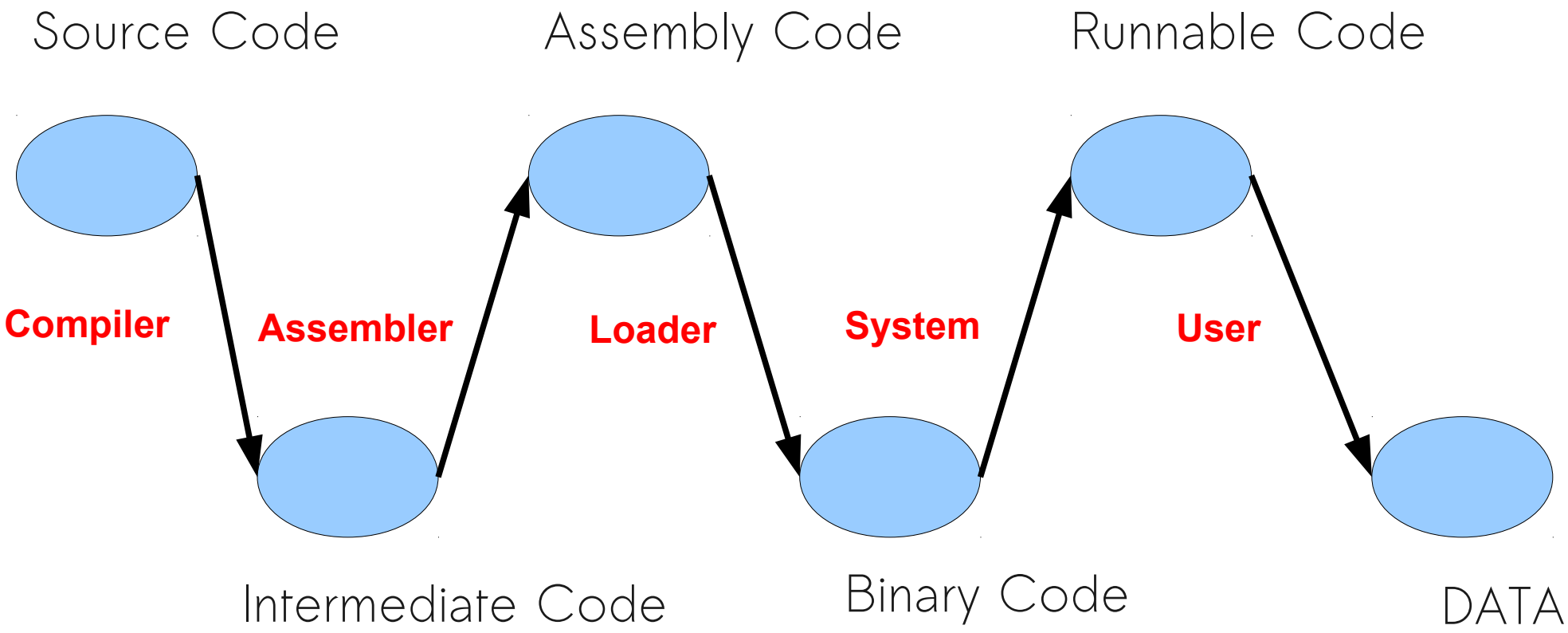
(2) **less** software

(3) programmer **happiness** ultimately leads to better **productivity**

隱藏在我們身邊的 Compiler

- Java / .Net (虛擬機器 +Just-In-Time compiler)
- 網路瀏覽器
  - Mozilla/Firefox (ActionMonkey/Tamarin)
  - WebKit (SquirrelFish)
  - Google Chrome (V8 engine)
- Web 應用程式： JSP/Servlet, SilverLight/.Net
- 移動通訊平台： Java ME, Android, iPhone, Portable Native Client
- 繪圖軟體： Adobe PixelBender, Shader
- 3D 高品質圖形處理： Gallium3D / OpenGL / Direct3D / RenderScript (Android)

傳統的 Compiler 流程



IR (Intermediate Representation) 可說是 Compiler 的心臟

LLVM = Low Level Virtual Machine

Specialize 技巧

以 color space 轉換來說，執行時期得負擔大量且繁瑣的運算，如 BGRA 444R→ RGBA 8888

```
for each pixel {
  switch (infmt) {
  case RGBA 5551:
    R = (*in >> 11) & C
    G = (*in >> 6) & C
    B = (*in >> 1) & C
  ... }
  switch (outfmt) {
  case RGB888:
    *outptr = R << 16 |
              G << 8 ...
  }
}
```

Run-time specialize

```
for each pixel {
  R = (*in >> 11) & C;
  G = (*in >> 6) & C;
  B = (*in >> 1) & C;
  *outptr = R << 16 |
            G << 8 ...
}
```

Compiler optimizes shifts and masking

Speedup depends on src/dest format:
- 5.4x speedup average, 19.3x speedup max (13.3MB/s to 257.7MB/s)

Compiler 領導技術的時代

- 運算模式已大幅改觀
- Framework-driven
- SIMD/vectorization, Cell, SMP/multi-core
- 虛擬化 (Virtualization) 技術的時代

  → 更多元、更安全、更有效率地使用硬體
- 資訊技術的雜交 (cross-over)
- LLVM 的大一統宏願

案例:
Portable Native Client, OpenCL (GPGPU)

到處都有 VM

Java Virtual Machine (JVM)

.NET Common Language Runtime (CLR)

Smalltalk

Squeak

Parrot (Perl 6)

Python

YARV (Ruby 1.9)

Rubinius

Tamarin (ActionScript)

Valgrind (C++)

Lua

TrueType

Dalvik

Adobe Flash (AVM2)

p-code (USCD Pascal)

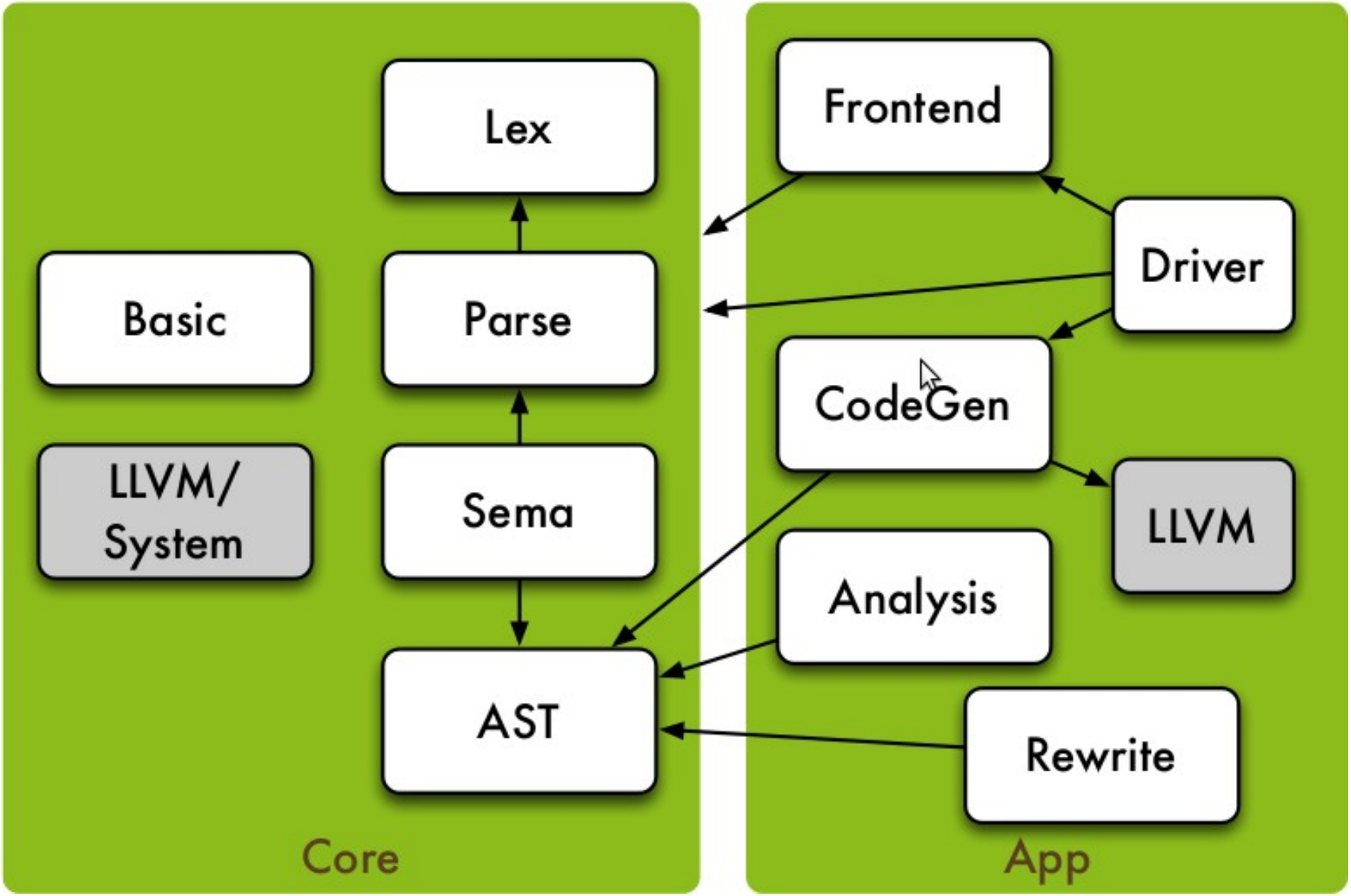
Zend

LLVM 可作為上述的編譯器應用的根基

探索 LLVM

LLVM

- Low-Level VM → bit-code
- 完整的編譯器基礎建設
  - 可重用的、用以建構編譯器的軟體元件
  - 允許更快更完整的打造新的編譯器
  - static compiler, JIT, trace-based optimizer, ...
- 開放的編譯器框架
  - 多種程式語言支援
  - 高彈性的自由軟體授權模式 (BSD License)
  - 活躍的開發 (50% 開發者來自 Apple Inc.)
  - 豐富的編譯輸出： C, ARM, x86, PowerPC, ...



Eric Schmidt 是 UNIX lex 共同作者

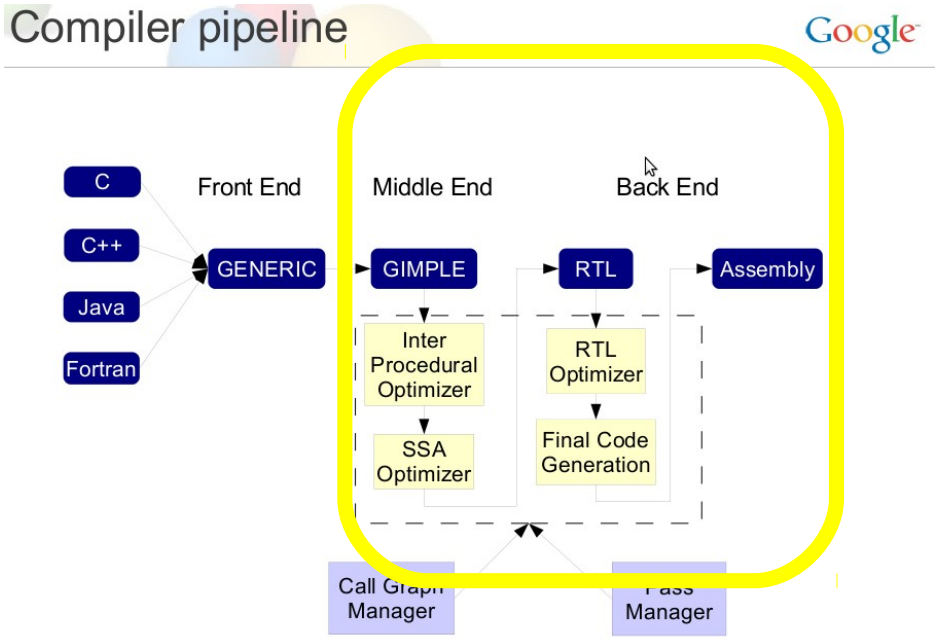
GCC vs. LLVM

GCC

C, C++, Obj-C, Fortran, Java, Ada, ...

x86, ARM, MIPS, PowerPC, ...

binutils (ld as)



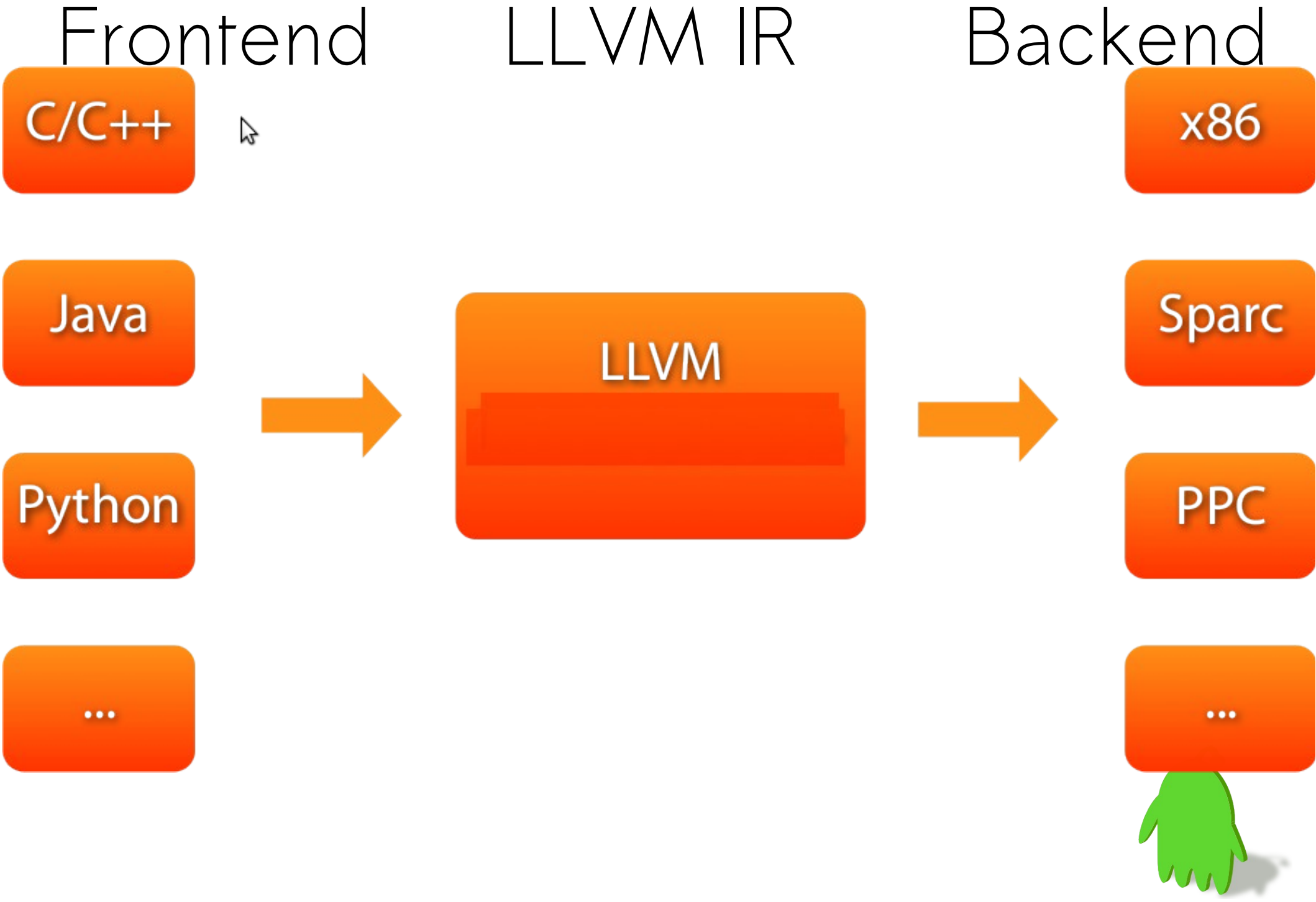
Compiler Driver

LLVM

C, C++, Obj-C

BSD-Style License

JIT/Interpreter


Frontend
LLVM IR
Backend
C/C++
Java
Python
...
LLVM
x86
Sparc
PPC
...

Frontend LLVM IR Backend

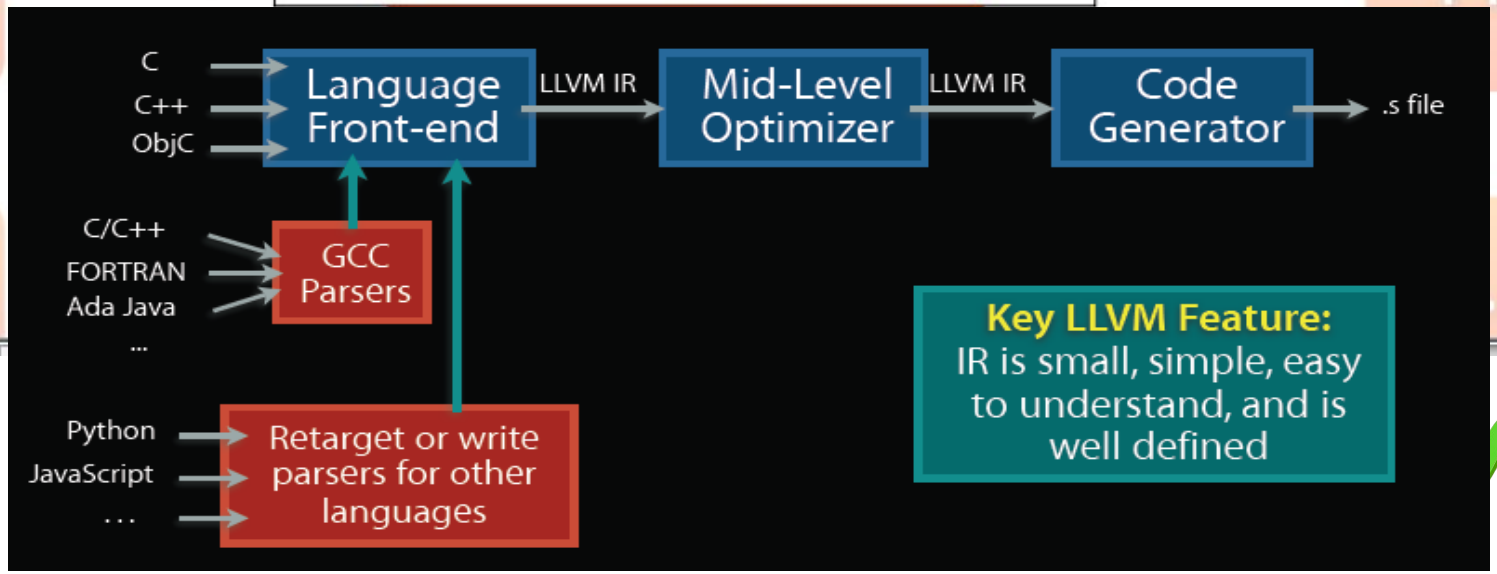
C/C++

x86

```
int add_func(
  int a, int b)
{
  return a + b;
}
```

```
define i32 @add_func(i32 %a, i32 %b) {
entry:
     %tmp3 = add i32 %b, %a
     ret i32 %tmp3
}
```

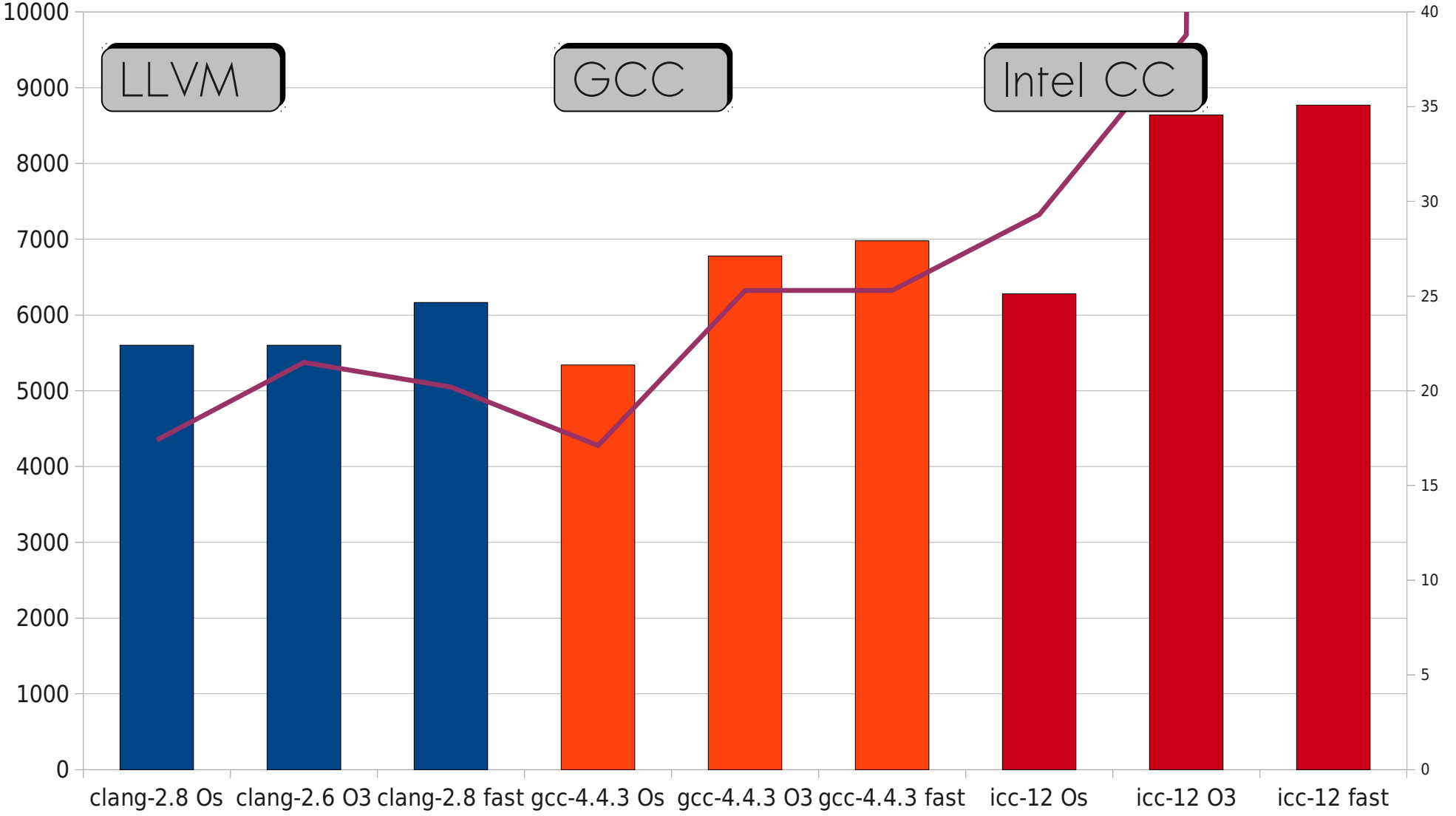
```
_add_func:
  movl  8(%esp), %eax
  addl  4(%esp), %eax
    ret
```



CodeMark 參考數據

Pentium4 3.0GHz Ubuntu: -Os -O3 -fast

size: kbytes

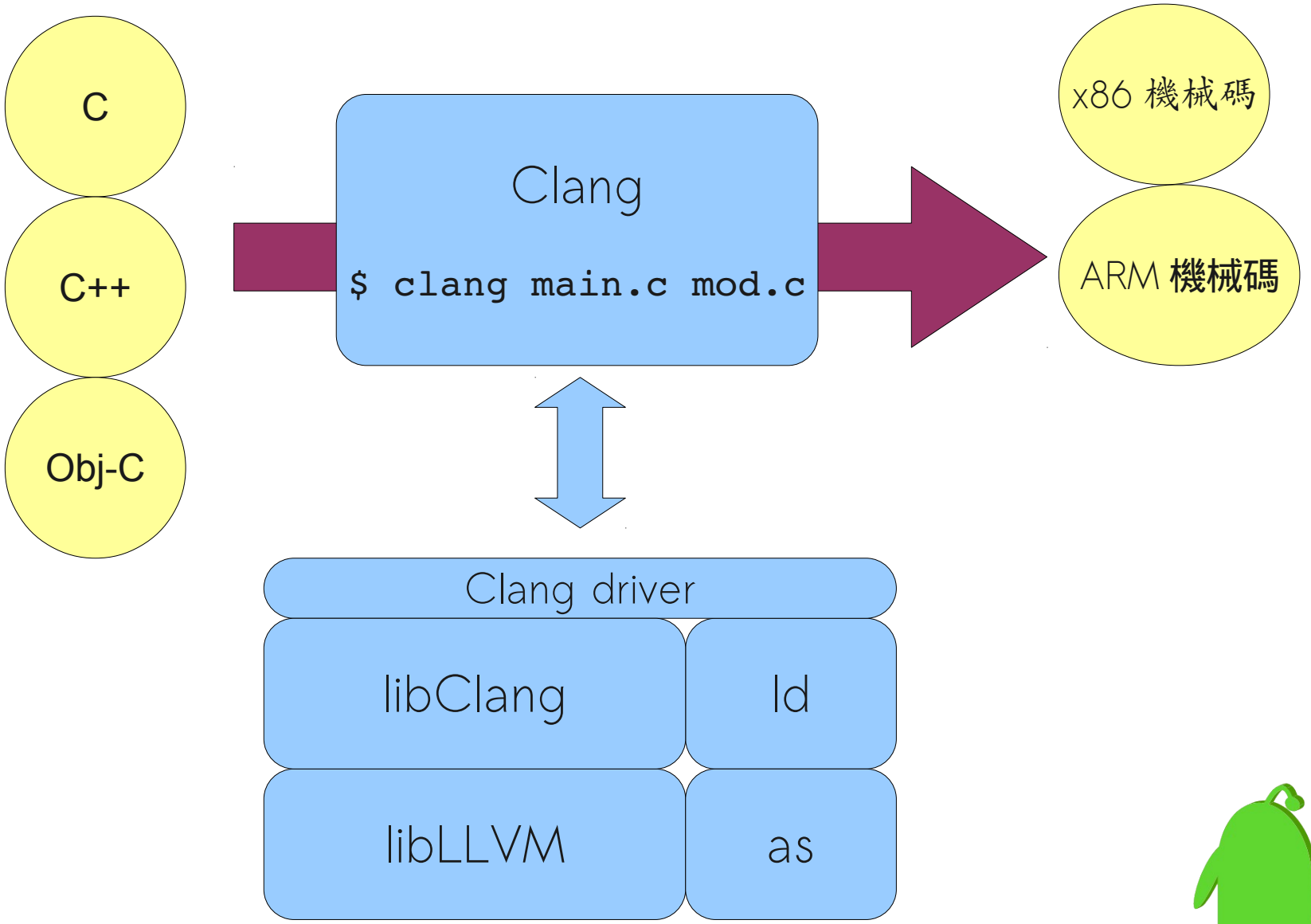


看起來，LLVM 效能還不是最好，
那為何我們還要關注?

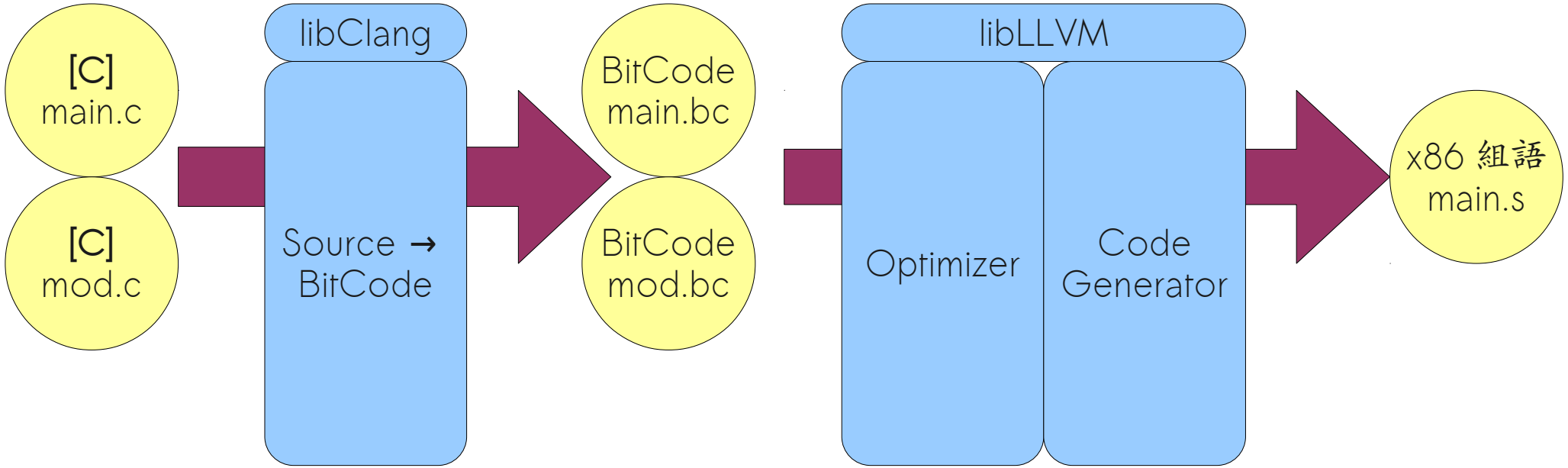
LLVM 不只是個具工業強度的編譯器

美妙的 LLVM + Clang

Clang: LLVM 的程式語言前端

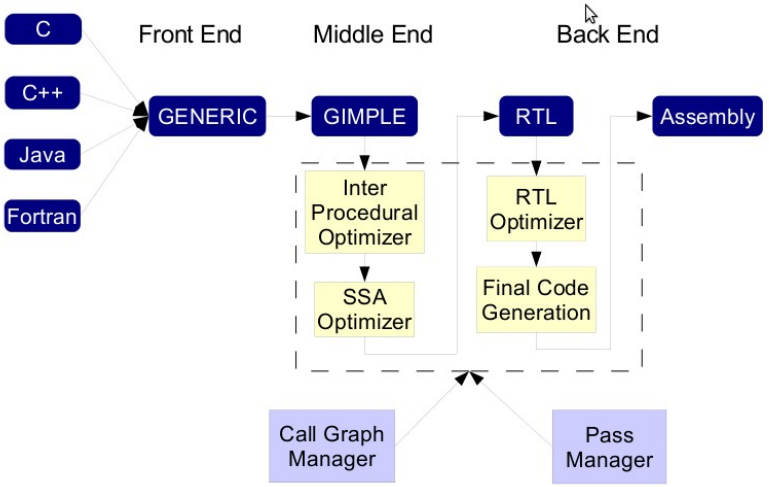


Clang 與 LLVM 的關聯

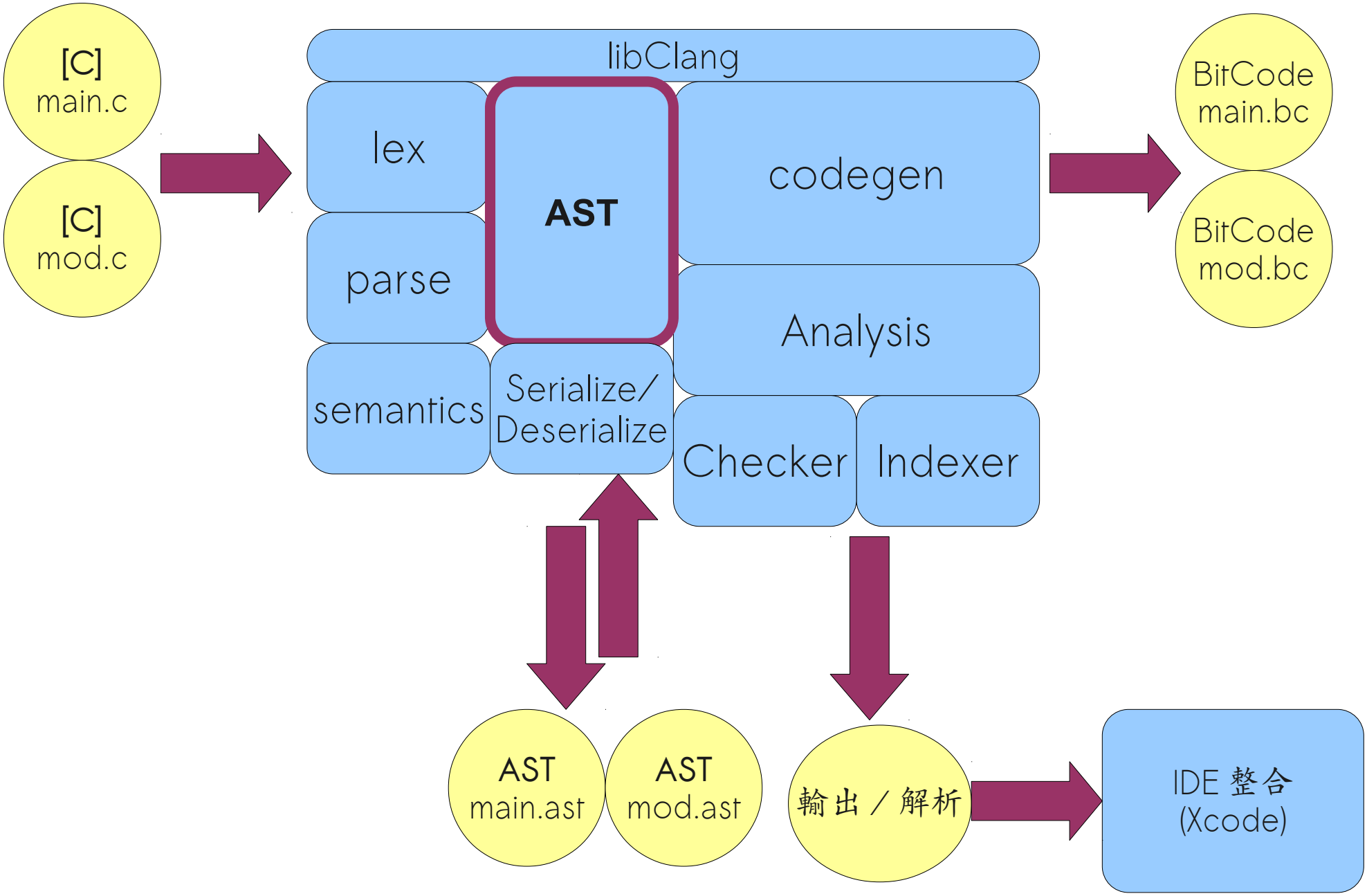


Compiler pipeline

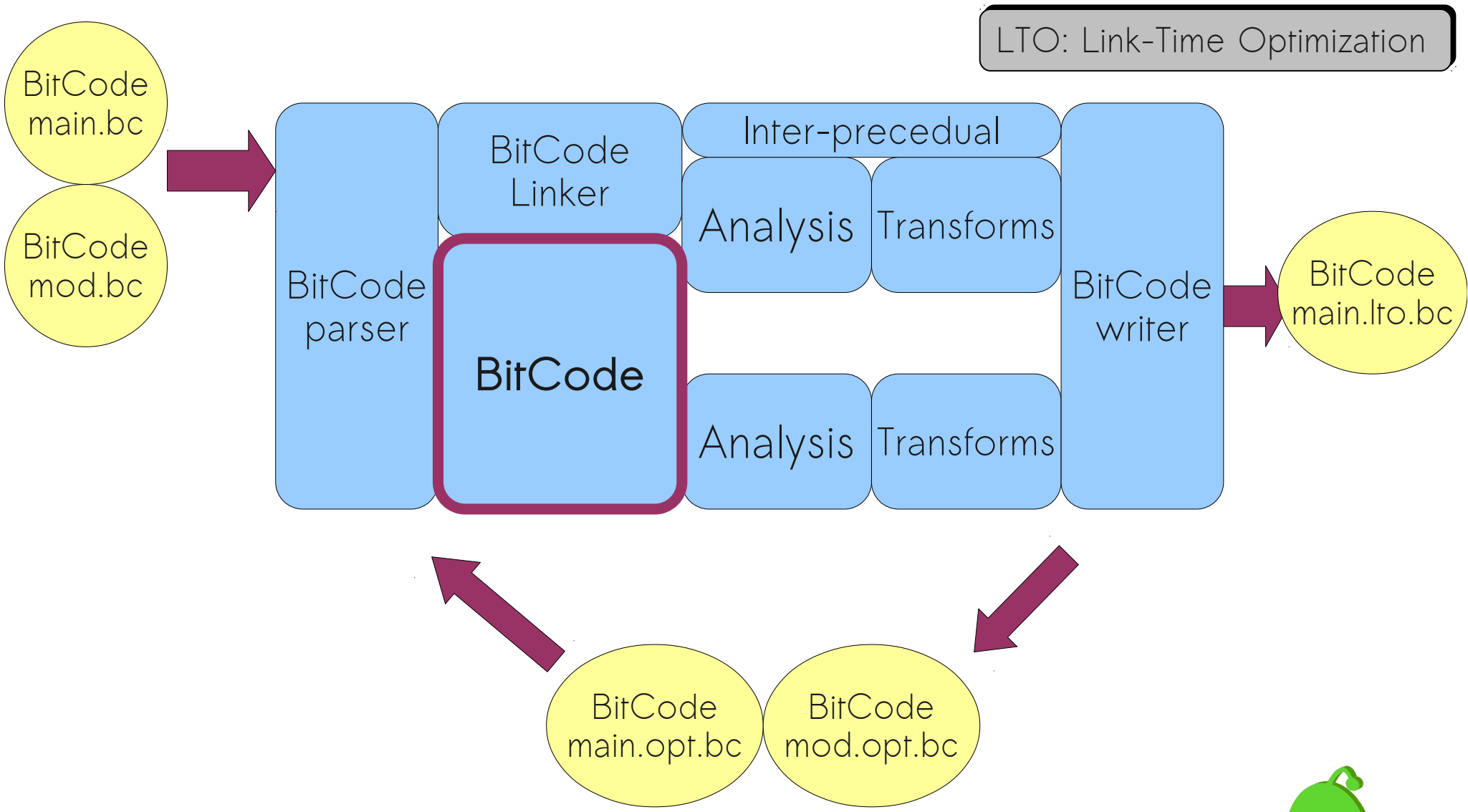
Google



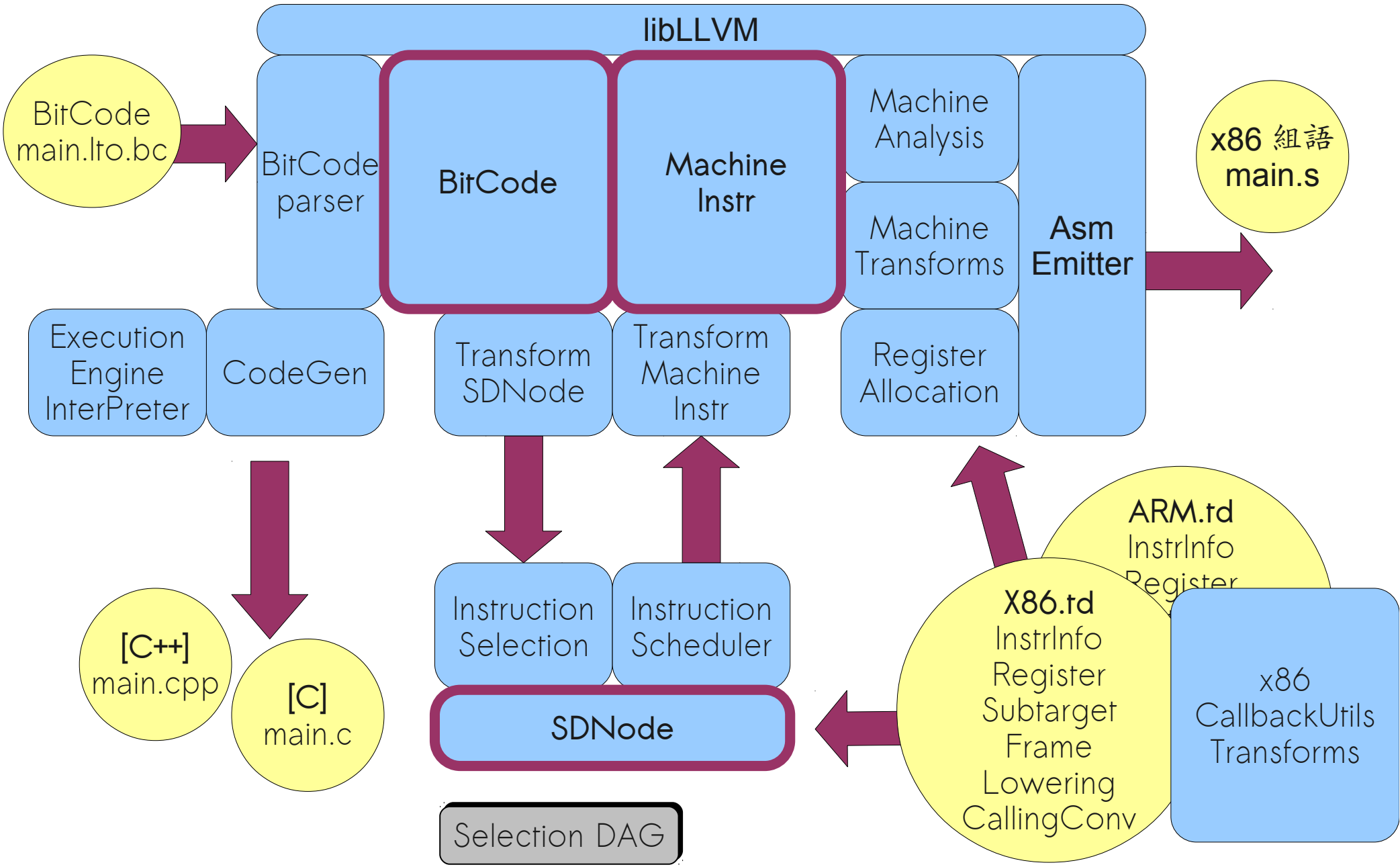
Clang 功能示意


[C]
main.c
[C]
mod.c
libClang
lex
parse
AST
semantics
Serialize/
Deserialize
codegen
Analysis
Checker
Indexer
BitCode
main.bc
BitCode
mod.bc
AST
main.ast
AST
mod.ast
輸出／解析
IDE 整合
(Xcode)

BitCode + Optimizer



LLVM Code Generation



先從 Hello World 開始

- 完整的 Compiler Driver

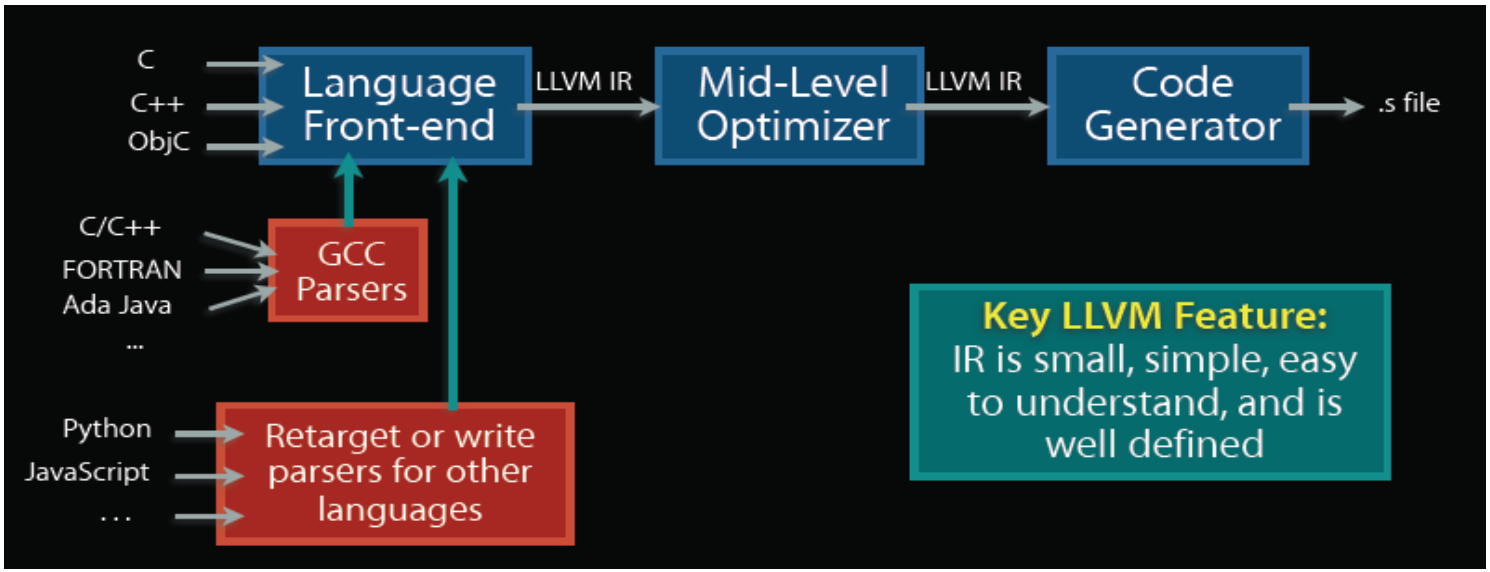
```
$ clang hello.c -o hello
```

- 生成 IR

```
$ clang -O3 -emit-llvm hello.c -c -o hello.bc
```

- 以 Just-In-Time compiler 模式執行 BitCode

```
$ lli hello.bc
```



Getting Started with the LLVM System
http://llvm.org/docs/GettingStarted.html

```c
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("Hello world!\n");
    return 0;
}
```

函式 printf() 後方僅有一個字串參數，前端預設將其轉換爲 puts()

- 反組譯 BitCode

```
$ llvm-dis < hello.bc
```

```
; ModuleID = '<stdin>'
target datalayout = "e-p:32:32:32-...
target triple = "i386-pc-linux-gnu"

@str = internal constant [13 x i8] c"Hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds ([13 x i8]* @str, i32 0, i32 0))
  ret i32 0
}

Declare i32 @puts(i8* nocapture) nounwind
```

- 輸出 x86 後端組合語言

```
$ llc hello.bc -o hello.s
```

LLVM in Google Android 3.0 SDK

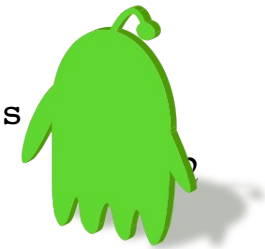
```
$ cd android-sdk-linux_x86/platform-tools
$ ./llvm-rs-cc --version
Low Level Virtual Machine (http://llvm.org/):
  llvm version 2.8svn
  Optimized build.
  Built Feb 16 2011 (19:26:29).
  Host: i386-unknown-linux
  Host CPU: penryn

  Registered Targets:
    arm    - ARM
    thumb  - Thumb
    x86    - 32-bit X86: Pentium-Pro and above
    x86-64 - 64-bit X86: EM64T and AMD64
$ ./llvm-rs-cc --help
OVERVIEW: RenderScript source compiler
```

Android SDK (3.0 = API version 11)
http://developer.android.com/sdk/

LLVM in Google Android 3.0 SDK

```
$ ./llvm-rs-cc --help
OVERVIEW: RenderScript source compiler
USAGE: llvm-rs-cc [options] <inputs>
OPTIONS:
  -I <directory>          Add directory to include search path
  -additional-dep-target <value>
                          Additional targets to show up in dependencies output
  -allow-rs-prefix        Allow user-defined function prefixed with 'rs'
  -bitcode-storage <value>
                          <value> should be 'ar' or 'jc'
  -emit-asm               Emit target assembly files
  -emit-bc                Build ASTs then convert to LLVM, emit .bc file
  -emit-llvm              Build ASTs then convert to LLVM, emit .ll file
  -emit-nothing           Build ASTs then convert to LLVM, but emit nothing
  -help                   Print this help text
  -java-reflection-package-name <value>
                        Specify the package name that reflected Java files belong to
  -java-reflection-path-base <directory>
                          Base directory for output reflected Java files
```



測試 SDK 內建範例 RenderScript

```
android-sdk-linux_x86/platform-tools$ ./llvm-rs-cc \
  ../samples/android-11/
RenderScript/HelloWorld/src/com/android/rs/helloworld/helloworld.rs \
  -I ../platforms/android-11/renderscript/include \
  -I ../platforms/android-11/renderscript/clang-include
```

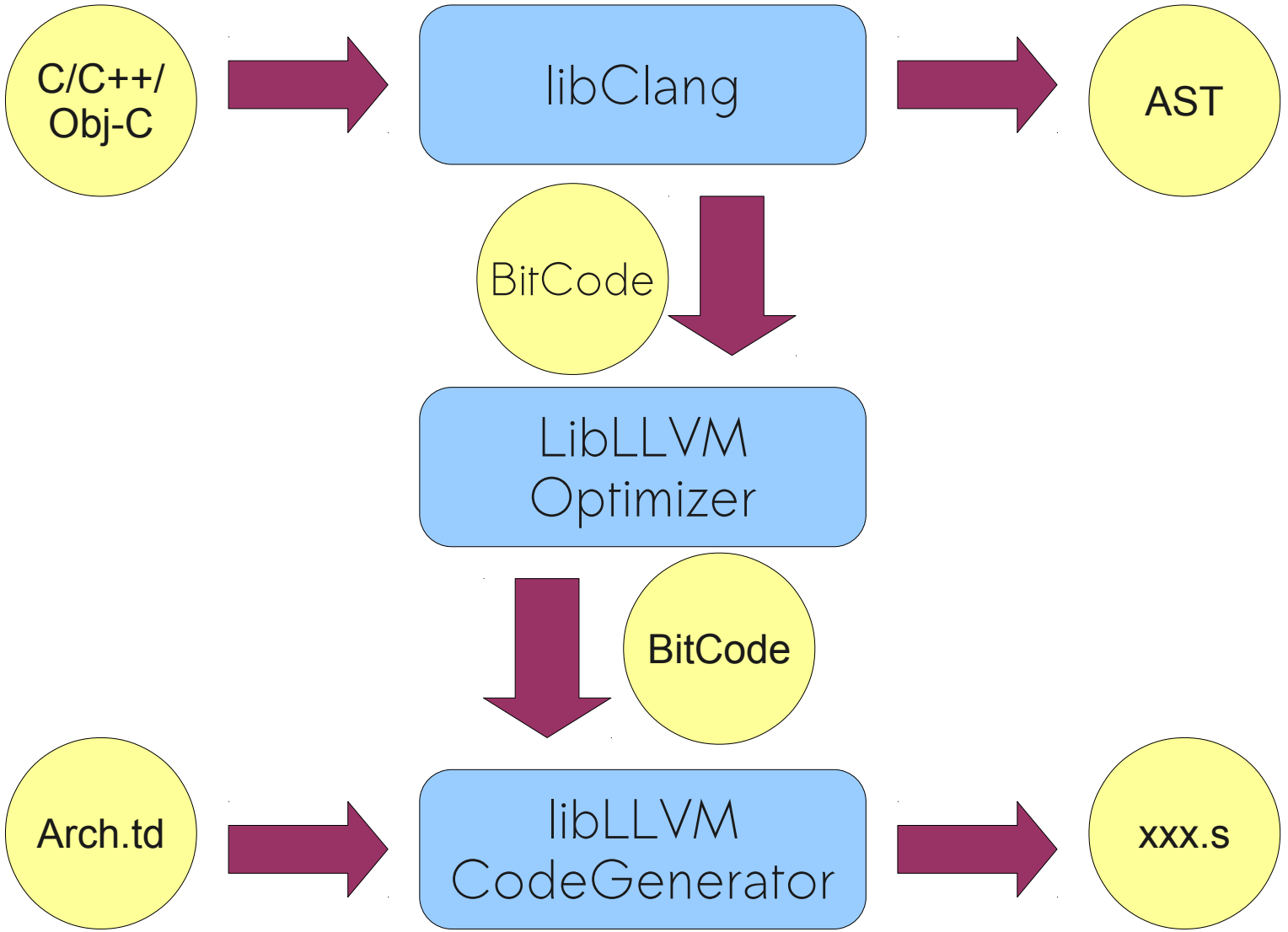
```
// helloworld.rs
// This is invoked automatically
// when the script is created
void init() {
    gTouchX = 50.0f;
    gTouchY = 50.0f;
}
```

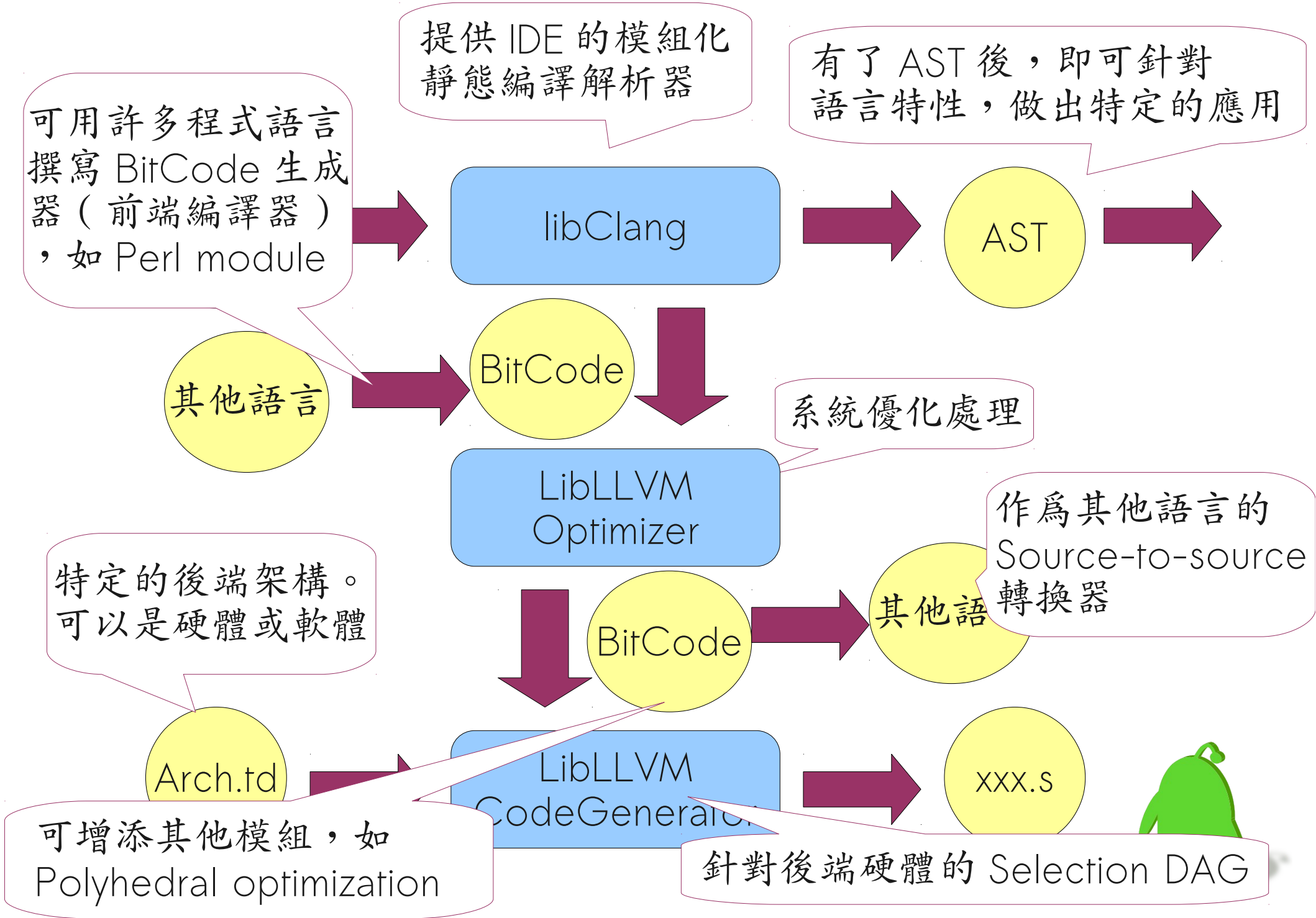
llvm-dis < helloworld.bc

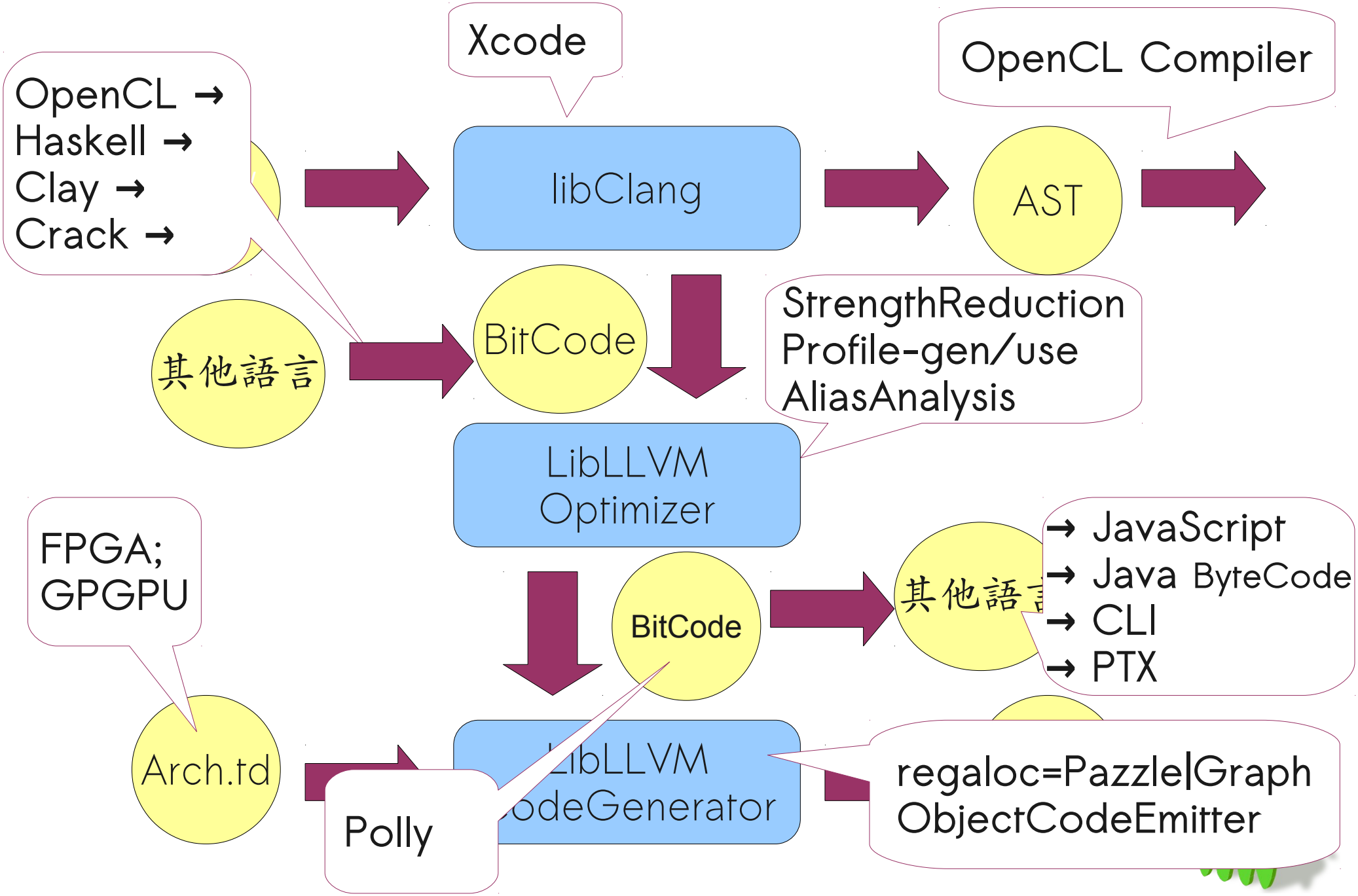
```
@gTouchX = common global i32 0, align 4
@gTouchY = common global i32 0, align 4

define void @init() nounwind {
  store i32 50, i32* @gTouchX, align 4
  store i32 50, i32* @gTouchY, align 4
  ret void
}
```

LLVM 給予無限可能


C/C++/
Obj-C
libClang
AST
BitCode
LibLLVM
Optimizer
BitCode
Arch.td
libLLVM
CodeGenerator
xxx.s


提供 IDE 的模組化
靜態編譯解析器
有了 AST 後，即可針對
語言特性，做出特定的應用
可用許多程式語言
撰寫 BitCode 生成
器（前端編譯器）
，如 Perl module
libClang
AST
其他語言
BitCode
系統優化處理
LibLLVM
Optimizer
作爲其他語言的
Source-to-source
轉換器
特定的後端架構。
可以是硬體或軟體
BitCode
其他語
Arch.td
LibLLVM
CodeGenerato
xxx.s
可增添其他模組，如
Polyhedral optimization
針對後端硬體的 Selection DAG


OpenCL →
Haskell →
Clay →
Crack →
Xcode
libClang
OpenCL Compiler
AST
其他語言
BitCode
StrengthReduction
Profile-gen/use
AliasAnalysis
LibLLVM
Optimizer
FPGA;
GPGPU
BitCode
→ JavaScript
→ Java ByteCode
→ CLI
→ PTX
Arch.td
Polly
regaloc=Pazzle|Graph
ObjectCodeEmitter

將 LLVM 應用在非編譯器領域

LLVM 力：

「不是爲了取悅硬體而寫編譯器，而爲自己寫編譯器」

- LLVM + Gallium3D: Mixing a Compiler With a **Graphics** Framework
  - http://people.freedesktop.org/~marcheu/fosdem09-g3dllvm.pdf

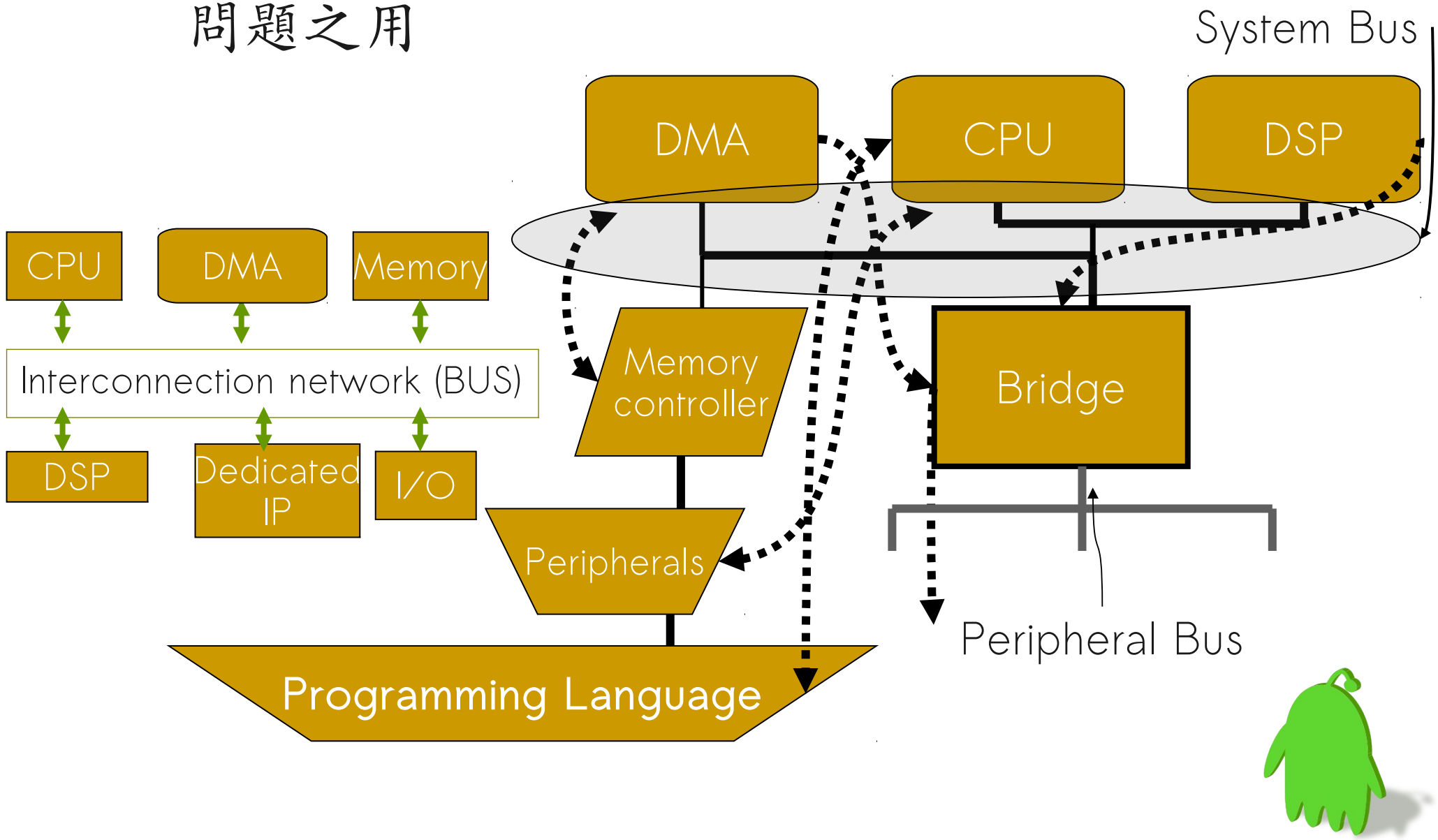
- Runtime Code Generation for **Huffman Decoders**
  - "The speedup improvement is **23.2%** at average and ranges from 32.2% to 14.2%."
    http://solar.cslab.ece.ntua.gr/~kkourt/papers/huff-jit-report.pdf

- A method for JIT'ing algorithms and data structures with LLVM
  - "For small AVL Trees (with less than ~3.000 nodes), we can get an average performance of **26%** over traditional method"
  - http://pyevolve.sourceforge.net/wordpress/?p=914

程式語言的變遷
(Low-Level 觀點)

程式語言的變遷(低階觀點)

- 職業無貴賤，程式語言是提出來解決人類面臨的問題之用



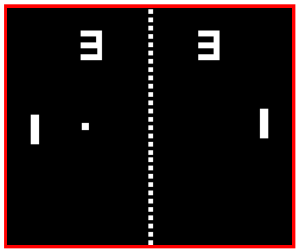
Indirection

"All problems in computer science can be solved by another level of **indirection**."

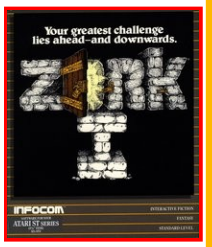
~ Butler Lampson, 1972 ~

- UNIX v6 (1976) 提供以 C **語言** 重寫的作業系統
- 需求驅使的發展模式
  - 數學 / 工程 → Lisp → Lisp machine (?)
  - 軟體工程 → Smalltalk
  - 網際網路 → Java

遊戲產業驅使 Programming Language



1972 Pong (硬體)



1980 Zork (高階直譯語言)

1993 DOOM (C)

1998 Unreal (C++, Java-style scripting)

2005-6 Xbox 360, PlayStation 3
with 6-8 hardware threads

2009 Next console generation.
Unification of the CPU, GPU.
Massive multi-core, data parallelism, etc.

今日的 Indirection 以 VM 形式存在

Charles Oliver Nutter (JRuby)

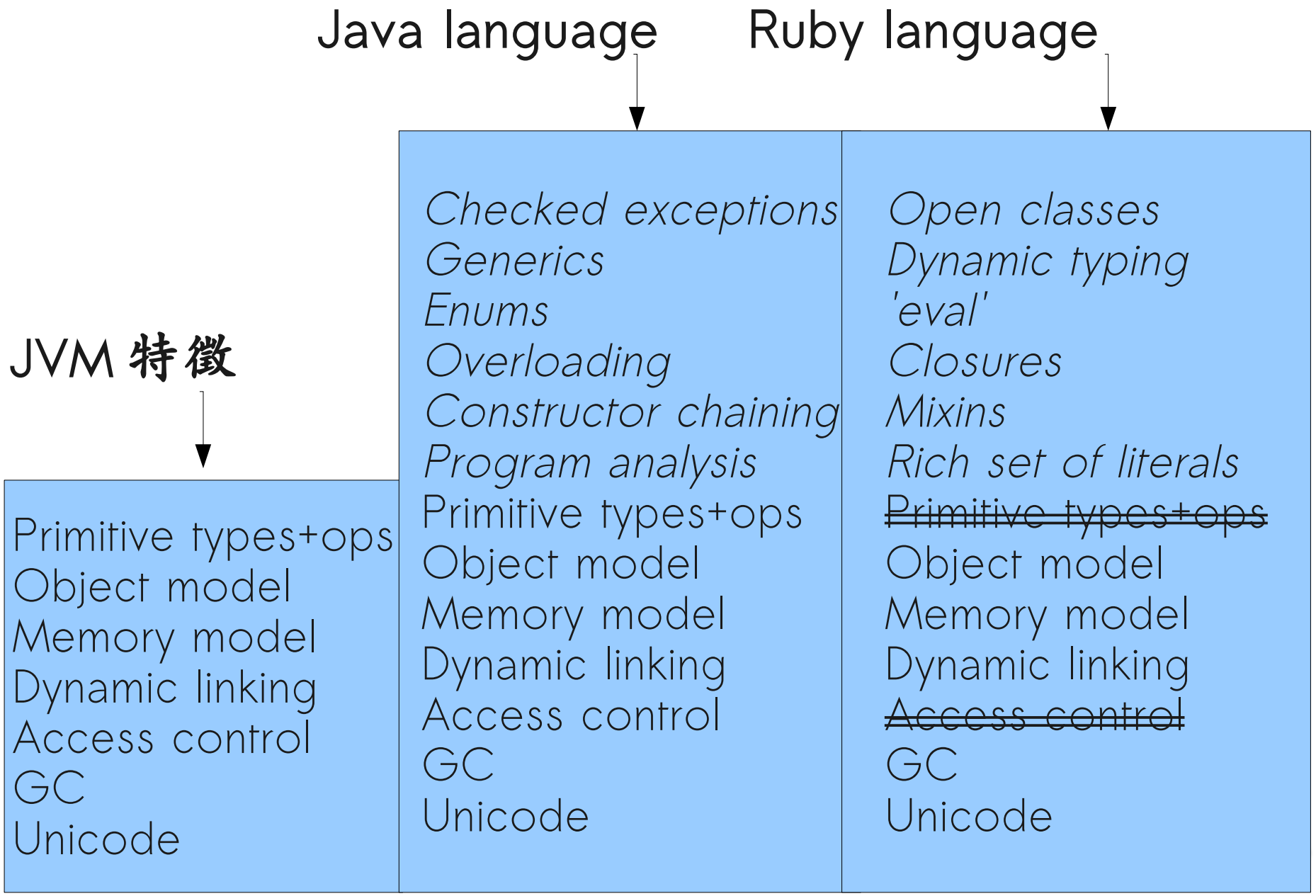
"**Building a Multilanguage VM**" (2009)

- Today, it is silly for a compiler to target actual hardware
  - Much more effective to target a VM
  - Writing a native compiler is lots more work!
- Languages need runtime support
  - C runtime is tiny and portable (and wimpy)
  - More sophisticated language runtimes need
    - Memory management
    - Security
    - Reflection
    - Concurrency control
    - Libraries
    - Tools (debuggers, profilers, etc)
- **Many of these features are baked into VMs**

Languages on the JVM

Sleep
WebL
Funnel
Tea
iScript
Bex Script
Jickle
Zigzag
JESS
Modula-2
Simkin
CAL
Lisp
JavaScript
Nice
Correlate
Basic
JudoScript
Simkin
Drools
Groovy
Eiffel
Luck
Icon
v-language
Pascal
Prolog
Mini
Scala
Tcl
PLAN
Hojo
Rexx
JavaFX Script
FScript
Tiger
Anvil
Yassl
foo
Oberon
E
Smalltalk
Logo
JHCR
JRuby
Tiger
Scheme
Ada
G
Clojure
Sather
Phobos
Dawn
Processing
TermWare
LLP
Pnuts
BeanShell
PHP
Forth
C#
SALSA
ObjectScript
Yoix
Jython
Piccola

JVM vs. Java Language vs. Ruby



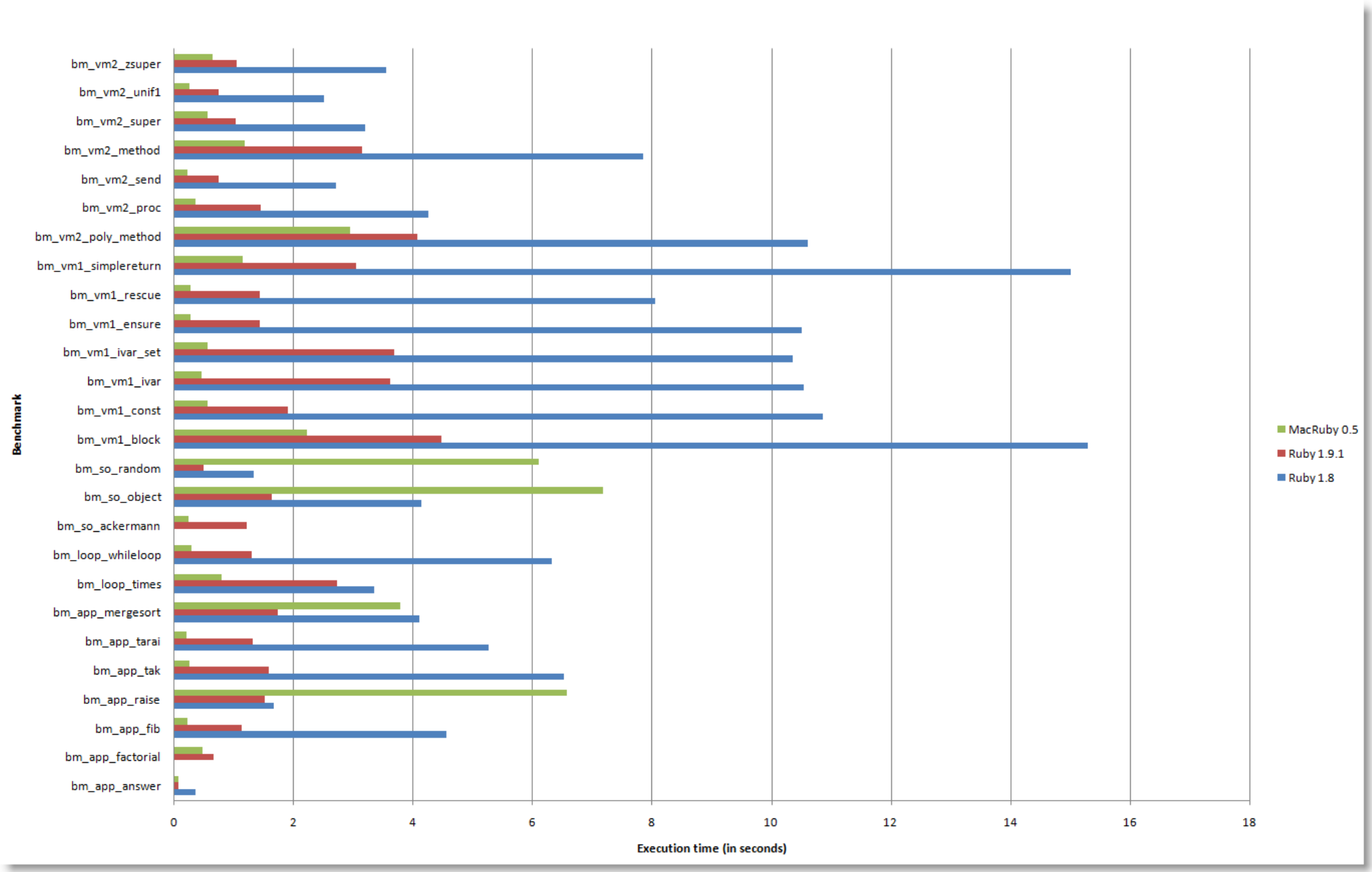
在 JVM 上實做 Ruby 語言 (JRuby)

- 好處：
  - 利用既有 JVM 在平台優化的效能與彈性
  - 存取豐富的 Java 資源
- 難處：
  - Dynamic typing 讓已有優化技術變得難以發揮
  - JRuby 必須維護自己的 type system
    - Reflection overhead
  - 無法以有效率且健全的方式來實做 Ruby "eval"

The Da Vinci Machine Project
http://openjdk.java.net/projects/mlvm/
OpenJDK Multi-language VM

在 LLVM 上實做 Ruby 語言

- 開放實做：
  - MacRuby : http://www.macruby.org/
  - Rubinius : http://rubini.us/
- 若充分對應到 LLVM 的設計，應可發揮若干效能的改善與 runtime support
  - Out-source'd JIT Runtime
  - Why MacRuby Matters (Present & Future)
    http://programmingzen.com/2009/03/29/why-macruby-matters/



http://programmingzen.com/images/macruby-large-chart.png

LLVM 與程式語言實做

- LLVM 在 bitcode 層面即考慮到動態語言的需求
- VMKit: Java, .Net
- 允許多個階段的優化 : profiler, offline optimizing
- 提供 Accurate Garbage Collection
- 在移動裝置上的 JIT compiler
  - [RenderScript] Android 3.0: 實做 code cache ，試圖降低編譯的成本
  - 目前 LLVM 的效能仍無法趕上若干特製的 VM 。案例： IcedTea/Shark, Dalvik

Build Programming Language Runtime with LLVM 實例

Brainfuck

- Brainfuck 是種極爲精簡的程式語言，由 Urban Müller 發展。當初的目標爲提出一種簡單的、可用最小的編譯器來實現、符合 Turing complete 的程式
- Brainfuck 僅有八個指令，其中兩個是 I/O 動作

Brainfuck

- 對應到 C 語言：若 char *p 指向記憶體區塊的話，Brainfuck 語言的八個指令可對照爲以下：

Brainfuck	C
`>`	`++p;`
`<`	`--p;`
`+`	`++*p;`
`-`	`--*p;`
`.`	`putchar(*p);`
`,`	`*p = getchar();`
`[`	`while (*p) {`
`]`	`}`

Brainfuck

- Brainfuck 語言

```
+++++[-]
```

- 等價於 C 語言

```
*p+=5;
while(*p != 0){
    *p--;
}
```

Brainfuck	C
>	++p;
<	--p;
+	++*p;
-	--*p;
.	putchar(*p);
,	*p = getchar();
[	while (*p) {
]	}

```
#include <stdio.h>
void foo(char c) { putchar(c); }
```

```
define void @foo(i8 signext %c) nounwind {
  %1 = alloca i8, align 1                         ; <i8*> [#uses=2]
  store i8 %c, i8* %1
  %2 = load i8* %1                                ; <i8> [#uses=1]
  %3 = sext i8 %2 to i32                          ; <i32> [#uses=1]
  %4 = call i32 @putchar(i32 %3)                  ; <i32> [#uses=0]
  ret void
}

declare i32 @putchar(i32)
```

```
// declare i32 @putchar(i32)
Function* putchar = cast<Function>(
    module->getOrInsertFunction(
        "putchar", voidType, cellType, NULL));
putchar->setCallingConv(CallingConv::C);
```

呼叫底層系統 libc 的 putchar 函式

```
Function* makeFunc(Module* module,
                   const char* source,
                   int tapeSize = 400) {
  ...
  // declare i32 @getchar()
  Function* getchar = cast<Function>(
     module->getOrInsertFunction("getchar", cellType, NULL));
  getchar->setCallingConv(CallingConv::C);

  // declare i32 @putchar(i32)
  Function* putchar = cast<Function>(
     module->getOrInsertFunction("putchar",
     voidType, cellType, NULL));
  putchar->setCallingConv(CallingConv::C);

  // Contruct void main(char* tape)
  Function* main = cast<Function>(
     module->getOrInsertFunction("main", voidType, NULL));
  main->setCallingConv(CallingConv::C);
  ...
```

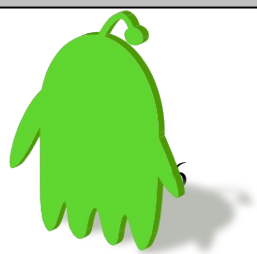
http://0xlab.org/~jserv/llvm/bf-llvm.cpp

```
Function* makeFunc(Module* module,
                   const char* source,
                   int tapeSize = 400) {
  ...
  Value* zero = ConstantInt::get(cellType, 0);
  Value* one = ConstantInt::get(cellType, 1);
  Value* minOne = ConstantInt::get(cellType, -1);
  ...

  BasicBlock* block =
      BasicBlock::Create(getGlobalContext(), "code", main);
  std::stack<bfLoopInfo> loops;
  IRBuilder<> codeIR(block);
  Value *head = codeIR.CreateAlloca(cellType,
      ConstantInt::get(indexType, tapeSize));
  Value *it = head;
  for (int i = 0; i < tapeSize; i++) {
    codeIR.CreateStore(zero, it);
    it = codeIR.CreateGEP(it, one);
  }
```

在 LLVM IR 中，預先若干定義的常數
zero = 0, one =1, minOne = -1

建立 LLVM IR



```
Function* makeFunc(Module* module,
                   const char* source,
                   int tapeSize = 400) {
  ...
  while(*source) {
    IRBuilder<> builder(block);
    switch(*source++) {
      case '>':
        head = builder.CreateGEP(head, one);
        break;
      case '<':
        head = builder.CreateGEP(head, minOne);
        break;
      case '+': {
        Value *headValue = builder.CreateLoad(head);
        Value *result =
            builder.CreateAdd(headValue, one);
        builder.CreateStore(result, head);
        break;
      }
      ...
```

將 brainfuck 轉成 LLVM IR

```
Function* makeFunc(Module* module,
                   const char* source,
                   int tapeSize = 400) {
    ...
    case '-': {
      Value *headValue = builder.CreateLoad(head);
      Value *result =
           builder.CreateSub(headValue, one);
      builder.CreateStore(result, head);
      break;
    }
    case '.': {
      Value* output = builder.CreateLoad(head);
      builder.CreateCall(putchar, output);
      break;
    }
    ...
```

稍早準備的 putchar 函式

```
Function* makeFunc(Module* module,
                   const char* source,
                   int tapeSize = 400) {
  ...
  // Close the function
  IRBuilder<> builder(block);
  builder.CreateRetVoid();
  return main;
}

int main(int argc, char* argv[]) {
  ...
  // Setup a module and engine for JIT-ing
  std::string error;
  InitializeNativeTarget();
  Module* module = new Module("bfcode", getGlobalContext());
  ExecutionEngine *engine = EngineBuilder(module)
    .setErrorStr(&error)
    .setOptLevel(CodeGenOpt::Aggressive)
    .create();
```

開啟進階優化的 ExecutionEngine (JIT)

```
Function *makeFunc(Module *module,
                   const char *source,
                   int tapeSize = 400) {
  ...
}

int main(int argc, char* argv[]) {
  ...
  // Compile the Brainfuck to IR
  std::cout << "Parsing..." << std::flush;
  Function* func = makeFunc(module, source.c_str());

  // Run optimization passes
  std::cout << "Optimizing..." << std::flush;
  FunctionPassManager pm(module);
  pm.add(new TargetData(
      *(engine->getTargetData())));
  pm.add(createVerifierPass());
  pm.run(*func);
  ...
```

讓 LLVM 串起整個編譯器架構

```
int main(int argc, char* argv[]) {
  ...
  // Compile
  std::cout << "Compiling..." << std::flush;
  void (*bf)() = (void (*)())
      engine->getPointerToFunction(func);
  std::cout << " done" << std::endl;

  // and run!
  bf();

  return 0;
}
```

讓 function pointer 指向經由 JIT 編譯過的機械碼

架構於 LLVM 的程式語言實做 (1)

- **Unladen Swallow** (Google): faster Python

  $ ./perf.py -r -b call_simple --args "-j always," \
  ../q2/python ../q3/python

  - Min: 1.618273 -> 0.908331: 78.16% faster
  - Avg: 1.632256 -> 0.924890: 76.48% faster

  http://code.google.com/p/unladen-swallow

- **GHC**/Haskell's LLVM codegen
  - 3x faster in some cases
    http://donsbot.wordpress.com/2010/02/21/smoking-fast-haskell-code-using-ghcs-new-llvm-codegen/
- **LLVM-Lua** : JIT/static Lua compiler
  - http://code.google.com/p/llvm-lua/

架構於 LLVM 的程式語言實做 (2)

- IcedTea Version of Sun's OpenJDK (RedHat)
  - Zero: processor-independent layer that allows OpenJDK to build and run using any processor
  - **Shark**: Zero's JIT compiler: uses LLVM to provide native code generation without introducing processor-dependent code.

    http://icedtea.classpath.org
- **Emscripten**
  - LLVM-to-JavaScript compiler
  - It takes LLVM bitcode and compiles that into JavaScript, which can be run on the web (or anywhere else JavaScript can run).

    http://code.google.com/p/emscripten/

http://0xlab.org